INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Topologias de *overlays peer-to-peer* para descoberta de recursos

FILIPE ROCHA PAREDES

Dissertação para obtenção do Grau de Mestre em

ENGENHARIA DE REDES DE COMUNICAÇÃO

Júri

Presidente:	Prof. Dr. Luís Eduardo Teixeira Rodrigues
Orientador:	Prof. Dr. Luís Manuel Antunes Veiga
Co-orientador:	Prof. Dr. Rodrigo Seromenho Miragaia Rodrigues
Vogais:	Prof. Dr. Hugo Alexandre Tavares Miranda

Setembro de 2008

Esta dissertação é dedicada à minha família.

Agradecimentos

No decorrer do mestrado recebi o apoio de diversas pessoas. Gostaria de agradecer à minha família e amigos chegados, e em particular aos meus pais Mário Pereira Paredes, à minha mãe Bárbara da Conceição Monteiro Rocha, ao meu irmão Nuno Miguel Rocha Paredes e a Carla Marisa Teixeira Pereira.

Gostaria de agradecer ao meu orientador, o professor Dr. Luís Manuel Antunes Veíga, pela preocupação e motivação que me proporcionou.

Gostaria também de estender os meus agradecimentos aos meus colegas de curso, Gonçalo Teixeira, Davide Figo, Samuel Silva, Sérgio Santos, João Campos Vicente, André Sabino, Tiago Casemiro, Bruno Capelas, André Morais, Carlos André, José Coelho, Tiago Sousa, André Martins, Luís Pedro, Frederico Silva, Luís Santos, João Matos, Manuel Cabral, entre muitos outros, e aos meus amigos Dália Portela, Catarina Silva, Vitor Costa e Joel Rodrigues.

O apoio destas pessoas foi fundamental no decorrer deste trabalho.

Sumário

Existem actualmente vários projectos que tentam melhorar o desempenho de aplicações através da utilização de ciclos excedentários existentes noutras máquinas ligadas à Internet, por vezes em troca, da partilha futura dos próprios ciclos ociosos. No entanto, nenhum destes projectos permite, em larga escala, a execução mais rápida de aplicações *desktop*, não modificadas, por parte de utilizadores comuns, sem a existência de uma infra-estrutura de partilha de recursos que necessite de uma aplicação cliente específica instalada no computador hospedeiro.

Para resolver este problema, esta dissertação propõe mecanismos de descoberta de recursos (e.g., CPU) para explorar diferentes topologias da rede *peer-to-peer* que maximizem métricas de desempenho do sistema. A solução faz parte de um projecto existente, o GINGER, que intenta a síntese de três abordagens: infra-estruturas *Grid* institucionais, aplicações populares de partilha de ciclos e aplicações de partilha de ficheiro P2P descentralizadas.

A solução consiste no desenvolvimento de uma infra-estrutura de *middleware* P2P baseada no conceito de uma *Gridlet*, uma unidade básica de trabalho (divisora em pequenas tarefas), consciente da semântica dos seus dados. São analisados vários critérios, como a largura de banda ou disponibilidade de recursos, para a escolha de vizinhos na rede *peer-to-peer* com vista ao encaminhamento de *Gridlets*.

Palavras-chave

Gridlet, Disponibilidade, O*verlay*, Recursos, Partilha

Abstract

There are currently a variety of projects that try to improve the performance of applications by using spare cycles from other computers connected over the Internet and, sometimes in return of their own spare cycles in the future. Nonetheless, none of those projects allows, in large scale, home users to run unmodified desktop applications faster without an infrastructure for resource sharing that implies the use of a specific client-application in the computer host.

To address such problem, this dissertation proposes mechanisms for resource discovery (e.g., CPU) to exploit different peer-to-peer network topologies that maximize the system performance metrics. This solution is part of an existent project, GINGER, that aims for the synthesis of three approaches: institutional grid infrastructures, popular cycle sharing applications and massively used decentralized P2P file-sharing applications.

The solution seek the development of a P2P middleware infrastructure based on the concept of a Gridlet, a semantics-aware unit of workload division and computation off-load. Several criteria, like bandwidth or resources availability are subject to analysis for the choice of neighbours in the peer-to-peer network for the routing of Gridlets.

Keywords

Gridlet, Availability, O*verlay*, Resources, Sharing

Índice

Índice de Figuras

Índice de Tabelas

Lista de Acrónimos

P2P – *Peer-to-Peer*

CPU – *Central Processing Unit*

RTT – *Round Trip Time*

GINGER – *Grid Infrastructure for Non-Grid EnviRonments*

DHT - *Distributed hash table*

I/O – *Input / Output*

IDs - *Identifiers*

1. Introdução

1.1 Enquadramento

Com o crescente acesso à Internet e o aumento das capacidades (de processamento, memória, armazenamento, etc.) dos computadores pessoais, não se deve negligenciar o poder computacional que pode ser obtido através da utilização de recursos ociosos disponíveis nestas máquinas. Com vista ao aproveitamento destes recursos, foram desenvolvidas infra-estruturas de *Grids* Institucionais e aplicações *Peer-to-Peer* (P2P), que têm permitido a utilização de tais recursos, nomeadamente, para a execução de aplicações paralelas (*Grids*) ou partilha de ficheiros entre várias máquinas ligadas à Internet (P2P).

