PC-PeerSim: Simulação Paralelizada de Overlays Peer-to-Peer em Clusters

João Manuel Parreira Neto

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Alberto Manuel Ramos da Cunha	DEI/IST
Orientador:	Luis Manuel Antunes Veiga	DEI/IST
Vogal:	João Manuel Santos Lourenço	FCT/UNL

Outubro 2010

Agradecimentos

Devo um agradecimento ao professor Luís Veiga e ao professor João Garcia que foram instrumentais para o sucesso desta tese.

Lisboa, 20 de Novembro de 2010

João Neto

“Simplicity is prerequisite for reliability.“

– Edsger Dijkstra

Resumo

Esta dissertação descreve a investigação, desenho e desenvolvimento de uma solução de simulação de uma rede sobreposta peer-to-peer no sistema Peersim.

Este projecto foi motivado pelos limites impostos pela versão actual do peersim que não faz uso de um processador *multi core* e cujo tamanho da simulação está limitado aos recursos de memória da máquina em que está a correr.

A solução para o problema passa por alterar a implementação do Peersim de modo a que este consiga efectuar processamento em paralelo (*multi threaded*). Como segundo objectivo, é utilizada a máquina virtual distribuída Terracotta para atingir a distribuição da execução da simulação sobre um cluster. Esta paralelização e distribuição tem que manter sincronização e coerência da simulação face à versão original.

Além das alterações feitas para que o simulador cumpra estes objectivos, contida neste documento, está também uma descrição detalhada da arquitectura e implementação da solução desenvolvida, passando pela nova API de protocolos, e pelas estratégias de partilha de memória na execução distribuída.

Os resultados apresentados neste documento comprovam a viabilidade da solução proposta para tornar a execução paralela. Em quase todos os cenários testados, os resultados demonstraram tempos de execução inferiores face à versão *single threaded*. Contrariamente, a versão distribuída mostrou-se altamente ineficiente, demonstrando o Terracotta como a abordagem errada para a resolução do problema.

A dissertação é terminada pela avaliação dos resultados obtidos e comparação dos mesmos com os dados obtidos usando a versão original, e conclusões e trabalho futuro derivados destes resultados.

Abstract

This dissertation describes the research, design and development of a solution for simulating peer-to-peer overlays in the Peersim system.
This project has its motivation in the imposed limits of the current version of peersim on not being able to take advantage of multi core processors and being severly limited by the available physical memory of the machine it's runing in, when defining the size of the simulation.
The solution to this problem begins with modifying the current version of Peersim in a way that it may use parallel (multi threaded) processing. As a secondary objective, we use the Terracotta distributed virtual machine to attain distributed execution over a cluster. These added features must maitain synchronization and the coherence inherent to the original version.
Besides these alterations, this document is comprised of a detailed description of the architecture and implementation of the developed solution, explaining the new protocol API and the memory sharing strategies used in the distributed execution.
The results presented here prove the viability and feasibility of the proposed solution to achieve parallel execution. In almost all the tested scenarios, the results showed faster execution times when compared to the single threaded version. In opposition, the distributed version revealed itself highly inneficient, demonstrating Terracotta as the wrong approach to solve the problem. This dissertation ends with a detailed description of the architecture and implementation of the developed solution and evaluation of the obtained results that prove the validity and usefulness of this project.

Palavras Chave
Keywords

Palavras Chave

Terracotta

Peer-to-peer

Grid

Simulação

Paralelismo

Overlay

Computação Distribuída

Keywords

Terracotta

Peer-to-peer

Grid

Simulation

Multithreading

Overlay

Distributed Computing

Índice

1 Introdução **1**

1.1 Introdução 1

1.2 Grid Computing 1

1.3 Peer-to-peer 3

1.4 Objectivo do Trabalho 4

1.5 Terracotta 4

1.6 Peersim 5

1.7 Estrutura deste Documento 5

1.8 Resumo 6

2 Trabalho Relacionado **7**

2.1 Introdução 7

2.2 Redes Peer-to-peer 7

2.2.1 Redes peer-to-peer estruturadas 9

2.2.2 Redes peer-to-peer não estruturadas 14

2.2.3 Conclusão 18

2.3 Sistemas Grid e Middleware para processamento paralelo distribuído 19

2.3.1 Projectos Correntes 20

2.3.2 Conclusão 23

2.4 Simulação de redes/overlays 24

2.4.1 Projectos Correntes 25

2.4.2 Conclusão 26

2.5 Resumo 27

3 Arquitectura da Solução **29**

3.1 Paralelização 29

3.1.1 Particionamento do *overlay* 32

3.2 Distribuição 33

3.3 Arquitectura 34

3.4 Resumo 37

4 Realização **39**

4.1 Particionamento 39

4.1.1 Largura-Primeiro *Single Threaded* 41

4.1.2 Aleatório 42

4.2 Processamento multithreaded sobre overlay 42

4.2.1 Motor *cycle-driven* 42

4.2.2 Motor *event-driven* 44

4.3 Sincronização da actividade entre nós 46

4.4 Execução distribuída 47

4.5 Sincronização da execução distribuída 48

4.6 Resumo 50

5 Avaliação dos Resultados **53**

5.1 Testes 53

5.2 Resultados da simulação paralelizada 55

5.2.1 *Event-driven*, Simulação exemplo 2 55
5.2.1.1 *Single Thread* 56
5.2.1.2 2 *Threads*, Particionamento Aleatório 57
5.2.1.3 2 *Threads*, Particionamento por Adjacência 57
5.2.2 Análise 57
5.2.3 *Event-driven*, Simulação Chord 58
5.2.3.1 *Single Thread* 59
5.2.3.2 2 threads, Particionamento Aleatório 60
5.2.3.3 4 threads, Particionamento Aleatório 60
5.2.4 Análise 60
5.2.5 *Event-driven*, Simulação Pastry 61
5.2.5.1 *Single Thread* 61
5.2.5.2 2 threads, Particionamento Aleatório 62
5.2.5.3 4 threads, Particionamento Aleatório 63
5.2.6 Análise 63
5.2.7 *Cycle-driven*, Simulação Exemplo 2 63
5.2.7.1 *Single Thread* 63
5.2.7.2 2 *Threads*, Particionamento Aleatório 64
5.2.7.3 2 *Threads*, Particionamento por Adjacência 66
5.2.8 Análise 66
5.3 Resultados da simulação distribuída 67
5.3.1 *Cycle-driven*, Simulação exemplo 2, PC-Peersim, 2 participantes na mesma máquina, Particionamento Aleatório 67
5.3.2 *Cycle-driven*, Simulação exemplo 2, PC-Peersim, 2 participantes em máquinas diferentes, Particionamento Aleatório 68

5.3.3 Análise 68

5.4 Resumo 69

6 Conclusão **71**

6.1 Resumo 71

6.2 Considerações globais de sucesso 73

6.3 Trabalho Futuro 74

7 Apêndice **i**

.1 Configuração Terracotta i

Lista de Figuras

2.1 Exemplo de tabela de vizinhos de um nó com identificador 67493 (Milojicic et al. 2002) . . . 10

2.2 Exemplo do reencaminhamento de uma mensagem de 0325 para 4598 (Stoica et al. 2001) . . . 11

2.3 Espaço cartesiano virtual onde os nós são endereçados . . . 12

2.4 Rede Viceroy com 4 níveis sendo a primeira linha o anel onde estão endereçados todos os nós . . . 13

3.1 Execução da simulação *Cycle-Driven* . . . 30

3.2 Execução da simulação *Event-Driven* . . . 31

3.3 Arquitectura geral do sistema . . . 35

4.1 Particionamento ideal . . . 40

4.2 Particionamento fragmentado . . . 41

4.3 Overlay separado por cores . . . 43

4.4 Evolução da *queue* . . . 44

4.5 *Queue* modificada para *multithreading* . . . 45

4.6 Estrutura original . . . 47

4.7 Estrutura alterada . . . 47

4.8 Execução distribuída . . . 51

5.1 Comparação de médias de execução para o exemplo 2 no motor *Event-Driven* para 250.000 nós . . . 55

5.2 Comparação de médias de execução para o exemplo 2 no motor *Event-Driven* para 2.500.000 nós 56

5.3 Comparação de médias de execução para o exemplo Chord no motor *Event-Driven* para 5.000 nós 58

5.4 Comparação de médias de execução para o exemplo Chord no motor *Event-Driven* para 250.000 nós 59

5.5 Comparação de médias de execução para o exemplo Pastry no motor *Event-Driven* para 50 nós 61

5.6 Comparação de médias de execução para o exemplo Pastry no motor *Event-Driven* para 5.000 nós 62

5.7 Comparação de médias de execução para o exemplo 2 no motor *Cycle-Driven* para 250.000 nós 64

5.8 Comparação de médias de execução para o exemplo 2 no motor *Cycle-Driven* para 2.500.000 nós 65

5.9 Comparação de médias de execução para o exemplo 2 no motor *Cycle-Driven* a correr uma simulação com 50 nós 67

5.10 Comparação de médias de execução para o exemplo 2 no motor *Cycle-Driven* a correr uma simulação com 500 nós 68

Lista de Tabelas

2.1 Comparação de overlays peer-to-peer estruturados 14

2.2 Comparação de overlays peer-to-peer não estruturados 18

2.3 Comparação de sistemas Grid 24

2.4 Comparação de simuladores peer-to-peer 26

Introdução

On 16 September 2007, Folding@home, a distributed computing network operating from Stanford University (USA) achieved a computing power of 1 petaflop – or 1 quadrillion floating point operations per second. The project uses the power of peoples' home computers, as well as their PlayStation3s, to simulate the processes inside living cells that can lead to diseases, such as Alzheimer's Disease.

– Guiness World Book of Records

1.1 Introdução

Desde que se criaram os primeiros mecanismos para programação em várias threads de execução que se tornou claro que a computação paralela seria uma ferramenta altamente valiosa para acelerar processos que outra forma seriam impraticáveis. O primeiro grande salto foi dado quando começaram a ser desenvolvidos os primeiros sistemas com vários CPUs, passou a ser possível efectuar computação paralela real por meio de hardware. Hoje em dia com as possibilidades dadas pela Internet e World Wide Web e a velocidade de ligações cada vez mais rápida, um sistema terá hipoteticamente acesso a um super computador virtual em constante crescimento, com um poder computacional semelhante a um recente super computador (ex.: IBM, Cray Inc., NEC). É neste contexto que se introduz o conceito de Grid Computing.

1.2 Grid Computing

A primeira definição para Grid Computing foi dada em meados dos anos 90 por Ian Foster como uma analogia para um sistema onde se pudesse aceder a recursos partilhados tão facilmente como podemos aceder a uma rede de electricidade (Power Grid (Druschel & 2002 2002)). Define-se por sistema Grid, uma infra-estrutura que permita e facilite a rápida partilha de recursos computacionais em larga escala para que estes estejam disponíveis para aplicações de

computação distribuída. Esta infra-estrutura deve conter ferramentas para a gestão de dados partilhados entre os vários intervenientes, planeamento, agendamento e gestão de pedidos de execução de operações sobre os dados (Foster 2002).

Desde o início que foi óbvia a utilidade de um sistema destes para suprir as largas necessidades computacionais de projectos científicos. Os projectos Folding@Home e Seti@Home são os proeminentes exemplos de computação pública que demonstram parte do potencial de um sistema Grid, funcionando em prol de causas científicas. Seguindo de novo a definição de Ian Foster, um sistema Grid deve permitir e facilitar a partilha de recursos em computadores ligados à Internet.

Uma evolução num sistema Grid, é a possibilidade de existir fora de um cluster ou de um conjunto de máquinas puramente dedicadas para a execução dos seus processos (por cluster entende-se um conjunto de máquinas sob administração comum, que poderão estar disponíveis para processamento paralelo local, para projectos instalados directamente nestas máquinas). O sucesso dos projectos Folding@Home e Seti@Home prende-se justamente com o facto de estes poderem fazer uso de computadores pessoais por todo o mundo, aproveitando os tempos ociosos (em que o CPU não está em uso intensivo) para efectuarem a recepção de dados e sua computação. O grosso dos cálculos acaba por ser feito e enviado para o servidor sem o utilizador do computador se aperceber disso, ou sem notar carga extra na máquina. Este tipo de aplicações que usam máquinas não dedicadas (mas sim voluntariadas) para efectuar as operações denominam-se de Cycle Sharing, na medida em que um indivíduo partilha os ciclos de CPU do seu computador para outras actividades. Nem todos os projectos Grid no entanto são apenas baseados em desktops pessoais, havendo vários que fazem uso dos recursos disponibilizados em um ou vários clusters.

Entende-se portanto, nesta visão alargada, por computação Grid, um sistema de computação distribuída em que várias máquinas partilham os seus recursos e efectuam trabalho coordenado por uma entidade orquestradora com vista a efectuar trabalho de elevada carga computacional, que à partida seria incomportável numa máquina apenas. A entidade central tem como tarefa principal a divisão do trabalho total em partes usáveis por um dos intervenientes (também conhecido como Task ou Job). Posto isto estamos em condições para definir concretamente os conceitos de computação paralela/distribuída, cluster e computação Grid:

- **Computação paralela/distribuída**: paradigma de computação em que um programa ou processo faz uso de mais do que uma unidade de processamento para aumentar a sua

performance. Para atingir isto, o programa/processo é dividido em várias partes capazes de ser processadas singularmente, e estas são executadas concorrentemente nas várias unidades de processamento da máquina em questão (CPUs multicore ou máquinas com mais do que um CPU) com memória partilhada ou em máquinas distintas capazes de comunicar em rede.

- **Cluster**: conjunto de computadores, integrados numa rede de alto débito e baixa latência, sob total controlo administrativo directo disponíveis para qualquer tipo de sistema de computação paralela/distribuída que os seus administradores decidam utilizar.

- **Computação Grid**: modelo de aplicações e infra-estrutura computacional baseado na agremiação de recursos computacionais altamente heterogéneos distribuídos para contribuição e resolução de um problema/tarefa definido.

Para se construir uma rede Grid é necessária uma infra-estrutura capaz de organizar os vários intervenientes (máquinas em partilha de recursos) e possibilite o fácil endereçamento, encaminhamento, entrada e saída de nós, e descoberta de recursos. Uma vez que a Internet não fornece suporte directo para este tipo de operação torna-se necessário introduzir uma camada de abstracção por cima do protocolo TCP/IP, denominada de rede Overlay (Overlay Network). Uma rede Overlay designa-se em poucas palavras por uma rede super imposta numa rede TCP/IP. É uma simples camada de abstracção por cima de uma rede que permite novas funcionalidades que seriam difíceis de conseguir de outra forma. As infra-estruturas comuns para computação Grid são infelizmente, pouco resistentes devido a todo o trabalho ser coordenado por uma entidade central. Com o objectivo de criar uma rede sem controlo centralizado que permita a execução de tarefas e a partilha de recursos/dados pelos seus intervenientes surge o conceito de overlay Peer-to-peer.

1.3 Peer-to-peer

Uma rede P2P define-se por uma infra-estrutura de rede descentralizada que permite facilidade na distribuição de dados ou recursos por várias máquinas (nós ou pares/peers) associados a um contexto, sem que haja portanto obrigatoriedade de organização central por parte de um servidor. As redes P2P estão hoje em dia muito associadas à partilha de ficheiros online, provando deste modo a viabilidade de um sistema distribuído com ênfase na eficiente pesquisa e

transmissão de dados, mas existem no entanto outras aplicações desta tecnologia, sendo exemplos disso software de conversação online ou software de armazenamento distribuído. Um rede P2P permite portanto a construção de um sistema organizado para transmissão e partilha de dados (ou de outros recursos com poder computacional) entre vários computadores heterogéneos (nós) sem necessidade de controlo central, em que todos os nós podem participar activamente, sendo estes capazes de resistir a falhas nos seus participantes e manter conectividade sem necessidade de intervenção de um agente coordenador.

1.4 Objectivo do Trabalho

É no contexto destas tuas tecnologias que se insere o trabalho a efectuar nesta tese de mestrado. O objectivo do trabalho consistem em implementar um motor que permita que uma aplicação desenhada para ser executada com base num overlay peer-to-peer possa ser executado num ambiente simulado mas com a possibilidade da mesma simulação correr num ambiente de execução paralela num computador multicore/multicpu, ou num cluster com mais do que uma máquina (ao invés de correr apenas sequencialmente num computador). É portanto criada uma camada de abstracção entre um overlay peer-to-peer completamente simulado (em que não existe uma ligação directa entre um nó do overlay e uma máquina física) e um sistema Grid que parte do overlay peer-to-peer e distribui a sua carga computacional por várias máquinas.
A modularidade e portabilidade do Java permite-nos atingir este objectivo com mais segurança pois esta linguagem está implementada nos mais variados sistemas operativos e suporta um conjunto bastante heterogéneo de hardware, e permite-nos acima de tudo fácil integração com o simulador de overlays peer-to-peer "PeerSim", implementado em Java.
Resumindo, o objectivo principal deste trabalho é fazer com que uma simulação consiga fazer uso de recursos de memória e CPU disponibilizados por um ou vários participantes para se poder atingir tempos de simulação menores, o que se traduz inevitavelmente numa maior eficiência.

1.5 Terracotta

Como ferramenta essencial na construção na camada de distribuição da aplicação surge o Terracotta. Com a emergência destas tecnologias Grid e Peer-to-Peer e a proeminência do Java como

a principal linguagem de programação completamente grátis e aberta (e com implementação para vários tipos de sistema operativo e hardware), começaram a surgir ferramentas para poder executar aplicações Java de uma forma distribuída por uma rede de várias máquinas. O Terracotta é constituído por uma máquina virtual java baseada numa arquitectura cliente/servidor em que é possível executar programas feitos originalmente para uma só máquina num cluster, sem haver necessidade de alterar a sua implementação, ou adicionar código.

1.6 Peersim

Um simulador peer-to-peer consiste numa aplicação que simule uma rede peer-to-peer com um número de nós arbitrário numa só máquina, e que permite a aplicação de protocolos e operações sobre esses mesmos nós. Este software permite simular o comportamento de um algoritmo ou estrutura de código numa rede de computadores, sem ter que efectivamente ser feita a instalação de várias máquinas e de uma rede física. O Peersim é um simulador de overlays peer-to-peer desenhado em Java e que tem à partida à sua disposição a implementação de vários protocolos, além de ter maior escalabilidade do que todos os outros simuladores disponíveis em Java, conseguindo suportar com razoável eficiência até 1 milhão (10^6) de nós simulados (Naicken et al. 2006)(Naicken et al. 2007). É facilmente configurável na medida em que é possível escrever novos protocolos de peer-to-peer de modo a acomodar qualquer tipo de problema.

1.7 Estrutura deste Documento

No restante deste documento apresentamos, no cap. 2, um levantamento e classificação dos principais projectos relativos às tecnologias apresentadas nesta introdução (overlays peer-to-peer, computação em Grid e simulação). Estes projectos são comparados segundo um conjunto de parâmetros que definem as suas características. De seguida é descrita a arquitectura (cap. 3) de solução proposta e sua implementação (cap. 4). No cap. 5 é apresentada a avaliação da solução proposta, sendo o documento terminado com a conclusão e estimativa de trabalho futuro (cap. 6).

1.8 Resumo

Neste capítulo foi feita uma introdução aos conceitos de peer-to-peer e computação Grid incluindo o seu enquadramento histórico. De seguida é introduzido o conceito de estudo e simulação de *overlays peer-to-peer* qual a motivação e origem deste trabalho. Por fim são enumerados os objectivos do mesmo.

2 Trabalho Relacionado

2.1 Introdução

Neste capítulo vamos abordar vários projectos correntes nas áreas abrangidas por esta tese:

- Redes *peer-to-peer*: Quais os protocolos *peer-to-peer* (estruturados e não estruturados) mais comuns e interessantes de estudar e classficar.
- Sistemas Grid e Middleware para processamento paralelo distribuído: Alguns exemplos de sistemas Grid público e institucionais, e middleware para a construcção dos mesmos.
- Simuladores de redes/*overlays*: Exemplos de ferramentas de simulação de redes físicas e redes sobrepostas.

2.2 Redes Peer-to-peer

As várias implementações diferentes de redes *peer-to-peer* podem ser categorizadas segundo o seu grau de centralização e se a rede é ou não estruturada. Uma rede peer-to-peer estruturada mantém em execução uma estrutura semelhante a uma *Hash Table* distribuída onde mantém informação sobre todos os intervenientes da rede de forma a manter a sua localização constantemente disponível e mantém também catalogados todos as chaves de modo a que seja rápida a sua localização no *overlay*. Uma chave representa algo pelo qual um nó é responsável (numa aplicação de partilha de dados, uma chave poderá representar um ficheiro ou um bloco de dados único, num servidor de DNS uma chave representará um par nome/endereço único, numa biblioteca poderá representar um artigo, etc.). A maneira como esta estrutura é mantida depende de implementação para implementação, pois a maneira como os nós contactam uns com os outros varia, sendo que normalmente cada nó só conhece um conjunto limitado de outros nós na rede. A cada nó, é atribuído um identificador único na rede, e a cada chave é também é atribuída um identificador único. Isto permite a construção de um grafo de toda

a rede em que rapidamente se identifique qual o nó responsável por determinada chave. A manutenção de uma rede estruturada é bastante complexa, pelo que é complicado e ineficiente manter coerência quando existe um grande volume de nós a entrar e a sair da rede com o seu conjunto de chaves, pelo que quase todas as populares são redes de partilhas de dados não são estruturadas. Uma vez que todos os *overlays peer-to-peer* estruturados são completamente descentralizados só se torna relevante analisar o grau de centralização em redes não estruturadas. Este é definido pela necessidade de existir uma semi-coordenação central na rede. Semi-coordenação, porque havendo total coordenação por parte de um agente servidor (na medida em que todas as operações são controladas por um servidor central), a semântica por trás de uma rede *peer-to-peer* deixa de estar associada ao sistema em questão. Um *overlay peer-to-peer* pode ser portanto:

- **Completamente descentralizado.** Todos os nós têm o mesmo estatuto na rede na rede. Esta alternativa tem a desvantagem de não haver um mínimo de controlo sobre o estado total da rede, e limita as opções para uma máquina se ligar ao overlay, pois este tem que conhecer um participante arbitrário na rede para se poder juntar à rede.

- **Parcialmente descentralizado ou híbrido**. Existem alguns nós denominados de super-peers que coordenam as entradas e saídas da rede, mas que por si só, apenas um nó pode não ter conhecimento da rede inteira.

- **Completamente centralizado.** Existe um servidor central que controla todas as entradas e saídas da rede e que pode ou não ter conhecimento dos objectos ou blocos de dados contidos nos nós. A desvantagem neste desenho é óbvia. O servidor central é um ponto de falha para toda a rede. (Milojicic et al. 2002)

Portanto, à semelhança de (Lua et al. 2005) a seguinte lista de trabalho efectuado na área peer-to-peer será classificado segundo a sua centralização e estruturação sendo também comparados os seguintes aspectos:

- **Aplicação**, no que toca a qual a utilização final dos programa.

- **Tempo de Procura**, dado por uma função do número de nós da rede e outras variáveis da rede (apenas nas redes estruturadas)

- **Tipo de Procura**, descrição simples do algoritmo de lookup (ex.: procura no servidor, procura no super-peer, procura por flood, etc.) (apenas nas redes não estruturadas)

Os projectos seguintes serão apresentados com a seguinte classificação: **Nome do Projecto [Estruturação, Centralização, Aplicação, Tempo de Procura / Tipo de Procura]**.

2.2.1 Redes peer-to-peer estruturadas

Chord [Estruturado, Completamente descentralizado, Uso geral, $O(\log N)$**].** O Chord é um sistema de lookup de chaves em que os nós estão organizados num espaço de identificadores circular. Cada um dos nós é representado por uma identificador, derivada de uma função de hashing sobre o seu endereço. Os nós são ordenados pelo seu identificador num anel virtual (chord ring) em que um nó x é designado de sucessor de y se o seu identificador for igual ou imediatamente a seguir ao identificador de x. Chegando ao fim do espaço de endereçamento de nós, o sucessor é o nó cujo identificador é igual ou imediatamente a seguir a 0. Nenhum nó conhece a totalidade da rede, mantendo informação apenas um número máximo de identificadores em memória (mais especificamente log(N) para N igual ao número de peers na rede). Este desenho tem a clara vantagem de permitir entradas e saídas de nós sem haver necessidade de propagar a alteração por toda a rede (apenas os nós vizinhos precisam de alterar o seu estado quando ocorre uma saída ou uma entrada). A cada chave é atribuído também um identificador sendo uma função de hash sobre a própria chave. Cada nó é responsável por aproximadamente o mesmo número de chaves sendo que uma chave k é armazenada no nó cujo identificador seja igual a k ou imediatamente seguinte chamando-se o nó sucessor da chave k. Quando um nó n entra na rede, o seu nó sucessor transfere-lhe algumas chaves que estavam sob seu controlo mas que agora pertencem a n. Quando um nó sai da rede, todas as suas chaves são transmitidas para o seu sucessor. Quando um peer precisa de localizar uma chave envia uma mensagem ao seu vizinho, que será propagada até chegar a um nó com identificador igual ou superior ao identificador da chave. O endereço do nó é depois devolvido ao remetente. Se a chave estiver armazenada no predecessor do peer, seria necessário percorrer todos os nós da rede para chegar à chave pretendida. Para isso, o chord faz uso de uma tabela denominada de finger table. Esta tabela contém m entradas, sendo m o número de bits que existem no espaço de identificadores da rede. Sendo k o identificador do peer, as entradas da finger table correspondem aos nós onde estão armazenadas as chaves de k+20 até k+2m.

Usando esta tabela, o peer pode procurar directamente o nó imediatamente abaixo da chave pretendida e começar o lookup a partir daí, optimizando o processo. (Androutsellis-theotokis & Spinellis 2004)(Stoica et al. 2001)

Tapestry [Estruturado, Completamente descentralizado, Uso geral, $O(\log_B N)$]. O Tapestry é um sistema de lookup semelhante ao Chord na medida em que os nós estão organizados num espaço distribuído de identificadores mas com um mecanismo de lookup completamente diferente. Este sistema usa uma variante da técnica de procura Plaxton para localizar o nó correspondente à chave pretendida. Cada nó mantém uma tabela de endereçamento dos seus vizinhos, com vários níveis, correspondentes ao número de dígitos do maior identificador possível. Cada nível armazena x entradas (Sendo x a base dos endereços de endereçamento. x é igual a 10 caso os endereços sejam decimais, 2 caso sejam binários, 16 caso sejam hexadecimais, etc).

	Level 5	Level 4	Level 3	Level 2	Level 1
Entry 0	07493	x0493	xx093	xxx03	xxxx0
Entry 1	17493	x1493	xx193	xxx13	xxxx1
Entry 2	27493	x2493	xx293	xxx23	xxxx2
Entry 3	37493	x3493	xx393	xxx33	xxxx3
Entry 4	47493	x4493	xx493	xxx43	xxxx4
Entry 5	57493	x5493	xx593	xxx53	xxxx5
Entry 6	**67493**	x6493	xx693	xxx63	xxxx6
Entry 7	77493	x7493	xx793	xxx73	xxxx7
Entry 8	87493	x8493	xx893	xxx83	xxxx8
Entry 9	97493	x9493	xx993	xxx93	xxxx9

Figura 2.1: Exemplo de tabela de vizinhos de um nó com identificador 67493 (Milojicic et al. 2002)

Pastry [Estruturado, Completamente descentralizado, Uso geral, $O(\log_B N)$]. O pastry é um overlay com um funcionamento muito semelhante ao Tapestry na medida em que funciona também com uma procura baseada na técnica de Plaxton, mas que no entanto usa uma tabela de vizinhança bastante mais complexa (continuando a ser baseada no entanto na separação dos vários dígitos do identificador de um nó. A construção da tabela de vizinhança deixa de ser

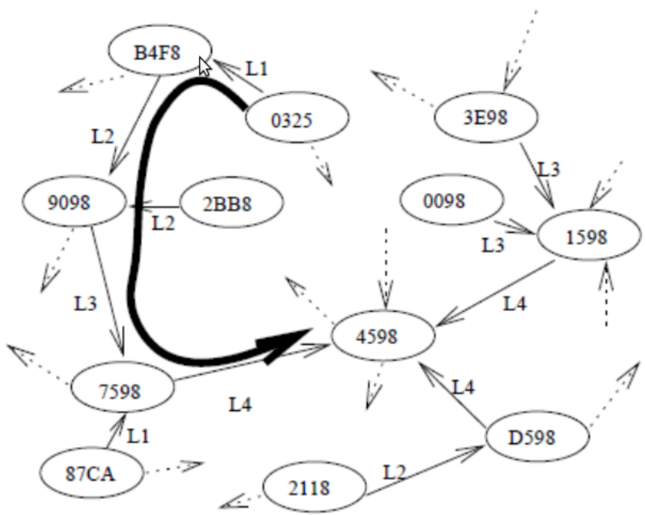


Figura 2.2: Exemplo do reencaminhamento de uma mensagem de 0325 para 4598 (Stoica et al. 2001)

baseada na base dos endereços mas sim no número de bits da base dos endereços. A procura é efectuada primeiro pelo prefixo do endereço e depois pela chave funcionando de modo semelhante ao sistema Chord, na medida em que os identificadores têm que estar também organizados num espaço de endereçamente circular. Este sistema é mais eficiente na medida em que na maior parte dos casos consegue fazer o lookup de um nó na rede em menos do que O(logBN) ”saltos’(Lua et al. 2005) sendo B a base dos identificadores(Androutsellis-theotokis & Spinellis 2004)(Rowstron & Druschel 2001)

Kademlia [Estruturado, Completamente descentralizado, Uso geral, $O(\log_B N) + c$]. O sistema Kademlia difere-se dos anteriores na medida em que é baseado numa métrica de distância entre os seus nós. Esta distância é medida usando uma operação lógica XOR (eXclusive OR) entre os bits que representam os identificadores. Esta distância é usada para efectuar a procura de uma determinada chave. Cada nó mantém uma tabela com 160 níveis (correspondentes aos 160 bits da chave), contendo cada nível uma lista de nós cujo valor da função de distância está entre 2i e 2i+1. Esta lista é ordenada pela última visita feita aos nós em questão estando em primeiro lugar os nós visitados à menos tempo. Quando é efectuada a procura de uma chave o nó escolhe X nós da lista, calcula a sua distância e envia pedidos de procura da chave em paralelo e de forma assíncrona para eles repetindo o processo. O nó de destino é o que apresentar a distância mais curta. (Lua et al. 2005)(Maymounkov & Mazières

2002)

CAN [Estruturado, Completamente descentralizado, Uso geral, $O(d.N1/d)$]. O CAN é um overlay peer-to-peer cujos nós se encontram organizados segundo um espaço cartesiano de n dimensões (comparativamente ao anel circular Chord e a malha Plaxton do Tapestry). A cada nó é atribuído um ponto no espaço de endereçamento cartesiano. O espaço total de endereçamento é particionado de modo a que cada nó ocupe uma "zona" cujo centro são as coordenadas do mesmo ponto. Assim sendo, dependendo do número de dimensões cartesianas na configuração do sistema, cada nó vai ter um conjunto de zonas adjacentes ocupadas que serão os vizinhos deste nó. (Paul et al. 2001)

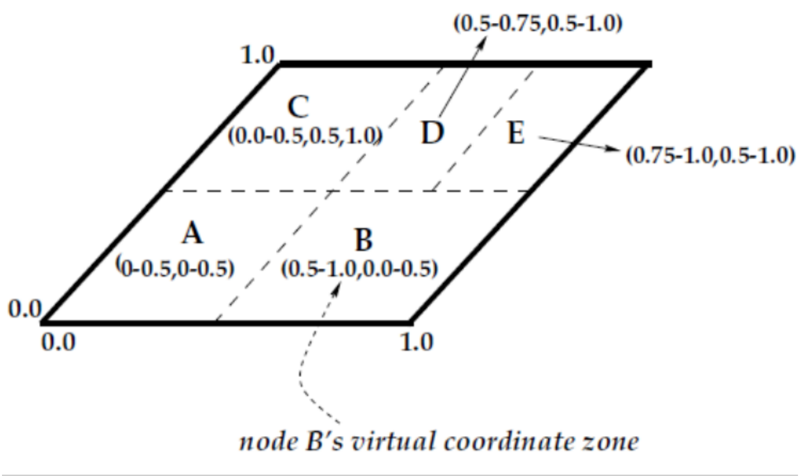


Figura 2.3: Espaço cartesiano virtual onde os nós são endereçados

Viceroy [Estruturado, Completamente descentralizado, Uso geral, $O(\log N)$]. A rede Viceroy foi concebida de modo a aproveitar vários conceitos de desenhos anteriores de modo a encontrar uma solução óptima combinando com as vantagens de uma rede Butterfly. Os nós são mapeados num anel virtual (semelhante ao usado no Chord, em que uma chave é armazenada no nó cujo identificador é igual ou imediatamente seguinte ao identificador da chave). Além do seu identificador (que é um decimal atribuído aleatoriamente entre 0 e 1), um nó também é associado a um nível que é escolhido aleatoriamente entre 1 e log N sendo N uma estimativa do total de nós na rede. Além do anel global de endereçamento, o sistema também

possui um anel por cada nível onde se relacionam os nós pertencentes ao mesmo nível. Os nós ligam-se entre níveis também, por meio de uma rede Butterfly em que cada nó de um nível está ligado a dois nós do nível L +1 e a um nó do nível L -1 sendo L o nível do nó. (Malkhi et al. 2002)

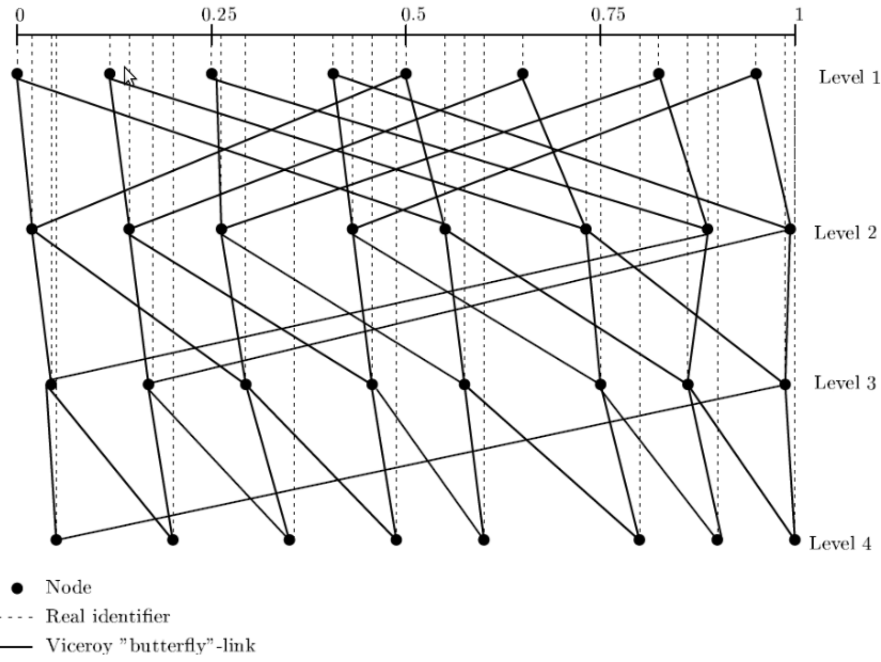


Figura 2.4: Rede Viceroy com 4 níveis sendo a primeira linha o anel onde estão endereçados todos os nós

Koorde [Estruturado, Completamente descentralizado, Uso geral, $O(\log n/\log\log n)$]. O Koorde é um overlay baseado no Chord, que consegue manter a performance de procura (O(log N)), mantendo um número constante de vizinhos por nó. A performance de procura é variável consoante o número de vizinhos que cada nó mantém, O(log n / log log n) sendo n o número de nós e O(log n) o número de vizinhos mantidos por nó. Para este efeito, os vizinhos do Koorde não são mantidos por uma finger table (Chord) mas sim por um grafo de De Bruijn. Através de operações de shift nos bits dos identificadores, os pedidos de routing são propagados pelo grafo e terminam no nó pedido. (Kaashoek & Karger 2003)

Cyclone [Estruturado, Completamente descentralizado, Uso geral, $O(\log n)$]. O Cyclone é um overlay que permite formar vários clusters de nós e agregá-los numa rede apenas, mantendo uma hierarquia. Os identificadores são atribuídos aleatoriamente mas são divididos

	Centralização	Aplicação	Tempo de procura	Variáveis do Sistema
Chord	Completamente descentralizado.	Infra-estruturas capazes de suportar vários tipos de aplicação.	$O(\log N)$	N = número de nós da rede
Tapestry			$O(\log_B N)$	N = número de nós da rede B = número de bits de endereçamento dos nós
Pastry			$O(\log_B N)$	N = número de nós da rede B = número de bits de endereçamento dos nós
Kademlia			$O(\log_B N + C)$	N = número de nós da rede B = número de bits de endereçamento dos nós C = constante pequena
CAN			$O(d.N1/d)$	N = número de nós da rede D = número de dimensões do espaço cartesiano virtual
Viceroy			$O(\log N)$	N = número de nós da rede
Koorde			$O(\log N/ \log\log N)$	N = número de nós da rede $\log N$ = número de vizinhos por nó
Cyclone			$O(\log N)$	N = número de nós da rede

Tabela 2.1: Comparação de overlays peer-to-peer estruturados

num prefixo e num sufixo, sendo o sufixo o identificador do cluster. Usando apenas o prefixo os vários clusters podem-se organizar segundo anéis tipo Chord autónomos. Usando o identificador completo, os vários clusters podem ser combinados num único cluster. As procuras são efectuadas à semelhança do Chord sendo os pedidos reencaminhados para outro cluster quando é atingido o maior nó pertencente ao cluster actual e que é menor do que o nó de destino. Esta aplicação do Cyclone em redes do tipo Chord denomina-se de Whirl. (Artigas et al. 2005)

2.2.2 Redes peer-to-peer não estruturadas

Gnutella [Não Estruturado, Completamente descentralizado, Partilha de ficheiros, Procura por flood pelos vizinhos]. O Gnutella é um serviço de partilha de ficheiros completamente descentralizado em que os seus participantes assumem tanto a tarefa de clientes (fazendo pesquisas de ficheiros pela rede e recebendo os resultados) como a tarefa de manter o funcionamento da rede reencaminhando os pedidos de procura de ficheiros para outros nós. Quando é feita uma procura, o nó envia o pedido todos os vizinhos que este tem conhecimento. À medida que os nós contactados encontrarem o ficheiro em questão, respondem para o remetente. Uma vez que a rede é completamente descentralizada, os nós podem entrar e sair à vontade sem grande

impacto no funcionamento global da rede. Esta é também altamente resistente a falhas pois não existem nenhuns pontos de falha tais como servidores ou índices centrais. Para um nó se juntar à rede, este tem que conhecer pelo menos um nó que esteja activo no overlay. (Androutsellis-theotokis & Spinellis 2004)

FreeHaven [Não Estruturado, Completamente descentralizado, Publicação anónima de dados]. Além do simples serviço de ficheiros, os overlays peer-to-peer vieram suprir uma necessidade pendente de um serviço de armazenamento de documentos/ficheiros com total anonimato, em que os participantes da rede não podem responsabilizados pelos conteúdos armazenados e que é resistente a tentativas de eliminar os mesmos documentos. O FreeHaven foi a resposta a esta necessidade. É um overlay totalmente descentralizado e a própria falta de estrutura permite total anonimato dos autores, publicadores, leitores, servidores e documentos (na medida em que não existe nenhuma ligação entre o documento e os vários intervenientes). A sua arquitectura é baseada num conjunto de nós sem controlo central funcionando com um sistema de confiança. Para se publicar um documento é preciso primeiro encontrar um servidor disposto a armazená-lo. Este ficheiro é repartido em várias partes e é alojado e replicado pelos vários nós da rede e vai sendo movido de nó em nó consoante o seu grau de confiança (cada servidor vai acumulando os endereços sobre todos os nós na rede à medida que vão sendo transferidos blocos de dados). O grau de confiança é estabelecido pelo historial de armazenamento. Quando um servidor aceita armazenar um documento, compromete-se a fazê-lo por um período definido de tempo específico. Se o nó libertar o documento ou sair da rede antes desse tempo o seu grau de confiança diminui. Cada nó mantém uma base de dados com os nós que mantêm ficheiros que este lhe enviou, e também um registo com o grau de confiança dos nós com que já trabalhou. Tal como o Gnutella, um nó/servidor Free Haven tem que ser inicializado com um endereço de um nó conhecido na rede. (Androutsellis-theotokis & Spinellis 2004)(Dingledine et al. 2001)

Freenet [Levemente Estruturado, Completamente descentralizado, Publicação anónima de dados, Pesquisa por chave e texto descritivo nó a nó]. Semelhante ao

Free Haven, a rede freenet tem como objectivo a publicação de textos e ficheiros sob anonimato no que se denomina numa rede censorship-free. A rede freenet pode-se categorizar por ser uma rede completamente descentralizada mas apenas semi-estruturada, na medida em que cada nó mantém uma tabela de encaminhamento com alguns nós com correspondência entre chaves (representando documentos, ficheiros, blocos de dados, etc.) e nós na rede representando sugestões e os nós mais visitados. (Baumgart et al. 2007) As pesquisas são efectuadas através da escolha de parte dos nós conhecidos e são reencaminhados ao longo da rede toda, até se atingir um resultado. Uma vez que a tabela de encaminhamento é construída dinamicamente e pode não abranger os resultados requeridos, a procura pode não devolver resultados levando a que o overlay não possa ser considerado estruturado. (Androutsellis-theotokis & Spinellis 2004)(Dingledine et al. 2001)

Kazaa [Não Estruturado, Parcialmente centralizado, Partilha de ficheiros, Pesquisa pelos ficheiros indexados no super-peer]. O Kazaa é uma rede de partilha de ficheiros muito popular muito semelhante à rede Gnutella mas com a existência de uma hierarquia de nós em que passam a existir super nós (Supernodes) responsáveis por algum controlo na operação da rede. Cada nó normal na rede está associado a um super nó que mantém conhecimento do catálogo de ficheiros que o nó normal está a partilhar. O sistema torna-se bastante mais rápido nas suas procuras que uma rede Gnutella, na medida em que as pesquisas são realizadas apenas nos super nós mas é obviamente menos resistente a falhas, uma vez que a falha de um super nó implica que os seus nós associados se tenham que ligar à rede de novo (e a um novo super nó). (Information & Liang 2004)

eDonkey [Não Estruturado, Parcialmente centralizado, Partilha de ficheiros]. A rede eDonkey é constituída à semelhança da rede Kazaa por um conjunto de nós servidores (Supernodes) que contém também os metadados relativos aos ficheiros partilhados pelos nós clientes associados a ele. A rede eDonkey, ao particionar os ficheiros em várias porções transmitíveis, permite aos clientes fazer a transferência do mesmo ficheiro de vários nós ao mesmo tempo fazendo um download multi-parte

à semelhança de um download por um browser de um servidor http. Cada ficheiro partilhado é identificado por uma função de hashing que permite que o mesmo ficheiro com nomes diferentes seja identificado pelo servidor/super-node como sendo o mesmo ficheiro. Nós com o ficheiro na lista de transferências mas que ainda não possuem o ficheiro inteiro também podem enviar as suas partes a outros clientes. À semelhança da rede Kazaa e Gnutella, as transferências são feitas directamente entre os nós clientes. Quando um nó cliente procura um ficheiro o seu nó servidor devolve-lhe a lista de ficheiros que correspondam ao nome pretendido e que nele estejam indexados. (Lua et al. 2005)(Heckmann & Bock 2002)

BitTorrent [Não Estruturado, Parcialmente centralizado, Partilha de ficheiros]. A arquitectura da rede bit torrent é também centralizada, mas contrariamente às redes Kazaa e eDonkey, os servidores são dedicados e não são nós participantes na rede. Estes servidores (chamados trackers) são participantes que se dedicam exclusivamente a manter uma lista de todos os nós que possuem o conjunto de ficheiros em questão. Os torrents são ficheiros que descrevem um conjunto de ficheiros (possivelmente organizado em directorias com vários ficheiros), seus tamanhos, nomes e checksums e o endereço do tracker responsável pelo conjunto de ficheiros. As transferências são depois efectuadas directamente entre os nós clientes exactamente à semelhança da rede eDonkey. Quando um tracker não está em funcionamento os ficheiros pelos quais este é responsável deixam de estar disponíveis. A rede Bit Torrent manté também um sistema de incentivo que favorece nós que têm elevada taxa de partilha e upload dando-lhes mais prioridade nas listas de espera. Os ficheiros torrent não estão disponíveis na própria rede e têm que ser servidos externamente à rede, de modo a que quando um nó deseja procurar um ficheiro ou conjunto de ficheiros, este tem já que ter o ficheiro torrent correspondente. Sites como TorrentReactor.net ou o Mininova.org, alojam ficheiros torrent e possuem motores de busca internos para sua pesquisa. (Lua et al. 2005)(Pouwelse & Pawea 2005)

Napster [Não Estruturado, Híbrido descentralizado, Partilha de música]. A rede Napster foi provavelmente o primeiro sistema peer-to-peer de partilha de

<table>
<tr><th></th><th>Centralização</th><th>Aplicação</th><th>Tipo de procura</th><th>Tipo de decentralização</th></tr>
<tr><td>Gnutella</td><td rowspan="3">Completamente descentralizado.</td><td>Partilha de ficheiros</td><td rowspan="2">Procura por flood pelos vizinhos</td><td>Todos os nós são equivalentes, não existindo hierarquia.</td></tr>
<tr><td>FreeHaven</td><td rowspan="2">Publicação anónima de dados</td><td>Todos os nós são equivalentes, sendo classificados por um sistema de confiança.</td></tr>
<tr><td>Freenet</td><td>Publicação anónima de dados, Pesquisa por chave e texto descritivo nó a nó.</td><td>Aproximação ao modelo estruturado. Os nós estão endereçados mas não existe um algoritmo de procura dependente da estrutura de endereçamento de nós.</td></tr>
<tr><td>Kazaa</td><td rowspan="3">Parcialmente centralizado.</td><td rowspan="2">Publicação de ficheiros</td><td>Pesquisa pelos ficheiros indexados no super-peer.</td><td>Os nós são associados aos seus super-peers e os super-peers comunicam uns com os outros.</td></tr>
<tr><td>eDonkey</td><td>Pesquisa pelos ficheiros indexados nos nós servidores.</td><td>O nós são associados a um dos vários servidores disponíveis.</td></tr>
<tr><td>BitTorrent</td><td></td><td>A procura é efectuada pelo tracker com que o software foi inicializado (através do respectivo ficheiro .torrent).</td><td>Não existe um servidor central nem super-peers mas um tracker que gere os nós que contêm os ficheiros pelo qual é responsável.</td></tr>
<tr><td>Napster</td><td>Híbrido descentralizado</td><td>Partilha de música</td><td>Pesquisa do ficheiro no servidor central.</td><td>Existe apenas um servidor central com todos os ficheiros disponíveis indexados e o nó a qual o ficheiro pertence</td></tr>
</table>

Tabela 2.2: Comparação de overlays peer-to-peer não estruturados

música a tornar-se conhecido e a ser usado em larga escala (tendo sido também o primeiro a ser alvo de um processo judicial bastante mediático que levou ao fim do seu funcionamento original). A sua arquitectura é composta por um único servidor central e os nós clientes que armazenam os ficheiros em si. O servidor central serve apenas de directório central com todos os ficheiros disponíveis na rede. Quando um cliente se liga à rede, transmite a sua lista ao servidor, e quando efectua uma procura o mesmo servidor devolve-lhe a list de nós cujo nome do ficheiro coincide com o padrão de procura. Tal como todas as outras redes analisadas, as transferências são efectuadas entre os pares. Uma vez que o servidor central não coordena todas as operações na rede, a rede é classificada como Híbrida Descentralizada (sendo um sistema híbrido entre uma rede puramente centralizada e uma rede semelhante à Kazaa). (Androutsellis-theotokis & Spinellis 2004)

2.2.3 Conclusão

Como se demonstrou, os protocolos estruturados enunciados não foram desenhados com um fim especifico mas na necessidade de construir um sistema de partilha e transmissão de dados, completamente descentralizado e capaz de suportar entradas e sai-

das de participantes sem que isso interfira numa procura. O tempo de procura é semelhante, sendo na sua maioria definido por $O(logn)$ com a notável excepção do Koorde, que sendo baseado no Chord consegue possivelmente acelerar por meio do seu grafo de De Bruijn e o CAN que usa um mapeamento quase geográfico para direccionar a pesquisa de um nó.
Os protocolos não estruturados já são normalmente baseados ou construídos com um fim especifico em vista (normalmente a partilha de ficheiros). Nestes sistemas já é introduzida alguma centralização de modo acelerar as pesquisas que de outra forma teriam que ser feitas por flood. Quanto mais forte for a centralização, mais dependente está o sistema dos seus *super-peer* ou servidor central e menor será a tolerância a falhas.

2.3 Sistemas Grid e Middleware para processamento paralelo distribuído

O conceito de sistema Grid engloba um conjunto muito vasto de projectos diferentes com métodos e objectivos radicalmente diferentes. Desde a aplicação final do projecto, qual o regime de partilha de recursos, tais como:

- **Desktop grid:** A aplicação corre como uma normal aplicação de desktop em que o utilizador voluntaria-se a oferecer parte ou totalidade dos recursos do seu computador pessoal para o projecto. Qualquer tipo de computador pessoa pode ser utilizado e pode apenas contribuir para grid quando este não está a ser utilizado.
- **Aplicação dedicada:** A aplicação transforma o computador numa unidade dedicada a 100% para computação na grid.
- **Marketplace:** O vários participantes na grid registam os seus recursos e aceitam cedê-los a outros participantes da rede para que estes possam executar as suas tarefas.

e qual o tipo de grid, segundo a seguinte classificação:

- **Grid Institucional:** Grid dedicada ao serviço interno de uma instituição ou empresa.

- **Grid Cooperativa/Voluntária:** Grid aberta a quem estiver disposto a oferecer os seus recursos para um objectivo comum num modelo aproximado a uma rede peer-to-peer.
- **Toolkit:** Software desenhado para construir aplicações de computação distribuída/paralela e sistemas grid.

Os projectos seguintes serão apresentados com a seguinte classificação: **Nome do Projecto [Tipo de Grid, Regime de Partilha, Aplicação].**

2.3.1 Projectos Correntes

SETI@Home [Grid Cooperativa/Voluntária, Desktop grid, Processamento de sinal áudio]. O projecto SETI @ Home foi concebido para suprir a falta de poder computacional para processar os dados obtidos pelo projecto SETI (Search for Extra Terrestrial Intelligence). O projecto SETI faz uso do maior radio telescópio do mundo, localizado em Arecibo, Porto Rico para pesquisar emissões no espectro das ondas de rádio em busca de sinais artificiais, possivelmente oriundos de civlizações extra terrestres. O radio telescópio armazena todos as suas observações em fitas de dados (fitas de 35 GB, gravadas num rácio de 5 MB por segundo, contendo cada fita 35GB). Estas fitas depois de cheias são enviadas por correio comum para a universidade de Berkeley nos Estados Unidos (pois não existe ligação de banda larga disponível em Arecibo) onde são introduzidas no servidor do sistema. Os dados são extraídos das fitas e particionados segundo as várias bandas de frequências e depois em blocos de 107 segundos. Este blocos são colocados no servidor de dados e resultados e são posteriormento transmitidos para máquinas com o software cliente a correr, processados e enviados de volta. Para uma máquina ser participante da rede, tem apenas que possuir uma ligação à internet e ter o software cliente instalado. Este corre como uma aplicação cycle-sharing na medida em que pode ser configurado para correr apenas quando a máquina está em idle ou para correr constantemente com a prioridade de processo no mínimo, mantendo a disponibilidade da máquina para outros trabalhos. A aplicação mantém também estado em disco para que os resultados da computação

não se percam caso o servidor esteja em baixo. (Anderson et al. 2002)

Folding@Home [Grid Cooperativa/Voluntária, Desktop grid, Processamento de simulações em bioquímica]. À semelhança do projecto SETI@Home, o projecto Folding@Home surgiu a partir de um problema com um custo computacional demasiado elevado para as máquinas disponíveis na altura. Neste caso trata-se de um problema relativo a medicina e biologia denominado de Protein Folding. A pesquisa deste problema permite ganhar conhecimento acerca do desenvolvimento de terapias/vacinas para várias doenças. Para dar resposta a isto foi criado um modelo de simulação que faz uso de uma rede semelhante à SETI@Home para enviar os dados para várias máquinas e receber os seus resultados. Sendo um projecto com uma importância imediata maior do que o SETI @ Home, rapidamente foram desenvolvidos outros tipos de clientes para que a rede possa ser alargada além dos desktops em cycle-sharing. Exemplos são o cliente para Playstation 3 e principalmente um cliente para tirar partido de GPUs em placas gráficas. Este último permite correr a aplicação cliente em cluster de placas gráficas (normalmente usados para fazer tratamento de vídeo e CGI em filmes) e usar os seus processadores de alto desempenho mantendo-se parte da grid como mais um nó participante. (Larson et al. 2009)

BOINC [Toolkit, Desktop grid, Várias aplicações]. Tanto o projecto SETI@Home como o projecto Folding@Home são construídos sobre uma framework de construção de sistemas Grid denominada de BOINC. A infra-estrutura BOINC foi desenhada para que um cientista de investigação consiga instalar um sistema Grid com pouco trabalho inicial e que permita ajudá-lo nas suas necessidades de computação usando uma rede de computadores voluntários. Esta infra-estrutura começou como parte do projecto SETI@Home e acabou por ser separada num infra-estrutura de middleware dado o potencial da tecnologia em outros projectos. A framework permite correr uma aplicação escrita em C ou C++ com muito poucas alterações ao código, e permite manter a rede em funcionamento com bastante pouca manutenção semanal. (Anderson 2004)

Grid4All [Grid Institucional, Marketplace, Partilha de recursos entre os vários intervenientes]. O projecto Grid4All faz parte de uma iniciativa da União Europeia para criar uma rede completamente democrática e aberta que permita a utilização de recursos computacionais disponíveis por qualquer pessoa ou empresa. O projecto pretende facilitar a gestão e os custos da criação e manutenção de recursos informáticos para suporte de infra-estruturas TI criando um sistema que seja completamente autónomo e gerido por si próprio, baseado em tecnologia peer-to-peer. O formato abstracto da rede assemelha-se a um mercado onde existem ofertas e pedidos de recursos de computação ou armazenamento sendo estes recursos alocados a um indivíduo ou uma organização por um determinado preço por unidade de tempo. (Vouros et al. 2008)

OurGrid [Toolkit, Marketplace/Aplicação dedicada, Partilha de recursos entre os vários intervenientes]. Num objectivo semelhante ao projecto Grid4All surge o projecto OurGrid que consiste numa rede aberta e grátis onde os participantes servem simultaneamente de fornecedores e requerentes de recursos. A OurGrid oferece um toolkit que permite construir um pequeno cluster que pode ser rapidamente usado para correr aplicações distribuídas e que pode também ser ligado à comunidade OurGrid onde irá partilhar os seus recursos com o resto da rede. As aplicações a correr na rede são limitadas apenas a aplicações que podem ser divididas em várias tarefas (tasks) divisíveis e não dependentes umas das outras. Estas aplicações são denominadas de aplicações BoT (Bag of Tasks). Estas tarefas são executadas em máquinas virtuais criadas na altura de correr a tarefa e destruídas no final, mantendo total segurança e anonimato na tarefa a correr. (Andrade et al. 2005)

Xgrid [Toolkit, Aplicação dedicada, Computação Distribuída]. O Xgrid é um software desenvolvido pela Apple para o seu sistema operativo Mac OS para efectuar computação distribuída numa rede local de computadores Mac OS. Usa o serviço Rendezvous para fazer a descoberta de nós para a rede, sem necessidade de configuração e o overlay peer-to-peer BEEP(Anderson 2004) para efectuar a comunicação necessária. Fazendo uso de uma configuração user friendly baseada em interface gráfica, o

objectivo deste software é apelar a que um investigador ou cientista consiga montar uma grid local sem necessidade de efectuar configurações nem de perder tempo a descobrir novas tecnologias. (Kramer & MacInnis 2004)

XtremWeb [Toolkit, Aplicação dedicada/Desktop Grid, Computação Distribuída]. Um projecto muito semelhante ao OurGrid é o XtremWeb baseado em tecnologia Java. Oferece as mesmas funcionalidades, permitindo várias configurações possíveis, desde uma grid privada sem ligação com o exterior até a uma comunidade semelhante à OurGrid. É possível configurar como participante da grid desde um pc desktop voluntário a correr um cliente em screensaver, até uma server farm por detrás de um router. À semelhança da infra-estrutura BOINC também é possível configurar um servidor de agregação de resultados para onde as máquinas em regime voluntário fazem upload de resultados directamente. (Fedak 2007)

GiGi [Grid Cooperativa/Voluntária, Desktop Grid, Computação Distribuída]. O GiGi é um projecto português em Grid Computing que consiste num motor de execução de gridlets altamente configurável baseado num overlay peer-to-peer. Cada gridlet consiste numa tarefa ou unidade de execução, os dados a processar e informação sobre o custo de execução da unidade, em função de memória ocupada e ciclos de cpu necessários. Os gridlets são enviados de nó em nó até a sua execução terminar sendo convertidos em gridlet-results e reenviados para os nós que os submeteram para a rede. Esta função de custo permite um balanceamento da carga da grid mais eficiente que não é possível noutros sistemas. (Veiga et al. 2007)

2.3.2 Conclusão

Os exemplos dados demonstram bem a multitude de aplicações finais que podem ser dadas a um sistema Grid, sendo que o objectivo final será sempre criar um ambiente onde se possam executar tarefas com grandes necessidades computacionais. Desde os sistemas cooperativos com um fim bastante bem definido como SETI@Home aos

	Tipo de Grid	Regime de Partilha	Aplicação
SETI@Home	Grid Cooperativa/Voluntária	Desktop grid	Processamento de sinal áudio captado pelo radiotelescópio do programa SETI.
Folding@Home			Processamento de simulações em proteínas para investigação em doenças incuráveis.
BOINC	Toolkit		Toolkit para construção de desktop grids à semelhança dos projectos @Home.
Grid4All	Grid Institucional	Marketplace	Mercado para oferta e procura de recursos computacionais fornecidos pelos intervenientes à semelhança de um mercado de acções.
OurGrid	Toolkit		Toolkit para construção de grids para computação distribuída ou para participação em sistema marketplace semelhante ao Grid4All.
Xgrid		Aplicação dedicada	Toolkit da Apple para a fácil construção de pequenas grids para computação distribuída orientado a projectos pessoais de cientistas.
XtremWeb		Aplicação dedicada/Desktop Grid	Tookit semelhante ao projecto BOINC que permite construir uma grid para computação distribuída.
GiGi	Grid Cooperativa/Voluntária	Desktop Grid	Infraestrutura para um sistema grid baseado em gridlets, com função de custo de execução associada

Tabela 2.3: Comparação de sistemas Grid

sistemas marketplace, a ideia final passa sempre por fornecer ou criar um meio onde se possa repartir tarefas complexas e pesadas por vários participantes. Foram também descritos algumas frameworks/toolkits, como o OurGrid ou o BOINC, que permitem a fácil criação destes ambientes sem necessidade de construir um sistema complexo de raiz.

2.4 Simulação de redes/overlays

Como já foi falado na introdução, um simulador de overlays peer-to-peer é desenhado de modo a criar um conjunto de nós virtuais e simular a sua interacção como se de uma rede física se tratasse. À semelhança de (Kramer & MacInnis 2004) e (peersim) os seguintes simuladores serão classificados segundo a sua arquitectura:

- **Eventos Discretos:** O simulador usa um sistema de eventos em que os vários nós enviam mensagens uns aos outros que são consequentemente sincronizadas e potencialmente atrasadas pelo simulador de modo a manter coerência da simulação.

- **Query-Cycle:** O total da simulação é composta por um conjunto de ciclos de queries, que consistem num conjunto de operações ordenadas de comunicação entre os pares que correm todas sequencialmente até ao final. Um ciclo só termina (e posteriormente só é iniciado o seguinte ciclo) quanto todos os nós tem a resposta a sua query.

extensibilidade a protocolos feitos por medida e escalabilidade (número máximo de nós). Os projectos seguintes serão apresentados com a seguinte classificação: **Nome do Projecto [Arquitectura, Extensibilidade, Número máximo de nós].**

2.4.1 Projectos Correntes

GPS [Eventos Discretos, Extensível a novos protocolos, N/A]. O GPS foi um simulador de overlays peer-to-peer baseado em Java e capaz de simular vários protocolos cujo desenvolvimento se encontra inactivo. A framework de simulação é baseada em eventos discretos. Os eventos podem ser acções por parte do utilizador, mensagens trocadas entre os nós ou eventos internos ao simulador. O GPS permite também usar vários tipos de protocolo peer-to-peer na medida em que o simulador é configurável por meio de um componente extensível em que se podem re-implementar funções como pesquisa de nós/chaves, entrada e saída de nós. (Yang & Abu-ghazaleh 2005)

P2Psim [Eventos Discretos, Extensível a novos protocolos, 3000 nós]. O P2PSim é um simulador feito em C++ capaz de simular à partida os overlays Chord, Koorde, Kademlia, Accordion, Kelips e Tapestry. Este simulador suporta até 3000 nós usando o motor de simulação baseado em eventos discretos, com garantia de correr a simulação em tempo útil e fornece algumas estatísticas de base sem necessidade de escrever novo código. É também possível implementar novos protocolos, à semelhança do GPS implementado estendendo o software existente e implementando as funções de pesquisa e junção de nó à rede. Infelizmente o projecto encontra-se com pouca actividade actualmente e tem muito pouca documentação. (Naicken et al. 2007)(peersim)

OverSim [Eventos Discretos (inclui simulação de camada TCP/IP), Extensível a novos protocolos, 100.000 nós]. O OverSim é uma framework de simulação de overlays baseada também em eventos discretos, que oferece grande escalabilidade, uma fácil configuração de protocolos (para permitir a implementação de overlays estruturados e não estruturados), um modulo de estatísticas bastante completo, e um visualizador na forma de uma interface gráfica que permite facilmente efectuar a depuração de protocolos do overlay como visualizar a topologia da rede (tanto a nível do overlay como a nível da simulação da camada inferior de simulação). O Oversim permite

	Arquitectura	Extensibilidade	Número máximo de nós
GPS	Eventos Discretos	Extensível a novos protocolos	N/A
P2Psim			3000 nós
OverSim			100.000 nós
Peersim	Eventos Discretos e Query-Cycle à escolha		106 (1 milhão) de nós usando o motor Query-Cycle

Tabela 2.4: Comparação de simuladores peer-to-peer

também modelar a rede ao nível da camada inferior (TCP/IP, UDP, etc) de modo a permitir a simulação de uma rede realista com limites de largura de banda, congestionamento e perda de pacotes ou a utilização de uma rede sem limites de eficiência para o estudo puro do comportamento da rede com elevado número de nós. À semelhança dos outros simuladores referidos, a implementação de novos protocolos faz-se definindo não só as funções de entrada, saída, e procura mas também o tratamento de mensagens, e os mecanismos para se poder efectuar a visualização gráfica da rede. O oversim corre com tempos aceitáveis numa rede simulada até 100.000 nós. (Baumgart et al. 2007)

Peersim [Eventos Discretos e Query-Cycle à escolha, Extensível a novos protocolos, 1 milhão de nós]. O Peersim que já foi descrito na introdução, será o simulador a usar no projecto. Este projecto é baseado em Java e permite a utilização tanto de um motor baseado em eventos como de uma simulação baseada no paradigma Query-Cycle. À semelhança dos anteriores projectos, este simulador também é extensível a protocolos feitos por medida mas apresenta uma maior escalabilidade registada do que os mesmos. (Naicken et al. 2007)

2.4.2 Conclusão

Dada a espeficidade do conceito de simualção de *overlays peer-to-peer*, são enunciados poucos projectos de relevância. O peersim releva-se como o mais extensível e mais completo, sendo um projecto open source feito em java com uma API bem definida para a introdução de novos protocolos. O OverSim apresenta-se como uma proposta interessante do ponto de vista de análise de comportamento da rede apresentando

um módulo de estatisticas e interface gráfica que permite uma análise mais fácil do comportamento e estado da rede.

2.5 Resumo

Neste capítulo foi feita uma enumeração de vários projectos nas áreas *Peer-to-Peer*, Computação Grid e simulação de *overlays*. Foram classificados vários projectos *Peer-to-Peer* segundo a sua centralização, tempo e tipo de procura, qual a sua aplicação final. Foram também descritos vários projectos na área da Computação Grid, desde projectos cooperativos com fins científicos, a sistemas institucionais de venda de recursos e também toolkits e frameworks de apoio à construção de sistemas Grid. Por fim, foram enunciados alguns projectos de simuladores *overlays* sendo realçada a extensibilidade e flexibilidade apresentada pelo Peersim.

3 Arquitectura da Solução

Para atingir o objectivo proposto, a versão actual do Peersim tem que ser transformada para abandonar a principal restrição que apresenta hoje: a sua execução puramente sequencial e *single threaded*. Embora o Terracotta ofereça uma solução *out of the box* para ligar várias máquinas virtuais Java (VM) sem ser necessário alterar código, isto não faria mais no Peersim do que podermos ter várias VM em comunicação a correrem simulações independentes. Portanto, cada simulação individual não veria a sua execução acelerada nem tão pouco disporia de mais memória. Recordando, o objectivo principal deste trabalho é fazer com que uma simulação consiga fazer uso de recursos de memória e CPU disponibilizados por um ou vários participantes para se poder atingir tempos de simulação menores, o que se traduz inevitavelmente numa maior eficiência.

A abordagem ao problema foi feita em duas fases principais:

- **Paralelização** - Modificar o motor de simulação do peersim para que este seja capaz de executar em várias threads.
- **Distribuição num cluster** - Estender o mesmo motor de simulação para que este consiga fazer uso da plataforma de memória distribuída disponibilizada pelo Terracotta.

3.1 Paralelização

Como foi referido nas secções anteriores, na sua versão original, o Peersim oferece dois motores independentes. Um motor ***cycle-driven*** e um motor ***event-driven***. Na sua versão *cycle-driven*, o Peersim efectua um número definido de ciclos em que percorre

todos os nós do overlay e por cada um, activa todos os protocolos que tenham sido definidos para executar a cada ciclo (Figura 3.1).

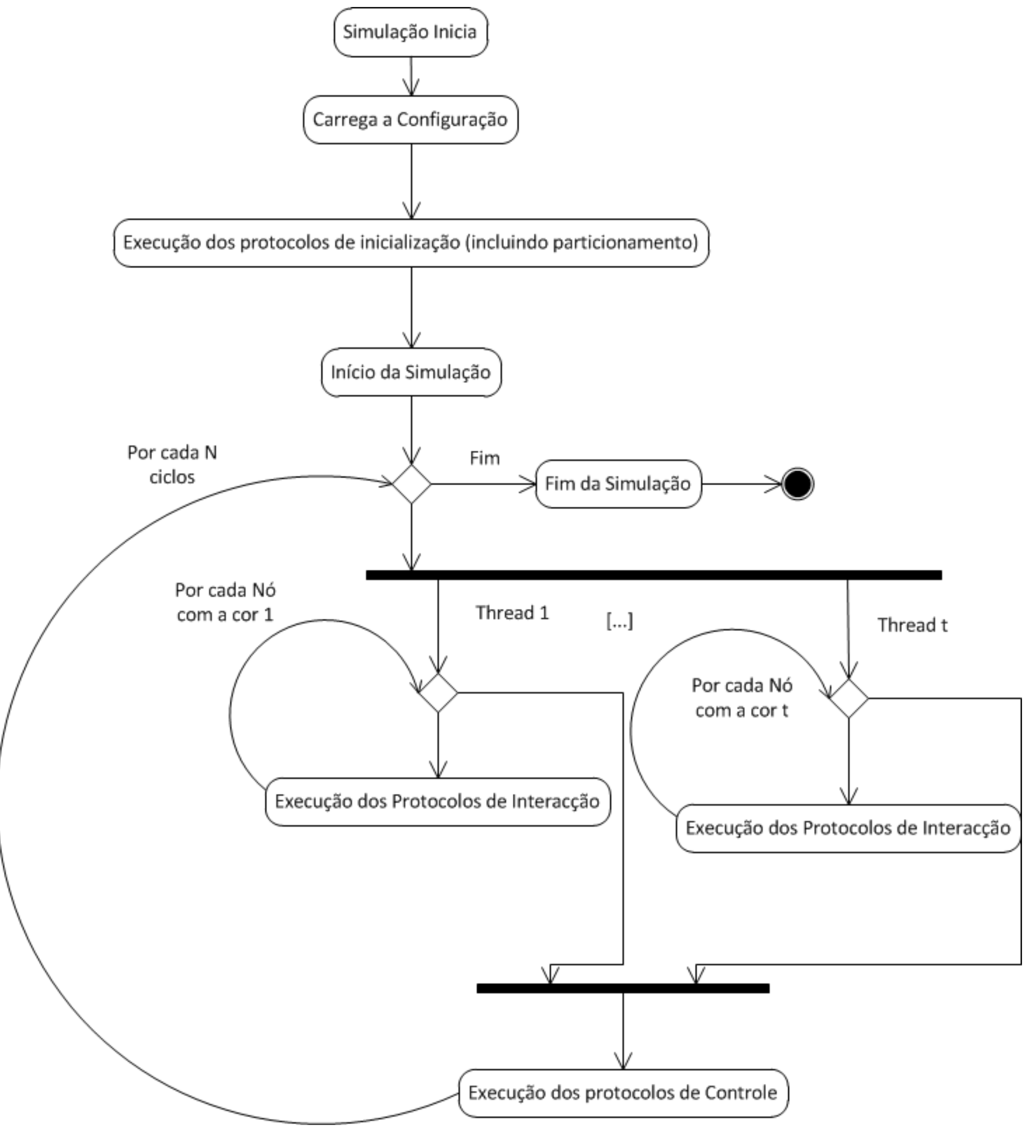


Figura 3.1: Execução da simulação *Cycle-Driven*

Na sua versão *event-driven*, o simulador inicializa-se com uma *queue* ordenada em que são inseridos eventos relativos à activação de todos os nós (entre eventos de controle da simulação). Esta *queue* vai sendo consumida num ciclo infinito, em que os eventos depois de activados são inseridos no final desta. Quando a simulação atinge um tempo limite (definido por um total de eventos a processar) os eventos deixam de ser reciclados para o fim da queue e quando esta fica vazia, a simulação termina

(Figura 3.2). Em ambos os casos a simulação percorre todos os nós ciclicamente, activando a cada um os protocolos configurados.

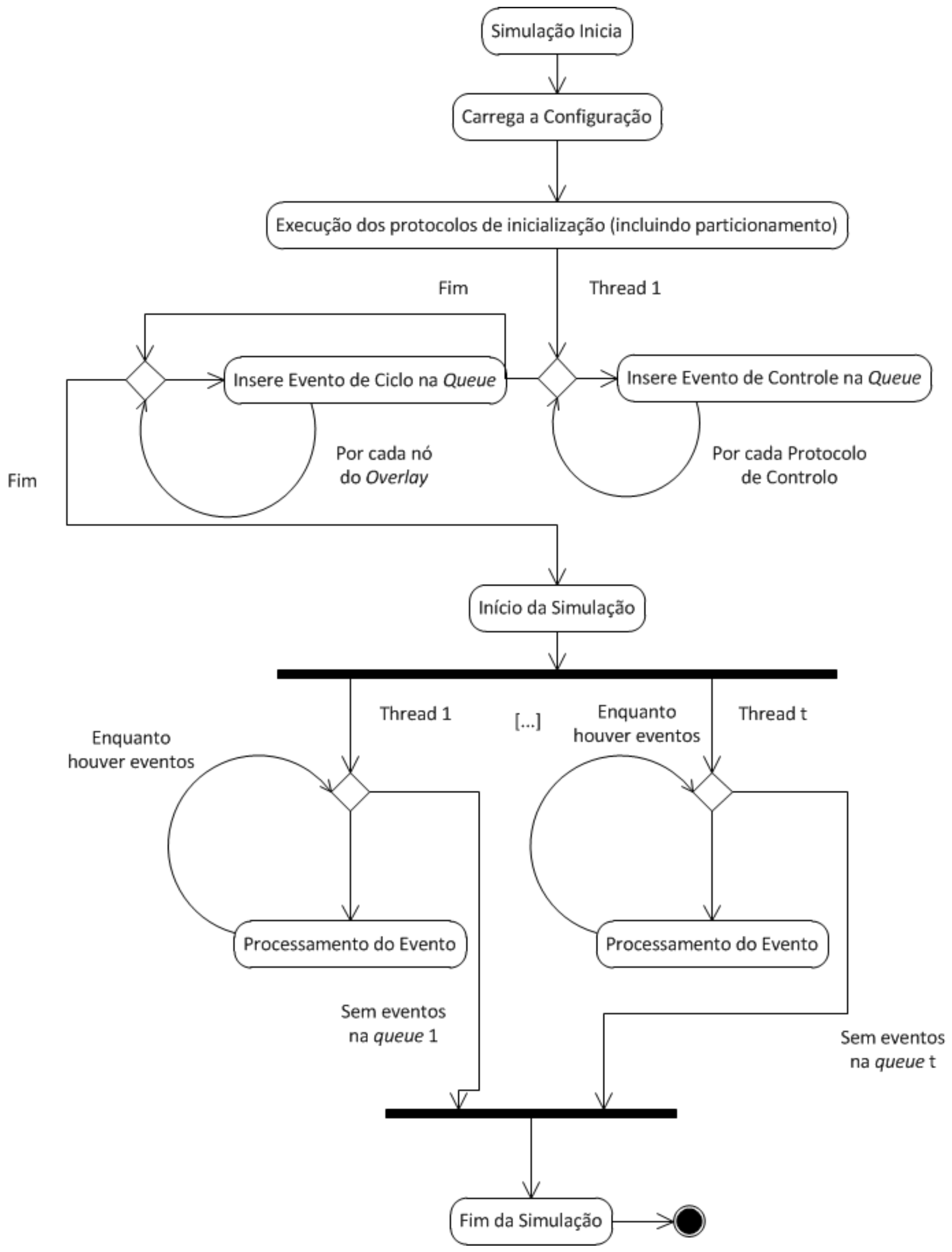


Figura 3.2: Execução da simulação *Event-Driven*

De modo a que a simulação possa ser executada em vários fios de execução simultaneamente implica que várias *threads* partilhem o conjunto de nós que representam o *overlay*. Portanto, numa abordagem simples, em ambos os motores a solução para o

problema da paralelização partiria por ter várias *threads* a alimentarem-se da lista ou *queue* e a sincronizarem toda a actividade efectuada em cada nó.
Esta solução, embora simples, peca pela sua ineficiência devido a ser necessário sincronizar todo o código de interacção entre os nós. Esta ineficiência advém do facto de código sincronizado demorar invariavalmente mais tempo a executar do que código desprotegido (*thread-unsafe*). Sendo que todas as *threads* alimentam-se de um conjunto de nós global, não é possível determinar qual das threads vai activar o nó X, sendo portanto necessário sincronizar todas as interacções. Surgindo desta necessidade de sincronizar a maioria da execução da simulação surge a hipótese de atribuir apenas parte do overlay a cada *thread* permitindo que se sincronize apenas interacções possivelmente concorrentes.

3.1.1 Particionamento do *overlay*

De modo a definir e saber quais os nós a serem activados e simulados por cada thread foi adicionado um mecanismo de particionamento do *overlay* ao peersim. O particionamento da rede, semelhante à coloração de um grafo, é determinada por um algoritmo que atribui a cada nó da rede uma cor que determina qual a thread a activar esse nó. O número de cores numa simulação é igual ao número de threads configuradas para executar a simulação.
Para que a coloração seja o mais eficaz possível, esta deve ser feita tendo em conta as relações entre os nós, ou seja, deve ser feita de modo a que durante a execução dentro de uma das threads, as interacções entre os nós sejam feitas na sua maioria por nós da mesma cor. Sendo que o simulador só efectuará sincronização ao simular a interacção entre nós de cor diferente, no caso ideal, a coloração deve ser feita de modo a uma percentagem máxima de ligações entre vizinhos da mesma cor. Embora o simulador seja sempre configurável com qualquer protocolo, este será sempre baseado nas relacções de vizinhança (ou então significa que estará a passar por cima do conceito do overlay *peer-to-peer*), garantindo que o esquema de coloração do overlay traz garantias de melhoria de performance globalmente.

3.2 Distribuição

Ultrapassado o limite de processamento associado à versão single-threaded, surge o limite à memória utilizada pela simulação. Esta barreira é a que verdadeiramente limita o peersim mo seu potencial pois impôe um limite ao tamanho do overlay que se pretende simular. Adicionalmente, a complexidade dos protocolos a usar na simulação podem fazer subir o custo de ocupação de memória associado a cada nó simulado (consequentemente o rácio de memória ocupada/número de nós do overlay).
O Terracotta e a sua capacidade de partilhar facilmente objectos entre várias VMs sem necessidade de modificar o código original da aplicação surge um pouco como a solução perfeita para atingir o objectivo, mas no entanto existem alguns problemas que necessitam de solução. Problemas estes que significam a diferença entre uma aplicação capaz de balancear a carga entre todos os participantes em vez de uma aplicação que continua a fazer o trabalho todo numa máquina apenas, e cujos restantes participantes estão apenas ociosos sem participar no trabalho.
Embora o Terracotta permita fácil ligação entre VMs, o seu regime de partilha de memória é puramente ao nível da *heap* java e não da memória virtual de ambas as máquinas. Ou seja, se um objecto (ou conjunto de objectos) for carregado numa VM, esta carga não é partilhada pela pelos vários participantes, mas apenas pela máquina que o carregou. O contrário obrigaria a um rebalanceamento da carga de cada vez que um participante entrasse ou abandonasse o *cluster*. Sendo que a maioria da carga usada na simulação é representada pela quantidade de nós que é carregada, esta terá que ser repartida pelos vários participantes para que o objectivo de aumentar o tamanho possivel de uma simulação seja atingido.
Combinando o carregamento distribuído com o desenvolvimento feito para atingir a paralelização, atinge-se a distribuição do processamento além da distribuição da carga em memória. Atribuindo uma cor a cada participante, utiliza-se o mecanismo de coloração para que cada máquina faça apenas parte do processamento total, e todas, concorrentemente, o realizem de forma mais rápida.
Embora o Terracotta não necessite que se altere o código da aplicação a utilizar, torna-se óbvio que para atingir os objectivos propostos, é necessário efectuar algumas

alterações funcionais. Estas alterações não incluem no entanto a partilha de objectos e a sua devida sincronização. Esta configuração é efectuada fora do código e de modo a manter a simulação o mais rápida e eficiente possível, apenas é partilhado entre as VMs a lista global de nós do overlay, a configuração de core,s e variáveis necessárias a servirem de trincos para que a simulação partilhada pelos vários participantes se mantenha sincronizada.

3.3 Arquitectura

A figura 3.3 representa o funcionamento da arquitectura geral do PC-Peersim. É descrito de uma forma simplificada como o overlay virtual acaba por ser suportado por um cluster real. No topo da figura temos a interface de entrada do Peersim onde se definem os parâmetros da simulação a correr. Na base temos as interacções entre os nós. Como refererido anteriormente, a grande extensão à arquitectura são os protocolos de particionamento da rede. O Terracotta Middleware surge como um wrapper entre a aplicação e a máquina virtual Java.

Em mais detalhe:

- **Aplicação Exemplo/Simulações Peersim** - A definição e configuração de simulações mantém-se inalterada em relação ao peersim original. Esta configuração define dados como o tamanho do overlay, o número de ciclos que a simulação vai correr e quais os protocolos de inicialização (que definem como é efectuada a inicialização de valores dos nós, a configuração dos seus vizinhos e, actualmente, o algoritmo de particionamento), controlo (que correm a cada ciclo para efectuar observações ou actualizações), e activação (interacções que os nós efectuam quando enviam uma mensagem).
- **Interface Peersim** - A interface peersim sofre algumas alterações face à sua versão original. A sua etapa de configuração passou a efectuar o carregamento total ou parciall do overlay (caso a simulação esteja a correr em mais do que uma

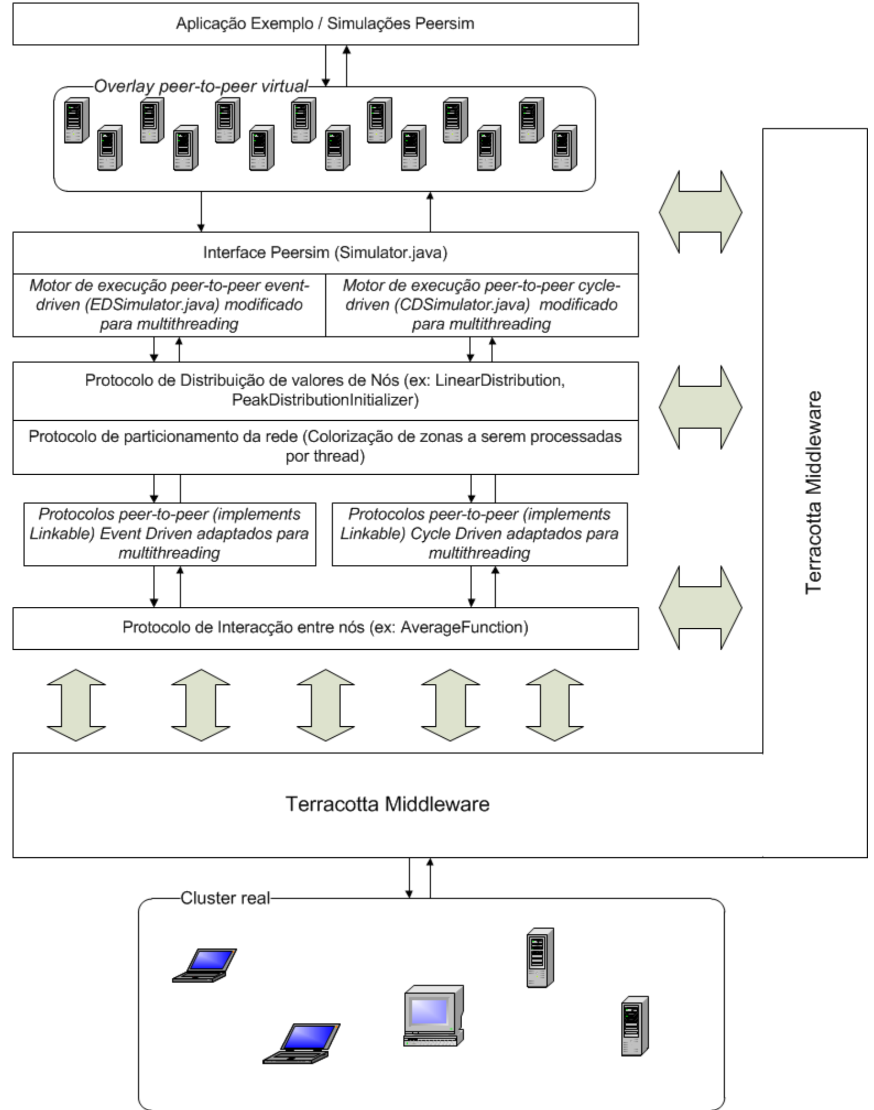


Figura 3.3: Arquitectura geral do sistema

máquina). Como referido no ponto anterior, os vários algoritmos de particionamento da rede foram definidos como protocolos do Peersim de modo a manter a arquitectura o mais fiel ao original possível. O particionamento é portanto, efectuado durante a fase de configuração e inicialização do simulador. Para suportar esta característica, as representações dos nós e da rede passaram a manter informação sobre a coloração dos mesmos.

- **Motor de Execução** - O motor de execução é o componente que efectivamente percorre todos os nós durante os vários ciclos (ou consumo de eventos) e efectua as operações configuradas. Consoante o tipo de motor e modo de funcionamento,

o motor de execução fará coisas diferentes:

- **Modo Original** - O motor *Cycle Driven* trabalha com a totalidade do overlay e efectua todos os ciclos até ao final. O motor *Event Driven* trabalha com uma *queue* global de eventos configurados com todos os nós do overlay até atingir o limite configurado de eventos.
- **Modo Distribuído** - O motor *Cycle Driven* trabalha sobre uma partição do overlay e fica bloqueado à espera que todos os participantes terminem o mesmo ciclo. O motor *Event Driven* não está disponível no modo distribuído.
- **Modo Multithreaded** - O motor *Cycle Driven* a cada ciclo lança uma thread por cada cor e cada thread trabalhar com a porção do overlay associada á sua cor.. O motor *Event Driven* lança também uma thread por cor em que cada uma trabalha com uma *queue* independente configurada com os nós associados à sua cor, sendo que cada thread espera ao final de x eventos de modo a que nenhuma thread se adiante.

• **Protocolo de Distribuição de Nós** - O protocolo de distribuição de nós é aquele que define como são inicializados os valores dos nós. Quando se trata de simulações com vários milhões de nós, torna-se impensável atribuir valores com significado um a um. Define-se então um protocolo de inicialização que faça esta atribuição de valor automaticamente, seja de uma forma aleatória, um inteiro seguindo uma distribuição linear, uma representação textual da data corrente, etc.

• **Protocolo de Particionamento do *overlay*** - Este protocolo contém a implementação necessária para efectuar uma distribuição eficaz de cores pela rede de modo a que se formem zonas ou aglomerados de nós com a mesma cor dentro do overlay.

• **Protocolos peer-to-peer** - Este componente define como é feita a comunicação entre os nós e como é organizada a vizinhança de cada nó do overlay. O Peersim por omissão trabalha com um simples protocolo denominado **WireKOut** em que todos os nós são inicializados com um número fixo de vizinhos aleatórios da rede. Um dos objectivos do peersim é a fácil integração de novos protocolos pelo que existem várias implementações *third party* de outros algoritmos.

- **Protocolo de Interacção entre Nós** - Por fim, este componente define o que fazem dois nós quando comunicam um com o outro, ou seja, o protocolo *peer-to-peer* a ser efectivamente simulado. Isto representa uma simples troca de mensagem na rede, mas para efeitos da simulação, faz sentido que esta operação seja facilmente observável sem ser necessário observar todos os nós um a um. Nas suas simulações exemplo, o peersim usa uma simples média entre dois inteiros (sendo cada um deles, o valor associado a cada nó).

3.4 Resumo

Neste capítulo foi descrita a arquitectura do PC-Peersim dando especial ênfase às modificações (em termos de arquitectura) que foram necessária para transformar a versão original.
O PC-Peersim destaca-se da versão original por apresentar duas novas características importantes: paralelização e distribuição. Em vez de termos um *overlay* a ser simulado numa única thread numa única máquina, o PC-Peersim consegue correr a mesma simulação em paralelo num cluster. Devido a isto foi descrita a necessidade de efectuar sincronização entre as threads que efectuam a simulação de modo a manter coerência na simulação e não existirem falsas leituras ou operações de escrita perdidas. Posto isto, é establecido o *rationale* por trás do processo de particionamento do *overlay*.
Da mesma maneira foi demonstrada a utilidade do Terracotta em atingir o objectivo de distribuir a execução do Peersim sobre um cluster. Foram também explicados quais as limitações do Terracotta e os procedimentos necessários para que o PC-Peersim consiga partilhar efectivamente a carga sobre os vários participantes de forma eficiente.

Realização

Aqui estão apresentados em mais detalhe a implementação dos principais componentes desta versão distribuída do peersim:

- Particionamento
- Processamento multithreaded do overlay
- Sincronização da actividade entre nós
- Execução distribuída
- Sincronização da execução distribuída

4.1 Particionamento

Várias abordagens foram feitas aos algoritmos de particionamento de modo a que estes produzissem um overlay com zonas coloridas o mais coesas possíveis. Vamos pressupôr para efeitos de exemplo que temos um overlay com 37 nós e 62 ligações entre vizinhos sendo estas bidireccionais, e que vamos efectuar uma simulação com 4 threads/cores. No caso ideal, depois do particionamento, o overlay ficará dividido em 4 "zonas" (uma por cada cor) (Fig. 4.1). Neste exemplo teríamos uma taxa de ligações entre nós da mesma cor bastante elevada (73.4 %, 13 ligações inter-partição em 62). Chamemos a este valor taxa de coesão daqui em diante. Numa situação mais normal, em que o particionamento fique mais fragmentado como na Figura 4.2, a coesão do particionamento diminui bastante, dado o aumento de ligações sobre as fronteiras levando a que a taxa de coesão baixe para os 55.4%. Tal como foi referido anteriormente, a sincronização necessária nas interações entre os nós é ditada pela pertença ou não dos nós à mesma cor. Torna-se portanto óbvio que quanto maior a

taxa de coesão maior será a eficiência do simulador. A aliar à eficiência da simulação sobre o overlay particionado é preciso adicionar o custo da própria execução do particionamento, sendo que obviamente estratégias diferentes levarão a diferentes tempos de execução deste. No limite, poderemos ter um algoritmo de particionamento que demora tanto ou mais tempo a completar do que uma simulação pouco demorada.

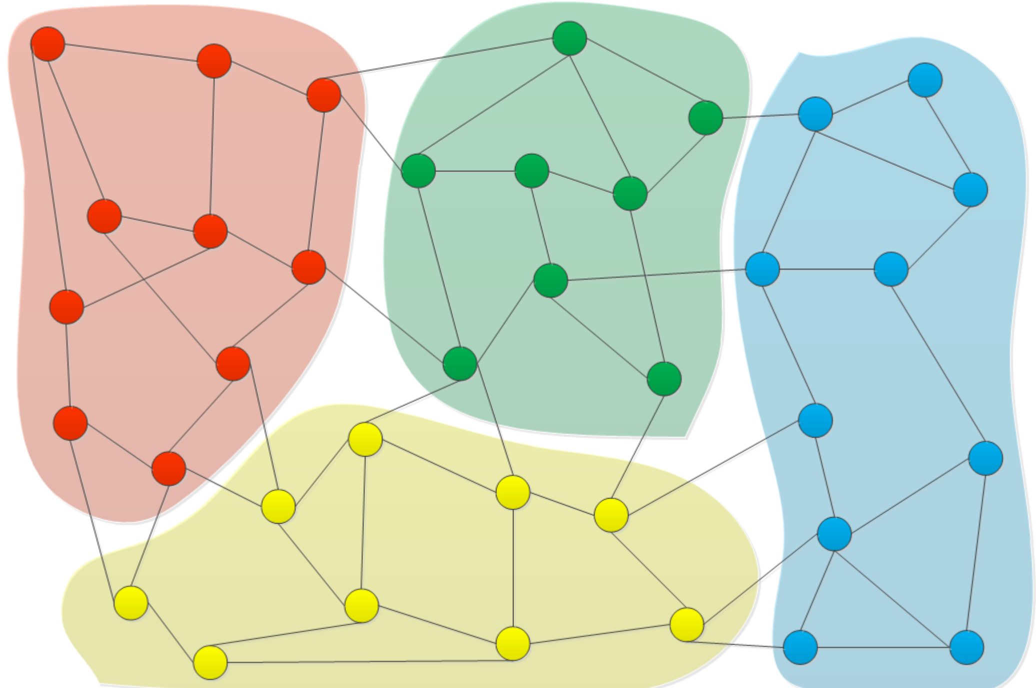


Figura 4.1: Particionamento ideal

Várias abordagens foram desenhadas e exploradas de modo a obter o algoritmo mais eficiente possível, desde versões *single-threaded*, *multi-threaded*, baseadas em pesquisa por largura-primeiro e profundidade-primeiro.
A primeira abordagem multithreaded revelou-se bastante ineficiente devido ao uso de recursão. O algoritmo lançava duas threads iniciadas em 2 nós aleatorios e para todos os vizinhos atribuia-lhes a devida cor e chamava-se a si proprio em cada um deles. Este processo era repetido até todo o overlay estar colorido. Em overlays grandes, este processo ocupava imensa memória, pois deixava muitas chamadas a métodos pendentes em pilha à espera de resolução. Mesmo eliminando a recursão (usando uma fila de nós a colorir), facilmente uma das threads consegue colorir bastante mais nós do que

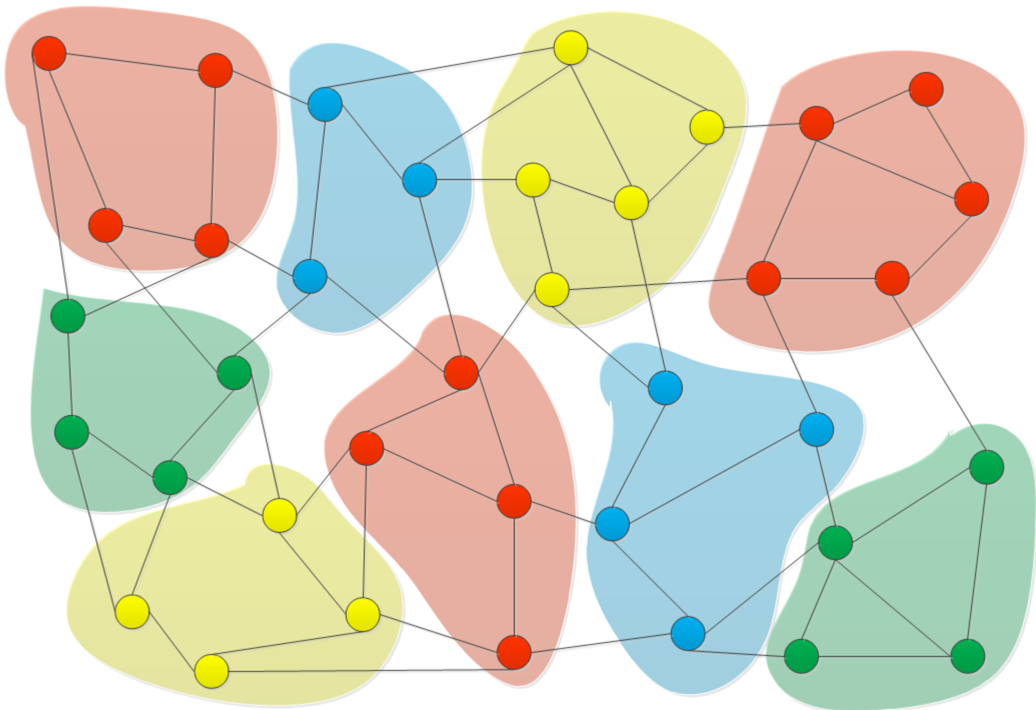


Figura 4.2: Particionamento fragmentado

as outras, ocupando mais tempo de CPU.
Fazendo uma abordagem *single-threaded*, é possivel controlar melhor o equilíbrio entre as cores uma vez que os nós são coloridos sequencialmente e facilmente contáveis. Por fim chegou-se a duas versões finais do algoritmo com performances já bastante boas:

- Largura-Primeiro *Single Threaded*
- Aleatório

4.1.1 Largura-Primeiro *Single Threaded*

Este algoritmo tenta formar a maior zona possível de uma cor e só passa para outra cor quando o número de nós coloridos é igual ao tamanho do overlay a dividir pelo número de cores. A execução é baseada em torno de uma lista que vai sendo alimentada com nós por colorir. Sempre que um nó é consumido este é colorido e são inseridos na cabeça da lista os seus vizinhos para que estes possam ser processados sequencialmente. O primeiro nó a ser inserido na lista é escolhido de forma aleatória. Se em algum ponto a lista ficar vazia, é escolhido outro nó sem cor e adicionado a esta

mesma lista e a execução continua com a cor seguinte. Se depois de circular todas as cores sobrarem nós por colorir, esta situação é corrigida adicionando a cor correspondente à maioria no conjunto dos seus vizinhos.
Para um overlay configurado com um número de vizinhos igual a $\log_{10} N$ sendo N o tamanho deste. Após vários ensaios este algoritmo atinge uma taxa de coesão na ordem dos **65%**.

4.1.2 Aleatório

A hipótese que apresenta um tempo de inicialização mas rápido é pura e simplesmente atribuir a cada nó uma cor aleatória do conjunto de cores disponíveis. Esta aproximação ao problema não tem em consideração a estrutura de vizinhos do overlay portanto não formará zonas coloridas compactas que sejam muito favoráveis. Para um overlay configurado com um número de vizinhos igual a $\log_{10} N$ sendo N o tamanho deste, este algoritmo atinge uma taxa de coesão aproximado aos **50%**. A grande vantagem desta alternativa de particionamento é a de ter um tempo de execução virtualmente instantâneo, pois tendo em conta que o overlay e a associação a vizinhos é inicializada aleatoriamente a cada execução da simulação, basta fazer uma divisão do conjunto total de nós e atribuir uma cor a cada divisão.

4.2 Processamento multithreaded sobre overlay

O processamento da simulação é efectuado usando uma thread por cada cor, de modo a que cada cor inicialize a actividade apenas no conjunto de nós da cor que lhe foi atribuída. Para facilitar isto, após efectuado o particionamento, o overlay deixa de ser representado apenas por uma pool de nós mas sim por várias pools associadas às respectivas cores (4.3).

4.2.1 Motor *cycle-driven*

No caso do motor *cycle-driven*, a adaptação para o processamento das várias cores por thread, foi bastante directa. Originalmente o simulador percorria todos os nós do over-

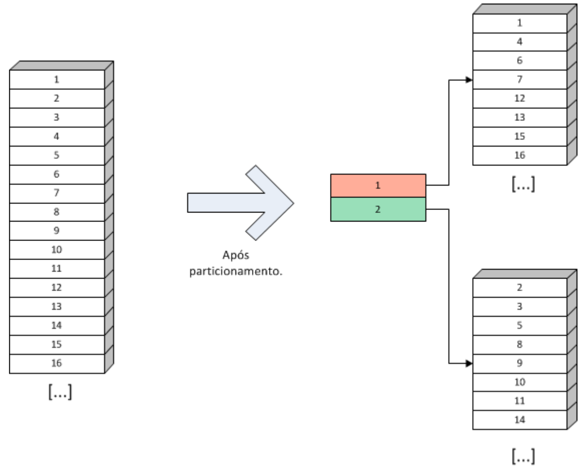


Figura 4.3: Overlay separado por cores

lay e activava o protocolo de interacção por cada um deles. Esta acção era repetida a cada ciclo até atingir o número de ciclos limite da simulação. Transpondo isto para o paradigma do overlay colorido, em vez de processar os nós todos a cada ciclo, são lançadas threads a cada ciclo (uma por cada cor) que processam o conjunto de nós que lhe pertencem. Após todas as threads terminarem, o simulador avança para o próximo ciclo e repete o processo. Dado esta necessidade de lançar threads a cada ciclo, esta solução não é eficiente para um número muito reduzido de nós, pois o custo de lançar as threads será superior ao próprio custo de processar todo o overlay. Quanto maior o custo de cada ciclo, ou a taxa de processamento efectuado a cada mensagem que é trocada entre nós, maior será a diferença de performance entre as duas versões. De modo a manter a funcionalidade de adição e remoção dinâmica de nós durante a simulação (sem uma perda de performance drástica), foi necessário usar listas não ordenadas na representação dos conjuntos de nós por cor descritos em (4.3). Como as regiões por cor são construídas consoante o resultado do algoritmo de particionamento, é necessário o uso deste tipo de lista (java.util.HashSet) para permitir a rápida adiçao e remoção de objectos. Ao fazer isto, foi eliminada a opção de percorrer os nós por ordem neste tipo de simulação, sendo apenas permitido o acesso aleatório, ou seja, a cada ciclo, o simulador não irá percorrer todos os nós por uma ordem específica, como era possível na versão original do peersim.

4.2.2 Motor *event-driven*

O motor *event-driven* é bastante mais complexo do que a versão *cycle-driven*. Em vez de haver um número definido de ciclos em que se percorrem todos os nós, o simulador é alimentado por uma *queue* (por vezes denominada por *heap*, designação que consideramos errónea) preenchida com eventos. Estes eventos significam a activação de um protocolo e podem estar associados a um nó ou a todo o overlay. Os eventos de Controle, referem-se à rede toda, e significam uma operação a efectuar sobre a totalidade do *overlay* (ex.: uma observação do estado do overlay, uma alteração ao número de nós, etc). O outro tipo de eventos bastante bem definido são os eventos de Ciclo. Embora o motor não seja baseado em ciclos, este mantém uma noção abstracta de ciclo que significa cada activação de um dado nó (ao contrário da noção de ciclo no motor *cycle-driven* que significa a activação consecutiva de todos os nós do *overlay*). Associado aos eventos está um valor que significa o tempo de activação do evento.

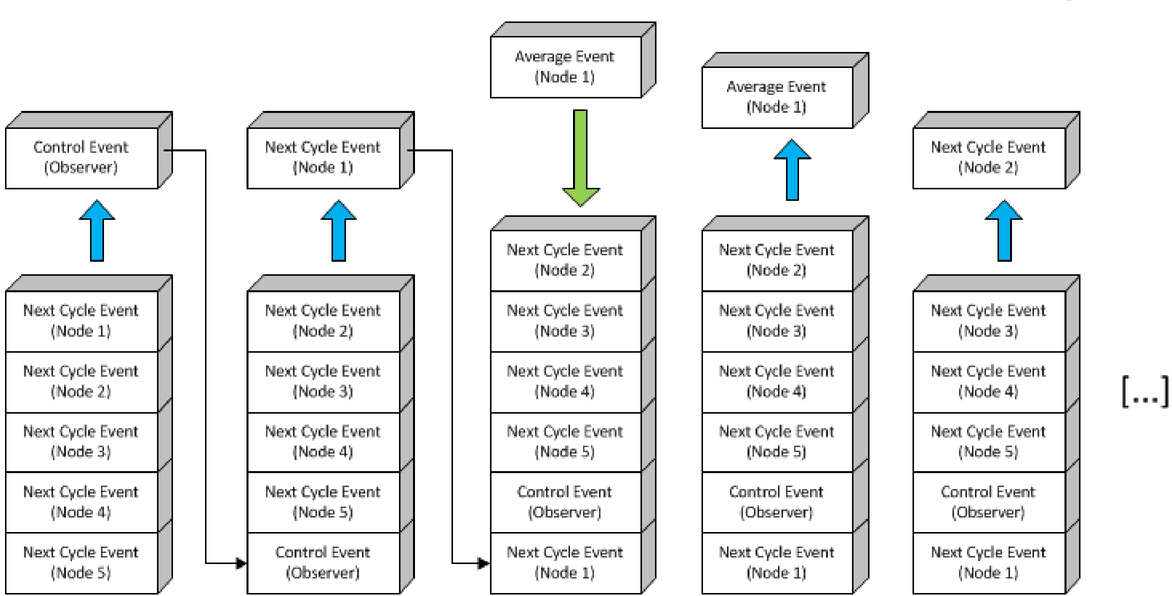


Figura 4.4: Evolução da *queue*

No inicio da simulação, a *queue* é carregada com todos os eventos de Controle (um evento por cada protocolo de controle a activar) e N eventos de Ciclo, sendo que N é igual ao número de nós do *overlay*. Estes eventos mantêm-se até ao final

da simulação, sendo que de cada vez que um deles é accionado, é enviado para o final da *queue* para voltar a ser accionado mais tarde. Da mesma forma, é somado um valor ao tempo de activação do evento de modo a corresponder ao novo tempo de activação. Quando esta soma é superior ao tempo limite da simulação, o evento não é reciclado fazendo com que a *queue* acabe por ficar vazia e a simulação termine. A complexidade do motor centra-se no scheduling de eventos, visto que o motor de execução simplesmente consome eventos da *queue* e executa o protocolo associado a ele até não ter mais eventos para consumir.

De cada vez que um evento de Ciclo é consumido, são introduzidos na *queue* N eventos, sendo N o número de protocolos de interacção a activar por cada nó, correspondendo normalmente ao envio de mensagens. Estes eventos quando são consumidos activam o protocolo definido e não são reciclados.

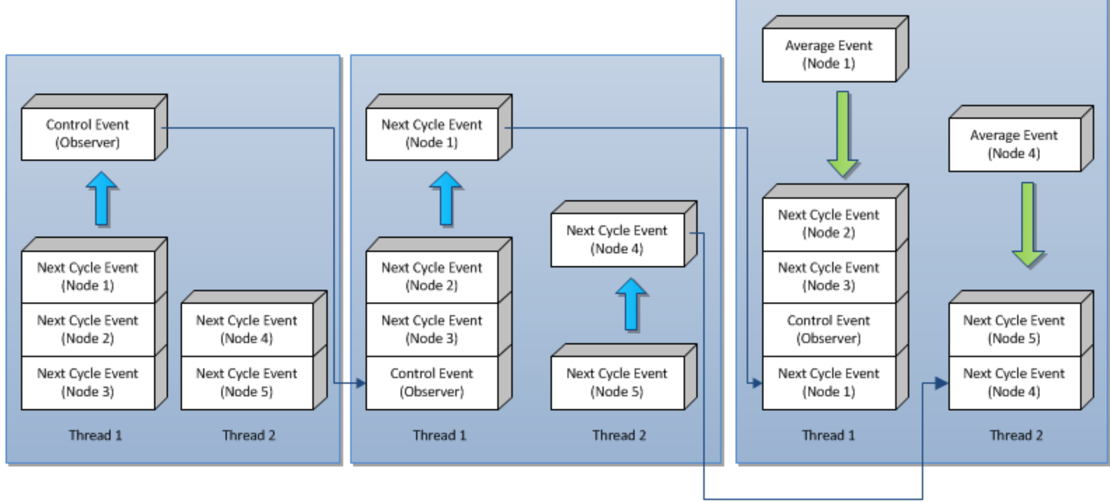


Figura 4.5: *Queue* modificada para *multithreading*

No exemplo ilustrado na figura 4.4 temos uma simulação configurada com 5 nós, um protocolo de controlo e um protocolo de interacção. O simulador consome o evento de controlo e envia-o para o fim da *queue*. De seguida, consome um evento de Ciclo associado ao nó 1 e envia-o para o fim da *queue*. De seguida é introduzido um evento relativo ao protocolo *AverageFunction* que irá fazer uma troca de valores com um vizinho do nó 1. Depois deste evento ser consumido, é descartado e posteriormente é consumido o evento de Ciclo relativo ao nó 2.

De modo a converter este motor para uma versão multithreaded, foi necessário transformar a *queue* de modo a que esta mantenha os eventos de ciclo e respectivos eventos de protocolo separados num conjunto de denominadas *mini-queues*, uma por cada cor. Os eventos de controlo ficam associados apenas a uma das threads/cores de modo a que sejam apenas executados uma vez. Deste modo, o motor pode ter as várias threads a correr desde o inicio da simulação a consumir eventos em vez de estar a iniciá-las a cada ciclo. Para que nenhuma thread se adiante demasiado relativamente ás outras, as threads param à espera das outras sempre que consumirem um número de eventos de ciclo igual ao número de nós. Na prática isto constitui uma barreira de sincronização na qual todas as threads se sincronizam de x em x eventos. Em 4.5 podemos ver a evolução do exemplo figurado em 4.4 em que o consumo dos eventos é efectuado pelas várias threads na *queue* modificada, sendo que os eventos de controlo só existem associados à primeira thread.

4.3 Sincronização da actividade entre nós

Uma vez que são os protocolos de interacção que escolhem o que fazer em cada nó do overlay, são estes também que vão determinar quais os nós de destino da interacção. Para que o processamento paralelo não tenha problemas, a sincronização tem que ser feita directamente neste ponto. Infelizmente os protocolos de interacção não são parte integrante do núcleo do peersim e são na maioria dos casos extensões. De modo a tentar manter a correcção de possíveis extensões ao peersim, a hierarquia de classes destes protocolos foi alterada para que exista uma especificação para a criação de novos protocolos que façam a correcta sincronização. Foi criada uma classe nova que define um protocolo de interacção, com um método **synchronizedNodeAccess** (que não pode ser *overloaded*) e que invoca o método abstracto **activateNodeInteraction** que tem que ser implementado em todos os protocolos. Para que a nova versão do peersim seja utilizada correctamente, o método **nextCycle** (que é chamado pelo simulador) apenas deve escolher qual ou quais os nós destino com quem comunicar e invocar o método **synchronizedNodeAccess** sendo que a actividade entre ambos os nós deve

ficar compartimentalizada em **activateNodeInteraction**. (Antes: 4.6 Depois: 4.7)

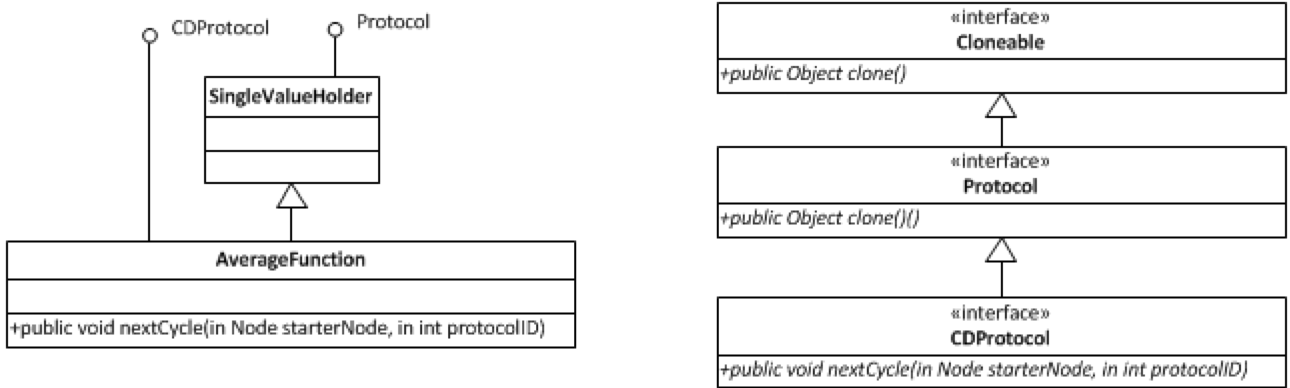


Figura 4.6: Estrutura original

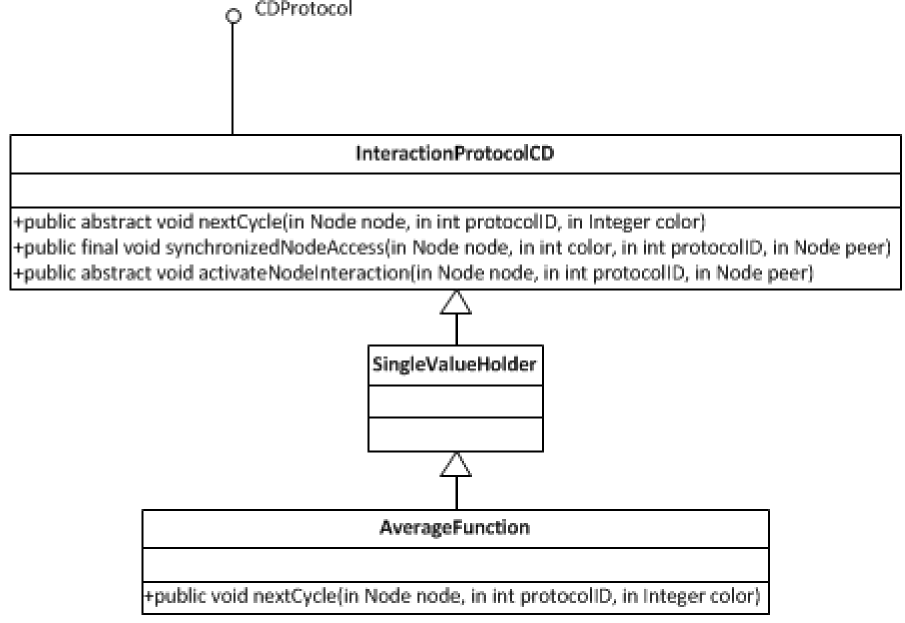


Figura 4.7: Estrutura alterada

4.4 Execução distribuída

Quando a simulação está em modo distribuído (sobre o terracotta) foi tomada a decisão de limitar cada participante a uma côr/thread de modo a simplificar o trabalho sem sacrificar os objectivos. Isto permitiu que fosse possível ter um protótipo a correr distribuído sobre o terracotta sem ter que adicionar a complexa gestão das várias cores

por participante e da sincronização extra sobre a sincronização do processamento *multithreaded* em cada um dos participantes. Do mesmo modo, apenas o motor *cycle-driven* foi implementado sobre o Terracotta.
Quando o simulador corre sobre o terracotta, este assume-se como coordenador da simulação se não houver mais nenhum participante ligado. A partir deste momento, todos os restantes participantes serão escravos do primeiro, na medida em que só efectuam processamento, mas não coordenação. Cabe ao coordenador reger o carregamento distribuído, efectuar o particionamento e iniciar a simulação em si. Na figura 4.8 é descrita a máquina de estados da simulação distribuída.

4.5 Sincronização da execução distribuída

Como referido anteriormente, o Terracotta faz uso de uma arquitectura cliente-servidor para ser possivel ter várias VM Java (participantes) em comunicação. O servidor/coordenador é iniciado num processo exterior ao peersim, e a aplicação a correr neste ambiente distribuído tem que ser executada por meio de um executável especifico fornecido pelo Terracotta e não usando directamente o executável da máquina virtual Java. Este *wrapper* faz instrumentação das classes das aplicações a partilhar (de modo a inserir as funcionalidades de partilha de memória) e de seguida coloca a aplicação a correr directamente na VM Java.
As funcionalidades de partilha de memória do Terracotta são baseadas numa simples configuração XML que é carregada pelo *wrapper*. A partilha de memória é efectuada ao nível de instâncias específicas de objectos. Usando expressões regulares, é possível configurar o Terracotta para que uma variável estática partilhada mantenha o mesmo valor em todos os participantes. Estes pontos de partilha são chamados de *roots*. Qualquer alteração feita por um dos participantes a uma variável ou campo acessivel a partir da instância partilhada é automaticamente sincronizada pelo servidor/coordenador para os restantes participantes. Eis um exemplo da configuração:

```
<roots>
    <root>
```

```
      <field-name>peersim.core.Network.regions</field-name>
   </root>
</roots>
```

Neste exemplo, o Terracotta irá fazer com que da primeira vez que for atribuída uma referência à variável **regions** na classe **peersim.core.Network** esta fique atribuída para todos os participantes e não possa ser mais alterada. Se a variável partilhada for uma colecção, qualquer participante pode adicionar objectos à colecção que estes ficarão acessiveis a todos os outros.
No caso do PC-Peersim, as estruturas a partilhar são as colecções e mapas que representam o conjunto de Nós que formam o *overlay* a simular, informação sobre os participantes registados para partilhar a simulação, os dados do particionamento do *overlay* e todo o tipo de locks e barreiras necessários de modo a sincronizar a actividade dos vários participantes. Por cada variável partilhada, o Terracotta obriga a que todos os seus acessos sejam sincronizados ou então lançará uma excepção. Para que não seja necessário alterar o codigo de modo a sincronizar o acesso a uma variável, que na versão original do peersim não necessitaria de tal cuidado, o Terracotta oferece uma configuração de locks a serem colocados em runtime. De novo, a configuração é feita por meio de expressões regulares:

```
<named-lock>
   <method-expression>
* example.aggregation.AverageFunction.activateNodeInteraction(..)
   </method-expression>
   <lock-level>write</lock-level>
   <lock-name>activateNodeInteraction</lock-name>
</named-lock>
```

Neste exemplo, o Terracotta irá aplicar sincronização a todas as chamadas a métodos na classe **example.aggregation.AverageFunction.activateNodeInteraction** cujo nome seja **activateNodeInteraction** independentemente do seu retorno e dos argumentos que o método necessita. Este lock em questão foi necessário porque este

efectua alterações em instâncias que são acessíveis por todos os participantes a partir de um *root*.
Deste modo é possivel transformar a execução do Peersim, numa versão distribuída, em que os participantes carregam apenas parte do *overlay* de modo a partilhar a carga em memória, e executam a simulação em apenas uma partição do mesmo. Em apêndice, está incluído o XML de configuração do PC-Peersim.

4.6 Resumo

Neste capítulo foram explicados alguns detalhes de implementação de maior relevância na realização deste projecto. Foram detalhadas as várias abordagens feitas ao algoritmo de particionamento, e quais os maiores entraves encontrados nas variantes *multi-threaded*. Foi explicado em detalhe o funcionamento do algoritmo final *single-threaded* e as vantagens do uso de um particionamento aleatorio. Foram detalhados as representações em memória do overlay particionado como foram feitas as alterações ao motor de simulação para que este atingisse o objectivo proposto, tanto na sua versão *cycle-driven* como na sua versão *event-driven*, sendo que no caso da versão *event-driven* foi demonstrado o funcionamento da nova *queue* de eventos capaz de suportar o processamento das várias partições em separado.
Por fim, temos uma introdução aos detalhes do funcionamento do Terracotta, das suas capacidades e de como é efectuada a partilha de memória e sincronização entre todos os participantes.

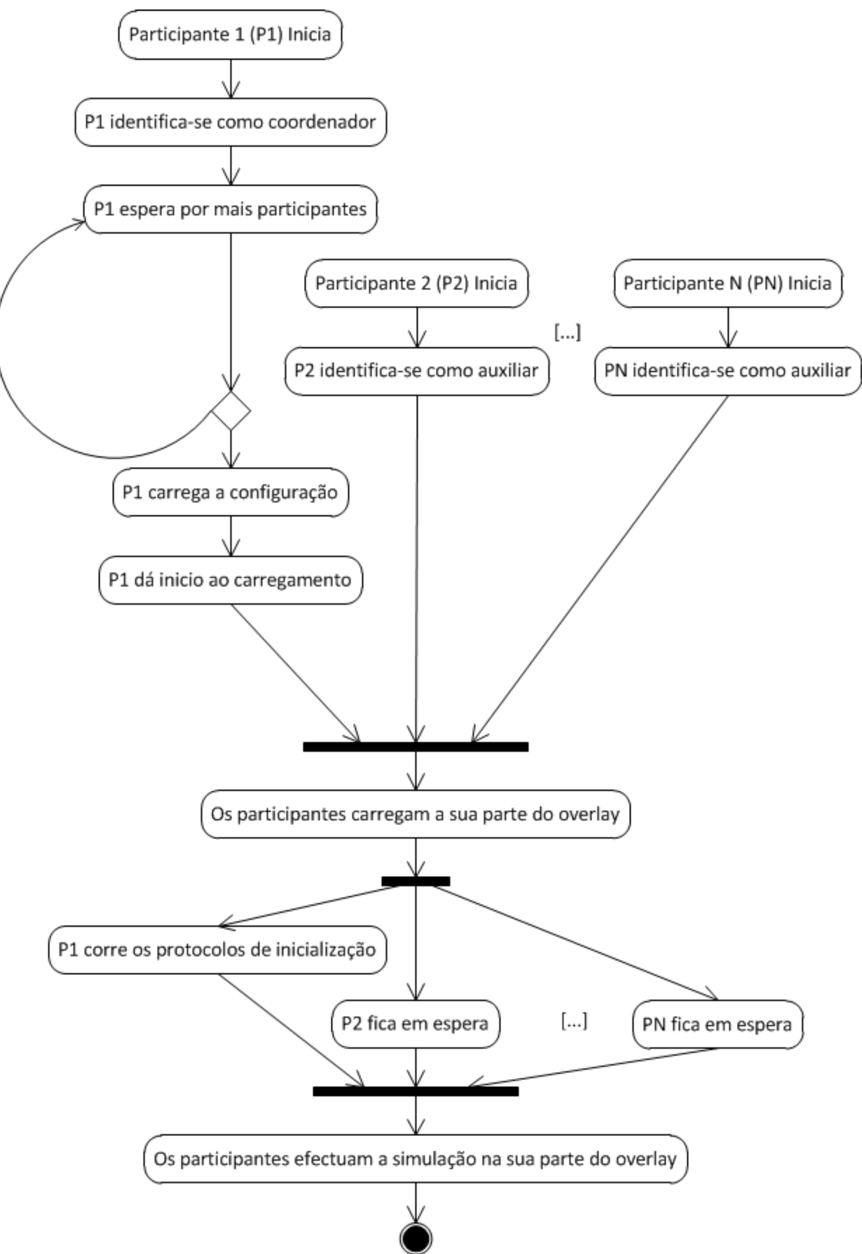


Figura 4.8: Execução distribuída

5 Avaliação dos Resultados

Neste capítulo apresentamos os resultados dos testes efectuados ao PC-Peersim num conjunto de configurações de simulação que ilustram variadas situações em que este sistema pode ser usado. Os testes foram efectuados usando uma máquina dual-core de modo a poder ser analizado um verdadeiro ganho de performance face à versão original. Todos os testes são também comparados com os resultados obtidos usando a versão *single-threaded* do sistema.

5.1 Testes

Os testes foram efectuados usando dois valores para o número de nós do *overlay* e dois valores para a extensão do tempo de simulação, tanto na simulação base do motor *event-driven* como no motor *cycle-driven*. Em ambos os motores foram feitos testes usando os protocolos base do Peersim e são baseados na simulação de exemplo nº 2 incluida na distribuição do Peersim .Esta simulação usa o protocolo interno do Peersim WireKout e um protocolo de interacção chamado AverageFunction que mantém um valor por cada nós e que a cada troca de mensagem, calcula a média entre os valores dos dois nós e atribui esse valor aos mesmos. Adicionalmente nesta simulação, são também testadas as alternativas de algoritmos de particionamento.
Além dos testes com o exemplo base do Peersim, foram incluidos no PC-Peersim implementações do Chord e do Pastry, tendo sido estes testados como exemplo de uma aplicação mais realista do Peersim e como exemplo tambem de simulação de protocolos externos ao *Peersim*. Estes exemplos foram testados não só com uma e duas threads, mas também com quatro para explorar a performance destes protocolos mais complexos com mais *threads* disponíveis.

Por fim serão efectuados testes sobre a implementação distribuída usando dois participantes, tanto sobre apenas uma máquina como sobre um cluster de duas máquinas. Este teste faz uso novamente da simulação exemplo 2 na sua versão *cycle-driven*. O teste sobre o cluster foi efectuado com duas máquinas numa rede LAN de 100 Mbits/s. O servidor Terracotta foi executado na mesma máquina que um dos participantes. Os testes serão portanto:

- Paralelização
  - *Event-driven*, Simulação exemplo 2, 1 thread
  - *Event-driven*, Simulação exemplo 2, 2 threads, Particionamento Aleatório
  - *Event-driven*, Simulação exemplo 2, 2 threads, Particionamento por Adjacência
  - *Event-driven*, Simulação Chord, 1 thread
  - *Event-driven*, Simulação Chord, 2 threads, Particionamento Aleatório
  - *Event-driven*, Simulação Chord, 4 threads, Particionamento Aleatório
  - *Event-driven*, Simulação Pastry, 1 thread
  - *Event-driven*, Simulação Pastry, 2 threads, Particionamento Aleatório
  - *Event-driven*, Simulação Pastry, 4 threads, Particionamento Aleatório
  - *Cycle-driven*, Simulação exemplo 2, Peersim Original
  - *Cycle-driven*, Simulação exemplo 2, PC-Peersim, 2 threads, Particionamento Aleatório
  - *Cycle-driven*, Simulação exemplo 2, PC-Peersim, 2 threads, Particionamento por Adjacência
- Distribuição
  - *Cycle-driven*, Simulação exemplo 2, 2 participantes sobre a mesma máquina, Particionamento Aleatório
  - *Cycle-driven*, Simulação exemplo 2, 2 participantes sobre duas máquinas, Particionamento Aleatório

Em todos os testes, os resultados serão apresentados usando o tempo necessário para completar a simulação face ao tamanho do *overlay* virtual e número de ciclos/tempo limite configurados.
O principal objectivo a atingir é o de comprovar que a simulação é sempre mais rápida e eficiente no tempo de execução face a à versão original e que consegue vencer as barreiras de limite de memória ao tentar simular *overlays* maiores sobre um cluster.

5.2 Resultados da simulação paralelizada

5.2.1 *Event-driven*, Simulação exemplo 2

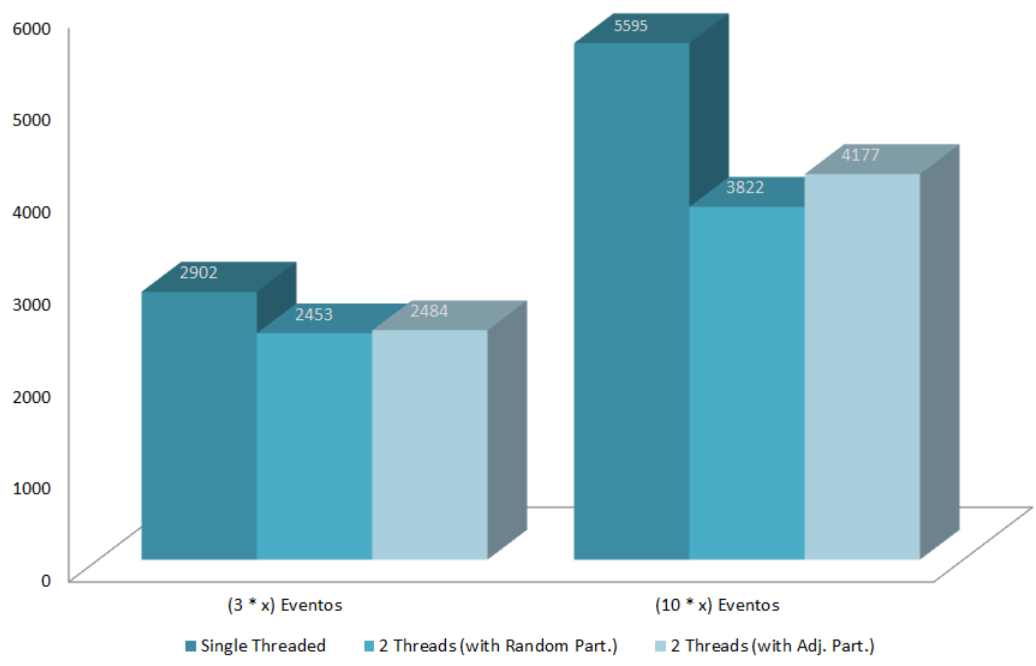


Figura 5.1: Comparação de médias de execução para o exemplo 2 no motor *Event-Driven* para 250.000 nós

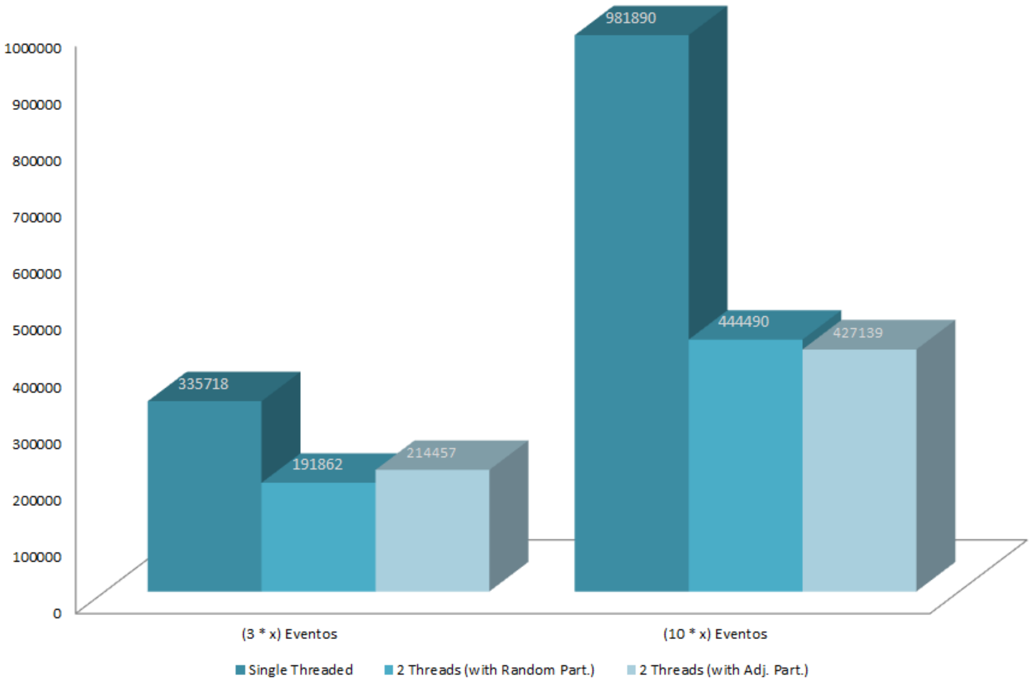


Figura 5.2: Comparação de médias de execução para o exemplo 2 no motor *Event-Driven* para 2.500.000 nós

5.2.1.1 *Single Thread*

	$(6 * x)$ **Eventos**		$(20 * x)$ **Eventos**	
	$x =$ **25000 Nós**	$x =$ **2500000 Nós**	$x =$ **25000 Nós**	$x =$ **2500000 Nós**
Ensaio 1	2914 ms	334264 ms	5492 ms	958411 ms
Ensaio 2	2869 ms	342367 ms	5787 ms	997352 ms
Ensaio3	2924 ms	330524 ms	5570 ms	989907 ms
Média	2902 ms	335718 ms	5595 ms	981890 ms
	00:00:02.902	**00:05:35.718**	**00:00:05.595**	**00:16:21.890**

5.2.1.2 2 *Threads*, Particionamento Aleatório

	$(6 * x)$ **Eventos**		$(20 * x)$ **Eventos**	
	$x =$ **25000 Nós**	$x =$ **2500000 Nós**	$x =$ **25000 Nós**	$x =$ **2500000 Nós**
Ensaio 1	2783 ms	192404 ms	3744 ms	472552 ms
Ensaio 2	2399 ms	188968 ms	3853 ms	426302 ms
Ensaio3	2178 ms	194215 ms	3869 ms	434616 ms
Média	2453 ms	191862 ms	3822 ms	444490 ms
	00:00:02.453	**00:03:11.862**	**00:00:03.822**	**00:07:24.490**

5.2.1.3 2 *Threads*, Particionamento por Adjacência

		$(6 * x)$ **Eventos** ($x =$ nº de Nós)		$(20 * x)$ **Eventos** ($x =$ nº de Nós)	
		25000 Nós	**2500000 Nós**	**25000 Nós**	**2500000 Nós**
Ensaio 1	Part.	130 ms	4055 ms	140 ms	5320 ms
	Sim.	2543 ms	210847 ms	3744 ms	472552 ms
	Total	2673 ms	214902 ms	4024 ms	429328 ms
Ensaio 2	Part.	119 ms	3988 ms	140 ms	5414 ms
	Sim.	2357 ms	210751 ms	4056 ms	416661 ms
	Total	2476 ms	214739 ms	4196 ms	422075 ms
Ensaio 3	Part.	113 ms	4005 ms	140 ms	5382 ms
	Sim.	2190 ms	209726 ms	3993 ms	430014 ms
	Total	2303 ms	213731 ms	4133 ms	427139 ms
Média	Total	2484 ms	214457 ms	3822 ms	427139 ms
		00:00:02.484	**00:03:34.457**	**00:00:03.822**	**00:07:24.490**

5.2.2 Análise

O motor *event-driven* mostrou-se um ganho de performance interessante. Ao contrário da versão *cycle-driven* este motor mantém o número de threads configuradas em permanente execução, alimentando-se do conjunto de eventos disponíveis nas *queues*. Este facto permite que a simulação consiga ser igualmente eficiente em *overlays* de baixa

(Fig. 5.1) e grande dimensão (Fig. 5.2), sendo que a diferença entre as performances é mais notória no *overlay* configurado com 2.500.000 nós. Quando maior o custo computacional de cada ciclo e de cada protocolo, maior será a possibilidade das threads ocuparem 100% do CPU ou core utilizado pela thread.

No que toca ao particionamento, a versão Aleatória revelou-se bastante mais eficiente em quase todos os casos, sendo que o ligeiro ganho sentido no teste mais longo usando 2.500.000 nós (Fig. 5.2) pode indicar que esta versão do processo seja mais interessante em grandes simulaçoes.

5.2.3 *Event-driven*, Simulação Chord

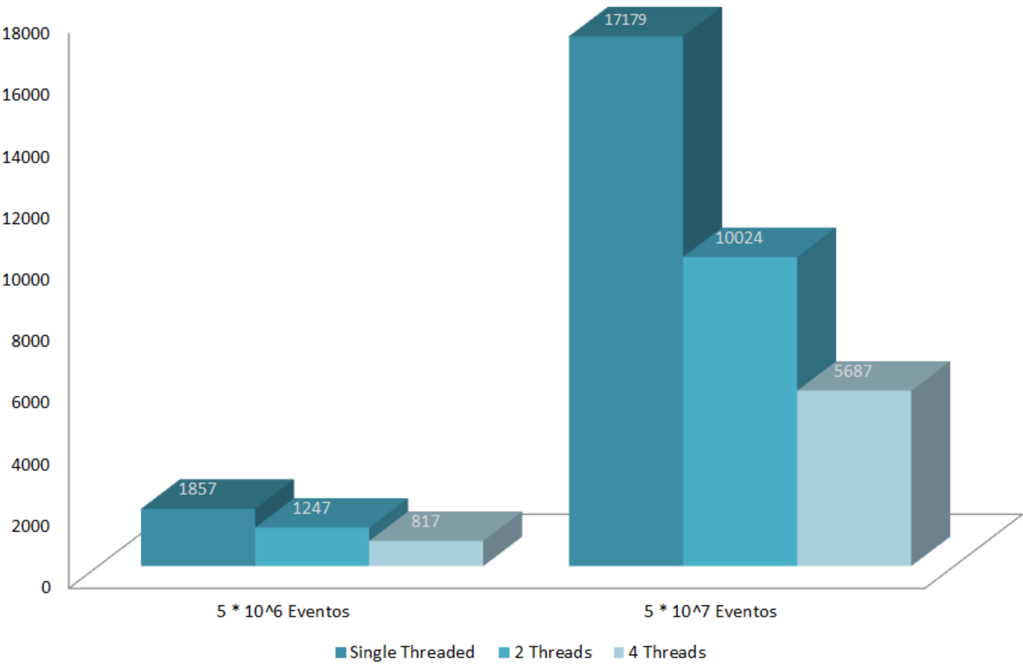


Figura 5.3: Comparação de médias de execução para o exemplo Chord no motor *Event-Driven* para 5.000 nós

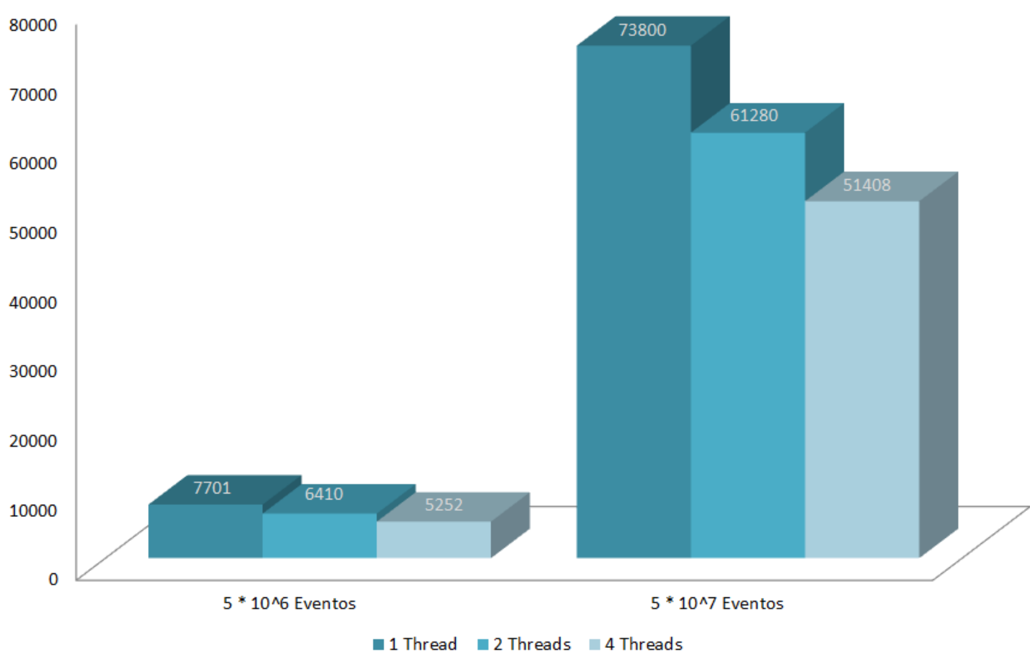


Figura 5.4: Comparação de médias de execução para o exemplo Chord no motor *Event-Driven* para 250.000 nós

5.2.3.1 *Single Thread*

	$5 * 10^6$ **Eventos**		$5 * 10^7$ **Eventos**	
	5000 Nós	**250000 Nós**	**5000 Nós**	**250000 Nós**
Ensaio 1	1817 ms	7784 ms	16231 ms	74766 ms
Ensaio 2	1868 ms	7628 ms	16158 ms	73581 ms
Ensaio3	1886 ms	7691 ms	16334 ms	73054 ms
Média	1857 ms	7701 ms	17179 ms	73800 ms
	00:00:01.857	**00:00:07.701**	**00:00:17.179**	**00:01:13.800**

5.2.3.2 2 threads, Particionamento Aleatório

	$5 * 10^6$ **Eventos**		$5 * 10^7$ **Eventos**	
	5000 Nós	**250000 Nós**	**5000 Nós**	**250000 Nós**
Ensaio 1	1283 ms	6420 ms	9928 ms	61308 ms
Ensaio 2	1214 ms	6412 ms	10133 ms	61548 ms
Ensaio3	1244 ms	6399 ms	10011 ms	60984 ms
Média	1247 ms	6410 ms	10024 ms	61280 ms
	00:00:01.247	**00:00:06.410**	**00:00:10.24**	**00:01:01.280**

5.2.3.3 4 threads, Particionamento Aleatório

	$5 * 10^6$ **Eventos**		$5 * 10^7$ **Eventos**	
	5000 Nós	**250000 Nós**	**5000 Nós**	**250000 Nós**
Ensaio 1	824 ms	5299 ms	5692 ms	51277 ms
Ensaio 2	839 ms	5245 ms	5751 ms	51834 ms
Ensaio3	789 ms	5213 ms	5618 ms	51114 ms
Média	817 ms	5252 ms	5687 ms	51408 ms
	00:00:00.817	**00:00:05.252**	**00:00:05.687**	**00:00:51.408**

5.2.4 Análise

Nos testes efectuados com uma simulação do protocolo Chord, os resultados foram muito interessantes, sentindo-se um grande ganho de performance em todos os casos. É de notar que tanto esta como a simulação do protocolo Pastry têm um custo computacional ao nível da execução das interacções muito superior á execução do simples cálculo da média no exemplo 2. Este factor faz com que o simulador ocupe efectivamente mais tempo de CPU na execução dos protocolos em si do que em sincronização e rotação de eventos.

Não foram efectuados testes com Particionamento por adjacência porque esta implementação de Chord não respeita as interfaces do Peersim a 100% no que toca à concretização das estruturas que representam as relações de vizinhança entre os nós.

5.2.5 *Event-driven*, Simulação Pastry

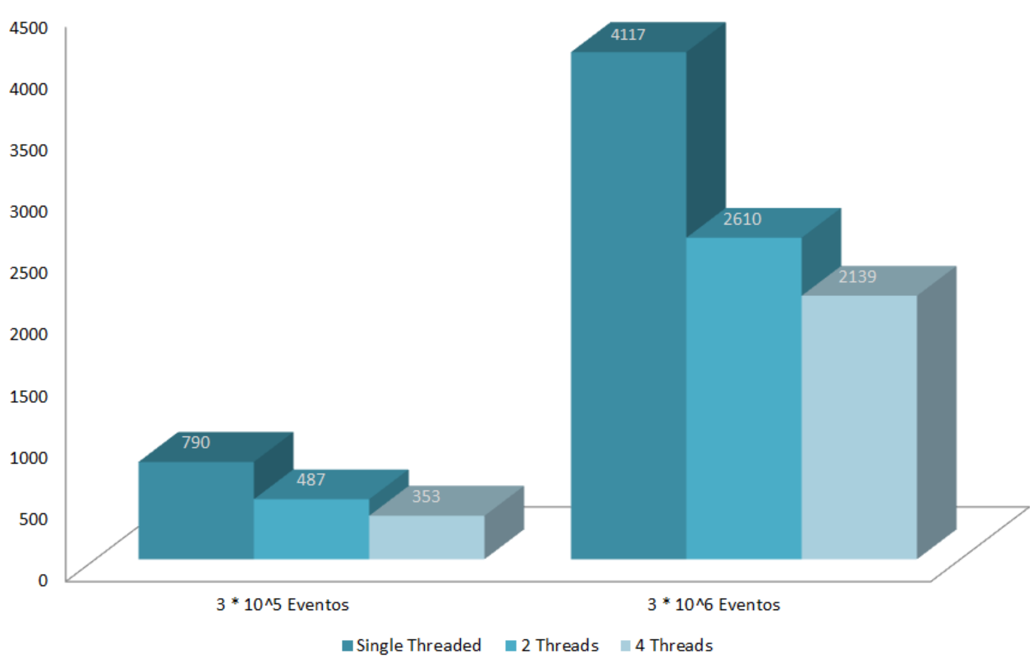


Figura 5.5: Comparação de médias de execução para o exemplo Pastry no motor *Event-Driven* para 50 nós

5.2.5.1 *Single Thread*

	$3 * 10^5$ **Eventos**		$3 * 10^6$ **Eventos**	
	50 Nós	**5000 Nós**	**50 Nós**	**5000 Nós**
Ensaio 1	764 ms	6822 ms	4128 ms	153142 ms
Ensaio 2	733 ms	6895 ms	4172 ms	153254 ms
Ensaio3	874 ms	6791 ms	4053 ms	149849 ms
Média	790 ms	6836 ms	4117 ms	152081 ms
	00:00:00.790	**00:00:06.836**	**00:00:17.179**	**00:03:52.81**

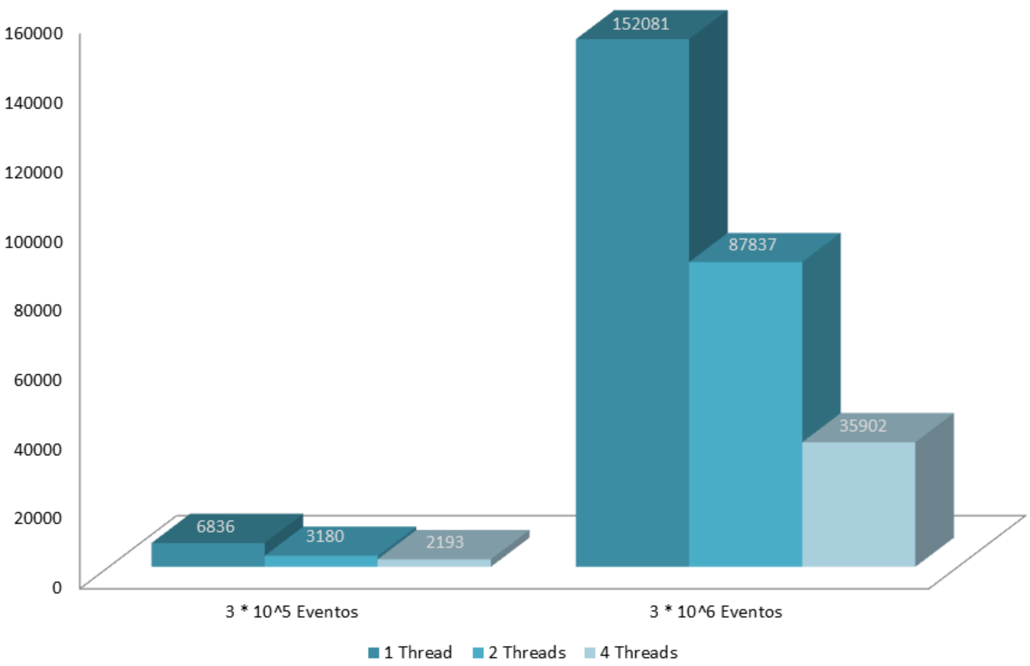


Figura 5.6: Comparação de médias de execução para o exemplo Pastry no motor *Event-Driven* para 5.000 nós

5.2.5.2 2 threads, Particionamento Aleatório

	$3 * 10^5$ **Eventos**		$3 * 10^6$ **Eventos**	
	50 Nós	**5000 Nós**	**50 Nós**	**5000 Nós**
Ensaio 1	431 ms	3062 ms	2324 ms	86721 ms
Ensaio 2	507 ms	3158 ms	2377 ms	86597 ms
Ensaio3	524 ms	3321 ms	3130 ms	90195 ms
Média	487 ms	3180 ms	2610 ms	87837 ms
	00:00:00.487	**00:00:03.180**	**00:00:02.610**	**00:01:27.837**

5.2.5.3 4 threads, Particionamento Aleatório

	$3 * 10^5$ **Eventos**		$3 * 10^6$ **Eventos**	
	50 Nós	**5000 Nós**	**50 Nós**	**5000 Nós**
Ensaio 1	346 ms	2197 ms	2092 ms	35136 ms
Ensaio 2	324 ms	2259 ms	2205 ms	36773 ms
Ensaio3	389 ms	2124 ms	2120 ms	35798 ms
Média	353 ms	2193 ms	2139 ms	35902 ms
	00:00:00.353	**00:00:02.193**	**00:00:17.179**	**00:01:13.800**

5.2.6 Análise

À semelhança dos testes com a simulação Chord, o mesmo ganho de performance foi sentido na simulação de Pastry. Observando os tempos necessários para simular uma rede de 5000 nós, conclui-se que tambem estamos na presença de um protocolo com grande custo computacional.
À semelhança da simulação Chord, não foram efectuados testes com Particionamento por adjacência.

5.2.7 *Cycle-driven*, Simulação Exemplo 2

5.2.7.1 *Single Thread*

	30 Ciclos		**150 Ciclos**	
	25000 Nós	**2500000 Nós**	**25000 Nós**	**2500000 Nós**
Ensaio 1	259 ms	112460 ms	327 ms	537850 ms
Ensaio 2	265 ms	108352 ms	311 ms	532923 ms
Ensaio3	275 ms	109244 ms	411 ms	549341 ms
Média	266 ms	110018 ms	349 ms	540038 ms
	00:00:00.266	**00:01:50.18**	**00:00:00.349**	**00:09:00.38**

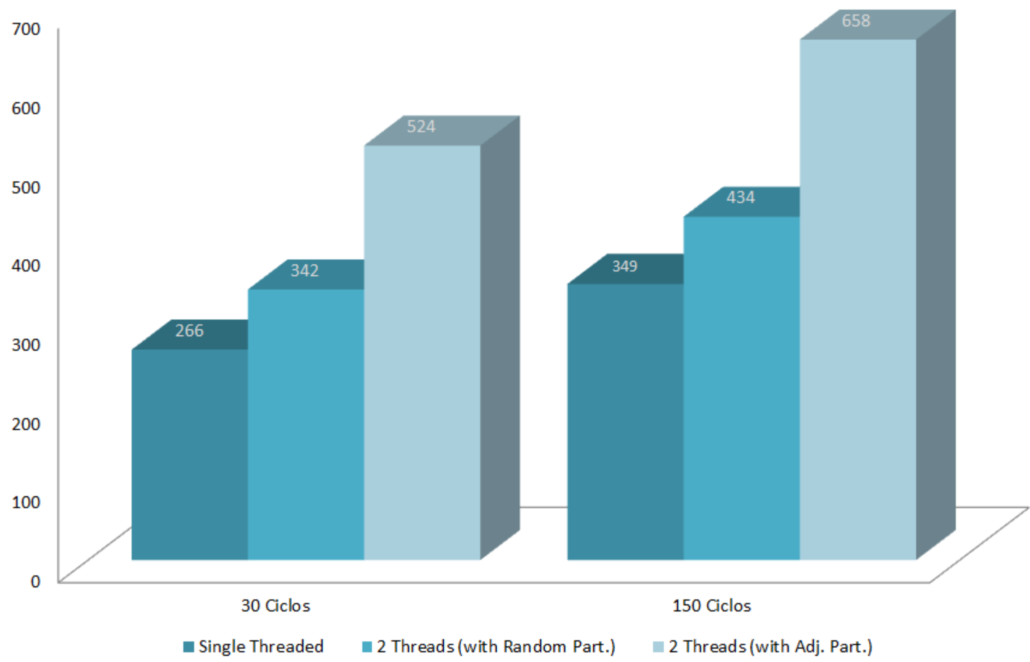


Figura 5.7: Comparação de médias de execução para o exemplo 2 no motor *Cycle-Driven* para 250.000 nós

5.2.7.2 2 *Threads*, Particionamento Aleatório

	30 Ciclos		150 Ciclos	
	25000 Nós	**2500000 Nós**	**25000 Nós**	**2500000 Nós**
Ensaio 1	326 ms	91291 ms	438 ms	426001 ms
Ensaio 2	388 ms	93452 ms	418 ms	415837 ms
Ensaio3	312 ms	92001 ms	446 ms	415271 ms
Média	342 ms	92248 ms	434 ms	419036 ms
	00:00:00.342	**00:01:32.248**	**00:00:00.434**	**00:06:59.36**

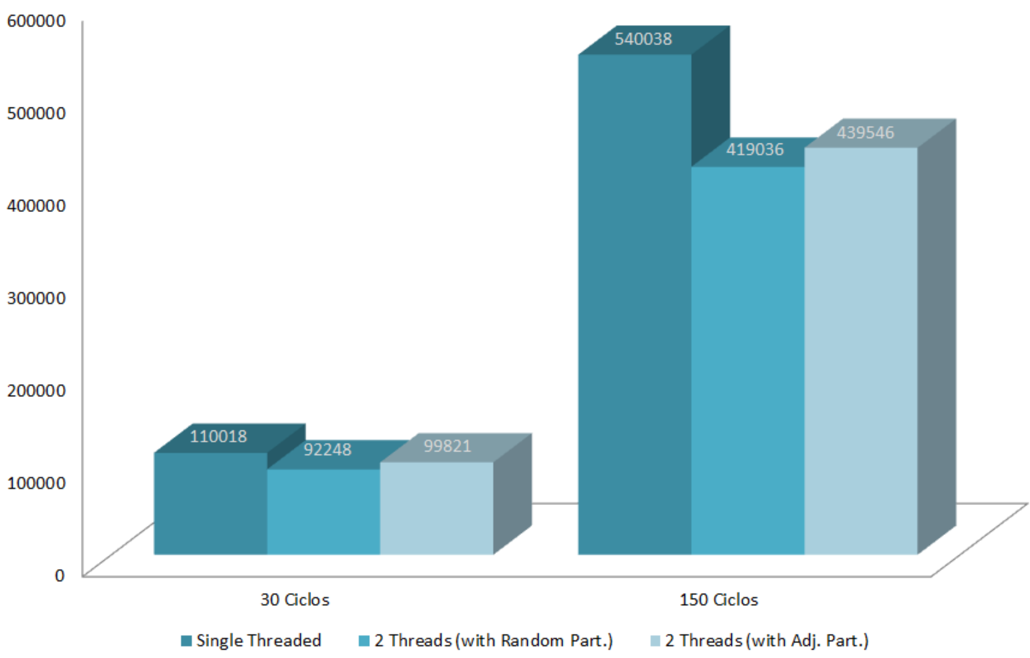


Figura 5.8: Comparação de médias de execução para o exemplo 2 no motor *Cycle-Driven* para 2.500.000 nós

5.2.7.3 2 *Threads*, Particionamento por Adjacência

		30 Ciclos		**150 Ciclos**	
		25000 Nós	**2500000 Nós**	**25000 Nós**	**2500000 Nós**
Ensaio 1	Part.	176 ms	6989 ms	182 ms	5867 ms
	Sim.	339 ms	91603 ms	508 ms	434573 ms
	Total	515 ms	98592 ms	690 ms	440440 ms
Ensaio 2	Part.	169 ms	6895 ms	173 ms	5460 ms
	Sim.	323 ms	91775 ms	465 ms	437514 ms
	Total	492 ms	98670 ms	638 ms	442974 ms
Ensaio 3	Part.	202 ms	6879 ms	183 ms	5707 ms
	Sim.	365 ms	95323 ms	464 ms	429518 ms
	Total	567 ms	102202 ms	647 ms	435225 ms
Média	Total	524 ms	99821 ms	658 ms	439546 ms
		00:00:00.524	**00:01:39.821**	**00:00:00.658**	**00:07:19.546**

5.2.8 Análise

O motor *cycle-driven* mostrou-se menos eficiente do que esperado. A concretização deste motor passou por duas versões durante o seu desenvolvimento. Originalmente, o motor lançava apenas um número de threads configurado e estas iam percorrendo os ciclos configurados, sendo que se sincronizavam ao fim de cada ciclo. No final, acabou por se revelar mais eficiente lançar este conjunto de threads todos os ciclos e eliminando a necessidade de sincronização a cada ciclo. De alguma forma, o custo por parte da VM de lançar uma thread nova acabou por superar as ineficiências da barreira usada para sincronizar as threads. Este ganho só é notório, no entanto, em simulações de maior escala, em que o custo de lançar estas threads seja absorvido pelo peso de simular cada ciclo. Isto é reflectido nos testes pela perda de performance na simulação *cycle-driven* para 250.000 nós (Fig. 5.7). Nós testes efectuados com um *overlay* 100 vezes maior, o ganho performance é bastante notório (Fig. 5.8).

5.3 Resultados da simulação distribuída

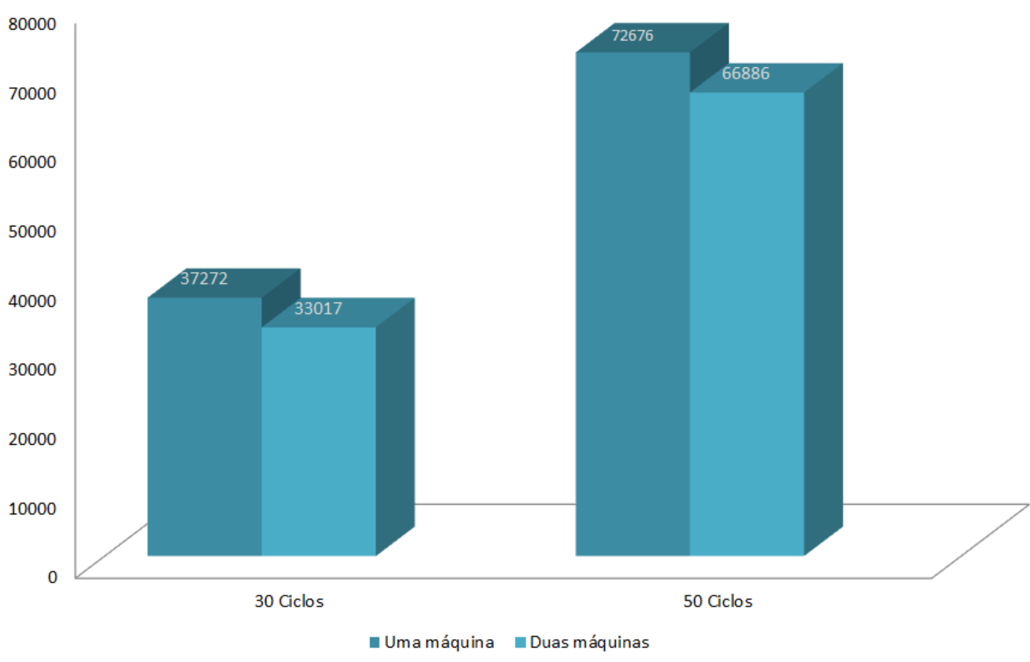


Figura 5.9: Comparação de médias de execução para o exemplo 2 no motor *Cycle-Driven* a correr uma simulação com 50 nós

5.3.1 *Cycle-driven*, Simulação exemplo 2, PC-Peersim, 2 participantes na mesma máquina, Particionamento Aleatório

	30 Ciclos		**60 Ciclos**	
	x = **50 Nós**	x = **500 Nós**	x = **50 Nós**	x = **500 Nós**
Ensaio 1	38946 ms	253080 ms	74024 ms	342579 ms
Ensaio 2	36746 ms	278495 ms	72185 ms	321579 ms
Ensaio3	36124 ms	257354 ms	71820 ms	325235 ms
Média	37272 ms	262976 ms	72676 ms	329797 ms
	00:00:37.272	**00:04:22.976**	**00:01:12.676**	**00:05:29.797**

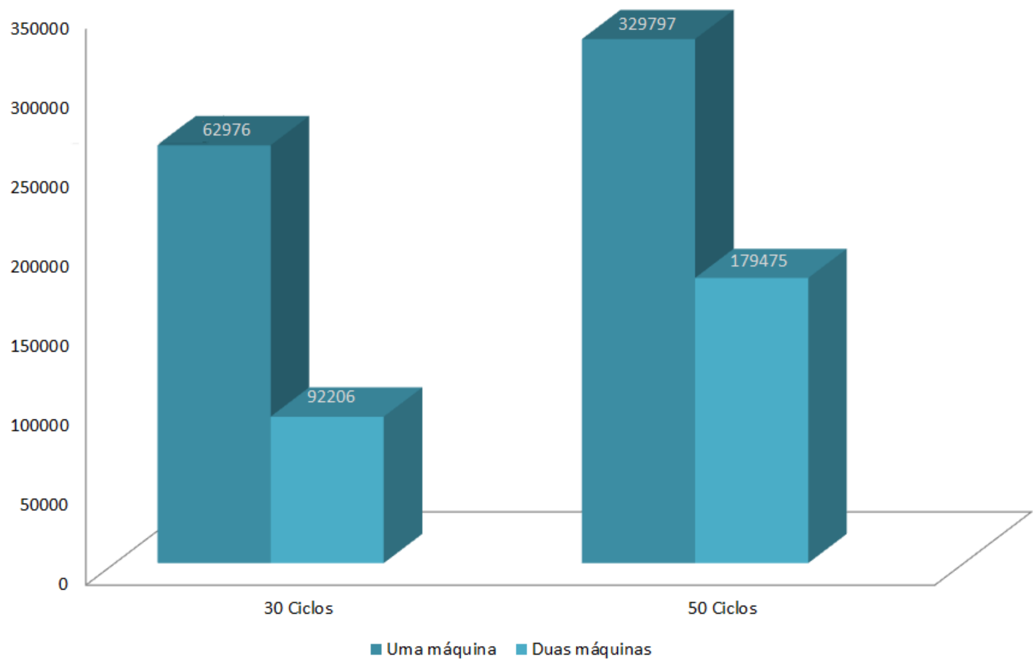


Figura 5.10: Comparação de médias de execução para o exemplo 2 no motor *Cycle-Driven* a correr uma simulação com 500 nós

5.3.2 *Cycle-driven*, Simulação exemplo 2, PC-Peersim, 2 participantes em máquinas diferentes, Particionamento Aleatório

	30 Ciclos		**60 Ciclos**	
	x = **50 Nós**	x = **500 Nós**	x = **50 Nós**	x = **500 Nós**
Ensaio 1	34240 ms	91290 ms	65104 ms	182410 ms
Ensaio 2	32573 ms	92486 ms	69057 ms	178021 ms
Ensaio3	32239 ms	92843 ms	66497 ms	177995 ms
Média	33017 ms	92206 ms	66886 ms	179475 ms
	00:00:33.17	**00:01:32.206**	**00:01:06.886**	**00:02:59.475**

5.3.3 Análise

Contrariamente à simulação apenas numa máquina, a versão distribuída sobre o Terracotta revelou-se extremamente ineficiente e lenta. Na mesma ordem de grandeza de tempo em que conseguimos correr 30 ciclos num *overlay* com 2.500.000 apenas numa

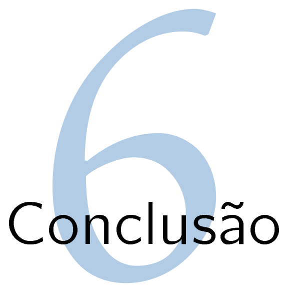
máquina, corremos 60 ciclos num *overlay* com 50 nós na versão distribuída.
Este overhead enorme relacionado com o uso do Terracotta pode-se explicar pelo acumular de alguns motivos:

- O peersim faz uso de um vector de Objectos para representar o conjunto de nós do *overlay*. Este vector é um dos objectos configurados como *root* do Terracotta, e pode-se conjecturar que o Terracotta não consegue partilhar de forma eficiente um vector de objectos sem o replicar para todos os participantes mantendo uma sincronização total sobre o mesmo em todos os participantes. Sincronização esta que é mantida pelo coordenador Terracotta. Coordenador este que aparentemente tambem mantém o vector em cache.
  Esta teoria é suportada pelo facto dos processos de todos os participantes e do coordenador Terracotta apresentarem bastante mais memória ocupada do que uma simulação do mesmo tamanho a correr directamente numa VM apenas. Embora não se possa concluir directamente que a memória ocupada se refere a esta replicação, as conclusões obtidas por vários utilizadores de Terracotta na partilha do mesmo tipo de objectos parece apontar neste sentido.
- Dada a natureza da simulação de promover interacções entre vários nós que podem "pertencer"a participantes diferentes, podemos concluir que irão existir muitas trocas de objectos entre os participantes. Estar trocas irão sempre ser feitas segundo um protocolo sobre TCP/IP, facto que não acontece na simulação original.

5.4 Resumo

A primeira conclusão a tirar do resultado destes testes de performance, é a de que efectivamente houve um forte sucesso na realização dos objectivos desta tese. Com excepção da simulação simulação distribuída, todos os testes revelaram que existe uma melhoria nos tempos de execução necessários para completar as experiências.
O motor *cycle-driven* mostrou-se efectivamente menos eficiente do que esperado no entanto demonstrou alguns ganhos de performance para simulações maiores. O motor

event-driven demonstrou o maior ganho de eficiência principalmente quando está a utilizar protocolos com grande carga computacional. A simulação distribuída revelou-se muito ineficiente em termos de memória ocupada e tempo de execução e serviu apenas para concluir a facilidade em partilhar objectos entre VMs usando o Terracotta e quais os problemas a evitar num futuro desenvolvimento de uma versão distribuída do projecto.



Conclusão

6.1 Resumo

Esta dissertação descreveu em detalhe o processo de investigação, desenho e realização de uma solução aumentada do Peersim, denominada PC-Peersim, que teve como objectivo efectuar uma implementação multithreaded do Peersim, passível de ser instalada num cluster.

No primeiro capítulo foi feita uma introdução aos conceitos de peer-to-peer e computação Grid que são centrais ao projecto. Estes temas foram enquandrados segundo um âmbito histórico de modo a ser possivel compreender a relevância que projectos nestas áreas foram tendo ao longo dos ultimos anos e de certa forma como o paradigma mais clássico da computação vai mudando. Foram também enunciados claramente quais os objectos desta tese e em que aspectos o PC-Peersim pode ser uma mais valia face à sua versão original.

No capítulo sobre trabalho relacionado foi feita uma enumeração de vários projectos nas áreas *Peer-to-Peer*, Computação Grid e simulação de *overlays* tendo estes sido classificados segundo as suas principais características diferenciadoras, sendo estas a centralização e tempo ou método de procura no caso dos projectos *Peer-to-Peer*, aplicação e regime de partilha no caso dos sistemas Grid e as limitações e performance dos sistemas de simulação. No caso dos sistemas *Peer-to-Peer* foram enunciados projectos de âmbito académico e também sistemas em uso para fins bastante específicos desde a partilha de conteúdos à publicação anónima. O mesmo se reflectiu na análise aos sistemas Grid que foram desde os sistemas institucionais de venda de recursos aos toolkits de construcção de redes cooperativas. Por fim, foi dado um especial

realçe ao Peersim dada a superior performance e extensibilidade que transpareceu na investigação feita, face aos outros projectos.

A arquitectura do PC-Peersim foi descrita com ênfase nas caracteristicas em que se destaca grandemente da versão original: paralelização e distribuição. Em vez de termos um *overlay* a ser simulado numa única thread numa única máquina, o PC-Peersim passa a conseguir correr a mesma simulação em paralelo num cluster. A necessidade de sincronização e coerência entre as várias *threads* de execução torna-se óbvia e a partir daqui é apresentado o processo de particionamento do *overlay*. Após o particionamento feito torna-se necessário definir um mecanismo de execução que garanta o sincronismo e coerência necessárias.
Ao nível da distribuição da execução num cluster, foi demonstrada a utilidade do Terracotta como mecanismo de partilha de memória e cache distribuída. Foram também explicados quais as limitações do Terracotta e os procedimentos necessários para que o PC-Peersim consiga partilhar efectivamente a carga sobre os vários participantes de forma eficiente.

Partindo da arquitectura foram também explicados alguns detalhes de implementação de maior relevância na sua realização. Foram detalhadas as várias abordagens feitas ao algoritmo de particionamento, e quais os maiores entraves encontrados nas variantes *multi-threaded*. Foi explicado em detalhe o funcionamento do algoritmo final *single-threaded* e as vantagens do uso de um particionamento aleatorio. Foram detalhados as representações em memória do overlay particionado como foram feitas as alterações ao motor de simulação para que este atingisse o objectivo proposto, tanto na sua versão *cycle-driven* como na sua versão *event-driven*, sendo que no caso da versão *event-driven* foi demonstrado o funcionamento da nova *queue* de eventos capaz de suportar o processamento das várias partições em separado.
Por fim, foi feita uma introdução aos detalhes do funcionamento do Terracotta, das suas capacidades e limitações e de como é efectuada a partilha de memória, definição de *roots*, e trincos, e sincronização entre todos os participantes.

6.2 Considerações globais de sucesso

Após a recolha de resultados, estamos em condições de dizer que os objectivos principais foram atingidos com grande sucesso. Em mais detalhe, podemos afirmar que conseguimos que é possível obter e executar simulações de uma forma mais eficiente e mais rápida usando o PC-Peersim. Após a execução dos exemplos, obtivemos em quase todo os exemplos um resultado satisfatório no que toca aos tempos de execução:

- O motor *cycle-driven* mostrou uma melhoria para cerca de 80% do tempo de execução original. Esta melhoria só é significativa para um grande número de nós. Sendo este motor o menos flexível dos dois, a relevância deste resultado poderá ser minorizada sendo todavia positivo. O particionamento por adjacência revelou-se pouco interessante, principalmente para um overlay pequeno e um número mais baixo de ciclos pois apresenta um tempo de processamento demasiado grande face ao tempo da respectiva simulação, fazendo com que no total, esta variante acabe por ser menos interessante que o particionamento aleatório.

- O motor *event-driven* já apresenta uma melhoria muito interessante na casa dos 50% da performance original. É aqui que surge a verdadeira vitória do projecto pois este é o motor mais usado para a criação de protocolos (sejam feitos à medida ou simulações de overlays conhecidos como Chord, Pastry, etc) pela comunidade de investigação em *peer-to-peer*. Os testes feitos com Chord e Pastry tiveram resultados muito bons tendo sido feitos adicionalmente testes em quad core sendo que existiu também um bom ganho de performance face aos testes em dual core. Como nestes casos, a simulação está a executar protocolos mais complexos e com uma carga computacional mais elevada, podemos afirmar que o desenvolvimento do PC-Peersim é sem dúvida uma mais valia.

- Na sua versão distribuída o PC-Peersim integrado no Terracotta apresentou resultados muito fracos. Dado o *overhead* de memória que o Terracotta adiciona à execução normal de um programa, aliado ao facto da necessidade de correr um coordenador e a fraca eficiência na partilha de memória de alguns tipos de classes, os testes acabaram por revelera que sem alterações mais fortes ao código, o

PC-Peersim ocupava tanta memória por participante para simular uma rede cerca de 10 vezes menor do que usaria correndo directamente apenas numa máquina. Como a natureza do Peersim envolve uma elevada troca de informação de nós, não é de todo eficiente que se faça uma partilha de memória de vários objectos e que estes andem a ser trocados por rede entre os participantes. Além da elevada carga de memória, esta execução revelou-se extremamente lenta. O objectivo de simular *overlays* maiores do que seriam possiveis apenas com uma máquina não fica cumprido, sendo que no entanto este exercicio se revelou útil para formular possibilidades de trabalho futuro para atingir este mesmo objectivo.

6.3 Trabalho Futuro

Após a realização desta tese e respectiva análise aos testes e objectivos, conclui-se que haverá trabalho muito interessante a fazer para atingir uma versão verdadeiramente distribuída e eficiente. O uso do Terracotta revela-se a abordagem errada, ficando até a ideia que seria possivel atingir o mesmo objectivo de uma forma mais simples, alterando a própria implementação. Usando um sistema de mensagens *lightweight* é possivel desenvolver uma versão do Peersim capaz de simular *overlays* de grande escala com uma boa eficiência de memória.
Após a análise dos resultados fica tambem a ideia que o motor *cycle-driven* pode ser reimplementado para uma maior eficência, embora seja discutível a sua utilidade a longo prazo. Por fim, considero que há imenso potencial para a implementação de grande parte dos protocolos *peer-to-peer* que foram analisados no capítulo de trabalho relacionado. As implementações encontradas de Chord e Pastry foram bastante úteis para análise de resultados, pelo que seria interessante construir uma maior base de código e conhecimento nesta área.

Bibliography

All, G. . Grid 4 all - main. http://www.grid4all.eu/.

Anderson, D. P. (2004). BOINC: a system for Public-Resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pp. 4–10. IEEE Computer Society.

Anderson, D. P., J. Cobb, E. Korpela, M. Lebofsky, & D. Werthimer (2002). SETI@home: an experiment in public-resource computing. *Commun. ACM 45*(11), 56–61.

Andrade, N., L. Costa, G. Germoglio, & W. Cirne (2005). Peer-to-peer grid computing with the OurGrid community.

Androutsellis-theotokis, S. & D. Spinellis (2004). A survey of peer-to-peer content distribution technologies.

Artigas, M. S., P. G. L?pez, J. P. Ahull?, & A. F. G. Skarmeta (2005). Cyclone: A novel design schema for hierarchical DHTs. In *Peer-to-Peer Computing, IEEE International Conference on*, Volume 0, Los Alamitos, CA, USA, pp. 49–56. IEEE Computer Society.

Baumgart, I., B. Heep, & S. Krause (2007). OverSim: a flexible overlay network simulation framework. In *2007 IEEE Global Internet Symposium*, Anchorage, AK, USA, pp. 79–84.

Dingledine, R., M. Freedman, & D. Molnar (2001). The free haven project: Distributed anonymous storage service. In *Designing Privacy Enhancing Technologies*, pp. 67–95.

Druschel, P. & I. 2002 (2002). *Peer-to-peer systems : First International Workshop,*

IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002 : revised papers. Berlin ;;New York: Springer.

Fedak, G. (2007, November). XtremWeb: building an experimental platform for global computing.

Folding@Home. Folding@home - main. http://folding.stanford.edu.

Foster, I. (2002). The grid: A new infrastructure for 21st century science. *Physics Today 55*, 42.

Heckmann, O. & A. Bock (2002, December). The eDonkey 2000 protocol. Technical report, Multimedia Communications Lab, Darmstadt University of Technology.

Information, J. L. & J. Liang (2004). Understanding KaZaA.

Kaashoek, M. & D. Karger (2003). Koorde: A simple Degree-Optimal distributed hash table. In *Peer-to-Peer Systems II*, pp. 98–107.

Kramer, D. & M. MacInnis (2004). Utilization of a local grid of mac OS X-Based computers using xgrid. In *High-Performance Distributed Computing, International Symposium on*, Volume 0, Los Alamitos, CA, USA, pp. 264–265. IEEE Computer Society.

Larson, S. M., C. D. Snow, M. Shirts, & V. S. Pande (2009, January). Folding@Home and Genome@Home: using distributed computing to tackle previously intractable problems in computational biology. http://adsabs.harvard.edu/abs/2009arXiv0901.0866L.

Lua, E. K., J. Crowcroft, M. Pias, R. Sharma, & S. Lim (2005). A survey and comparison of Peer-to-Peer overlay network schemes.

Malkhi, D., M. Naor, & D. Ratajczak (2002). Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, Monterey, California, pp. 183–192. ACM.

Maymounkov, P. & D. Mazières (2002). Kademlia: A Peer-to-Peer information system based on the XOR metric. In *Peer-to-Peer Systems*, pp. 53–65.

Milojicic, D. S., V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, & Z. Xu (2002). Peer-to-Peer computing.

Naicken, S., A. Basu, B. Livingston, & S. Rodhetbhai (2006). A survey of Peer-to-Peer network simulators.

Naicken, S., B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, & D. Chalmers (2007). The state of peer-to-peer simulators and simulations. *SIGCOMM Comput. Commun. Rev. 37*(2), 95–98.

Paul, S. R., P. Francis, M. H, R. Karp, S. Shenker, & W. V. Eske (2001). A scalable Content-Addressable network.

peersim. p2psim: a simulator for peer-to-peer (p2p) protocols. http://pdos.csail.mit.edu/p2psim/.

Pouwelse, J. & H. S. Pawea, Garbacki (2005). The bittorrent P2P File-Sharing system: Measurements and analysis. In *Peer-to-Peer Systems IV*, pp. 205–216.

Rowstron, A. & P. Druschel (2001). Pastry: Scalable, decentralized object location, and routing for Large-Scale Peer-to-Peer systems. In *Middleware 2001*.

Stoica, I., R. Morris, D. Karger, M. F. Kaashoek, & H. Balakrishnan (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 160.

Veiga, L., R. Rodrigues, & P. Ferreira (2007). GiGi: an ocean of gridlets on a "Grid-for-the-Masses". In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pp. 783–788. IEEE Computer Society.

Vouros, G. A., A. Papasalouros, K. Kotis, A. Valarakos, K. Tzonas, X. Vilajosana, R. Krishnaswamy, & N. Amara-Hachmi (2008). The Grid4All ontology for the retrieval of traded resources in a market-oriented grid. *International Journal of Web and Grid Services 4*(4), 418 – 439.

Yang, W. & N. Abu-ghazaleh (2005). GPS: a general peer-to-peer simulator and its use for modeling BitTorrent.

Apêndice

.1 *Configuração Terracotta*

```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd"
xmlns:tc="http://www.terracotta.org/config" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<servers>
<server host="localhost" name="sample"/>
</servers>
<system>
<configuration-model>development</configuration-model>
</system>
<clients>
<logs>terracotta/client-logs/peersim/</logs>
<dso>
<debugging>
<instrumentation-logging>
<class>true</class>
<roots>true</roots>
<locks>true</locks>

</instrumentation-logging>
<runtime-logging>
<lock-debug>true</lock-debug>
<wait-notify-debug>true</wait-notify-debug>
<new-object-debug>true</new-object-debug>
</runtime-logging>
</debugging>
</dso>
</clients>


<application>
<dso>
<instrumented-classes>
<include>
<class-expression>peersim.cdsim..*</class-expression>
</include>
```

```
<include>
<class-expression>peersim.core..*</class-expression>
</include>
<include>
<class-expression>peersim.config..*</class-expression>
</include>
<include>
<class-expression>peersim.part..*</class-expression>
</include>
<include>
<class-expression>peersim..*</class-expression>
</include>
<include>
<class-expression>example..*</class-expression>
</include>

</instrumented-classes>
<locks>
<named-lock>
<method-expression>void peersim.core.Network.addToDividedBorderNodes(..)</method-expression>
<lock-level>write</lock-level>
<lock-name>addToDividedBorderNodes</lock-name>
</named-lock>
<named-lock>
<method-expression>void peersim.core.Network.addToRegions(..)</method-expression>
<lock-level>write</lock-level>
<lock-name>addToRegions</lock-name>
</named-lock>
<named-lock>
<method-expression>void peersim.core.Network.putDividedBorderNodes(..)</method-expression>
<lock-level>write</lock-level>
<lock-name>putDividedBorderNodes</lock-name>
</named-lock>
<named-lock>
<method-expression>void peersim.core.Network.putRegion(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>putRegion</lock-name>
</named-lock>
<named-lock>
<method-expression>void peersim.core.Network.getNodes(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>getNodes</lock-name>
</named-lock>
<named-lock>
```

```
<method-expression>void peersim.core.Network.reset(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>reset</lock-name>
</named-lock>
<named-lock>
<method-expression>void peersim.core.GeneralNode.addColour(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>addColour</lock-name>
</named-lock>
<named-lock>
<method-expression>void peersim.core.Network.swap(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>swap</lock-name>
</named-lock>
<named-lock>
<method-expression>void peersim.core.Network.reset(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>reset</lock-name>
</named-lock>
<named-lock>
<method-expression>void peersim.core.Network.remove(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>remove</lock-name>
</named-lock>
<named-lock>
<method-expression>void peersim.core.Network.add(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>add</lock-name>
</named-lock>
<named-lock>
<method-expression>void example.*.*.nextCycle(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>nextCycle</lock-name>
</named-lock>
<named-lock>
<method-expression>* example.*.*.addNeighbor(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>addNeighbor</lock-name>
```

```
</named-lock>
<named-lock>
<method-expression>* peersim.vector.Setter.set(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>SetterSet</lock-name>
</named-lock>
<named-lock>
<method-expression>void peersim.core.OverlayGraph.setEdge(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>setEdge</lock-name>
</named-lock>
<named-lock>
<method-expression>void peersim.Simulator.registerMachine(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>registerMachine</lock-name>
</named-lock>
<named-lock>
<method-expression>void peersim.Machine.setColor(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>machineSetColor</lock-name>
</named-lock>
<named-lock>
<method-expression>void peersim.Machine.setNodeLoad(..)
</method-expression>
<lock-level>write</lock-level>
<lock-name>machineSetNodeLoad</lock-name>
</named-lock>
<named-lock>
<method-expression>* peersim.Simulator.setParticipantColor(..)</method-expression>
<lock-level>write</lock-level>
<lock-name>setParticipantColor</lock-name>
</named-lock>
<named-lock>
<method-expression>* example.aggregation.AverageFunction.activateNodeInteraction(..)</method-expression>
<lock-level>write</lock-level>
<lock-name>activateNodeInteraction</lock-name>
</named-lock>
<autolock>
<method-expression>* peersim.Simulator.main(..)</method-expression>
<lock-level>write</lock-level>
</autolock>
<autolock>
```

```
<method-expression>* peersim.Simulator.setParticipantColor(..)</method-expression>
<lock-level>write</lock-level>
</autolock>
<autolock>
<method-expression>* peersim.cdsim.CDSimulator.nextExperiment(..)</method-expression>
<lock-level>write</lock-level>
</autolock>
</locks>
<roots>
<root>
<field-name>peersim.Simulator.slaveMachine</field-name>
</root>
<root>
<field-name>peersim.Simulator.distributedMode</field-name>
</root>
<root>
<field-name>peersim.core.Network.regions</field-name>
</root>
<root>
<field-name>peersim.core.Network.dividedBorderNodes</field-name>
</root>
<root>
<field-name>peersim.core.Network.loadCursor</field-name>
</root>
<root>
<field-name>peersim.Simulator.barrier</field-name>
</root>
<root>
<field-name>peersim.core.Network.nodes</field-name>
</root>
<root>
<field-name>peersim.core.Network.len</field-name>
</root>
<root>
<field-name>peersim.Simulator.participantRegistry</field-name>
</root>
<root>
<field-name>peersim.Simulator.afterInit</field-name>
</root>
<root>
<field-name>peersim.Simulator.afterReset</field-name>
</root>
<root>
<field-name>peersim.Simulator.coordinatorID</field-name>
</root>
<root>
```

```
<field-name>peersim.Simulator.preSimulationLock</field-name>
</root>
<root>
<field-name>peersim.Simulator.participantColors</field-name>
</root>
<root>
<field-name>peersim.cdsim.CDSimulator.preResetLock</field-name>
</root>
</roots>
<additional-boot-jar-classes>
<include>java.util.RandomAccessSubList</include>
</additional-boot-jar-classes>
</dso>
</application>
</tc:tc-config>
```