Em diversas comunidades da Internet, tem-se assistido a uma grande expansão e popularização de aplicações P2P de partilha de recursos, quer de ciclos de processamento (SETI@*home* [1]) quer de ficheiros (por exemplo, a rede de distribuição de conteúdos P2P *BitTorrent* representava 35% do tráfego da Internet em 2005 [2]). Trabalhos relevantes na área de P2P, designadamente na construção de *hash-tables* distribuídas, incluem projectos como o *Chord* [3] e o *Pastry* [4].

A computação em *Grid* tem-se desenvolvido como um modelo computacional de nova geração, tanto no mundo científico como no comercial. A disseminação desta tecnologia incentivou o desenvolvimento de várias ferramentas com o objectivo de facilitar o acesso aos recursos em *Grids*. Estas aplicações correm numa infra-estrutura de *Grid* que interliga vários domínios institucionais de computadores, que possibilita a negociação e alocação de recursos de outros domínios dispersos geograficamente, possibilitando a execução de aplicações paralelas.

O poder computacional mundial e o espaço de armazenamento já não se encontram maioritariamente concentrado em complexos de super-computadores. Em vez disso, distribuem-se por centenas de milhões de computadores pessoais e consolas de jogos, pertencentes ao público em geral. A utilização de ciclos de processamento distribuído emergiu em aplicações como SETI@*home* que segue um modelo de cliente-servidor, onde um servidor central distribuiu as tarefas para os clientes, que voluntariamente oferecem os seus ciclos. Após a execução das tarefas durante os períodos ociosos da máquina, os resultados são enviados para o servidor central.

1.2 Motivação

Ao longo dos últimos anos foram apresentadas algumas propostas que tentam estabelecer uma ligação entre as infra-estruturas de *Grids* Institucionais (p.e. *Globus* [5]), aplicações populares de partilha de recursos (p.e. SETI@*home* [1]) e aplicações de partilha de ficheiros P2P descentralizadas. No entanto, nenhuma dessas infra-estruturas permite que utilizadores comuns executem, em grande escala, aplicações populares de forma mais rápida, através da utilização de

ciclos de processamento excedentários de outros utilizadores, em troca dos seus ciclos para a execução de aplicações.

1.3 Contribuição Esperada

Mediante a crescente evolução de tecnologias como *Grids* Institucionais e tecnologias P2P, este trabalho vem tentar colmatar as lacunas deixadas pelas infra-estruturas que juntaram a partilha de recursos às tecnologias anteriores. A solução proposta projecta um sistema de descoberta de recursos num *overlay* P2P com o objectivo de explorar diferentes topologias da rede que maximizem métricas de desempenho do sistema. Esse sistema consiste numa infra-estrutura de *middleware* P2P baseada no conceito de uma *Gridlet*, uma unidade básica de trabalho (divisora em pequenas tarefas), consciente da semântica dos seus dados. As *Gridlets* são submetidas para a rede e processadas a partir de recursos excedentários de outras máquinas da rede para o processamento de tarefas de aplicações populares, sem que seja necessárias modificações nessas aplicações.

1.4 Organização dos Capítulos

Esta dissertação está apresentada em seis capítulos. No primeiro capítulo é dada uma introdução ao tema do trabalho e uma breve descrição da solução apresentada. O segundo capítulo descreve o trabalho relacionado das tecnologias chave para a o desenvolvimento da solução. Num terceiro capítulo são abordados os objectivos de desenho, os requisitos e a arquitectura de alto nível do sistema projectado. Um quarto capítulo refere a implementação de baixo nível da arquitectura do sistema realizado, bem como a realização dos mecanismos de descoberta de recursos e a especificação da tecnologia utilizada. O quinto capítulo está reservado à apresentação e análise de testes para avaliar o desempenho e testar as funcionalidades implementadas. Para finalizar, o último capítulo reflecte as conclusões obtidas neste trabalho, deixando algumas sugestões para futuro desenvolvimento sobre o sistema preconizado.

2. Estado da Arte

Nas secções seguintes deste capítulo são descritas as três tecnologias (sistemas *peer-to-peer*, *grids* institucionais e *cycle sharing* ou *desktop grids)* que se pretende agregar, com referência a plataformas existentes para cada uma delas, sendo realizada uma análise e comparação dos sistemas mencionados. Posteriormente, é efectuada uma breve descrição da arquitectura do projecto a desenvolver, salientando os pontos-chave do mesmo. Para finalizar, a última secção reflecte as conclusões deste trabalho.

2.1 SISTEMAS PEER-TO-PEER

A computação P2P [9] tem promovido uma grande modificação nos padrões de utilização da Internet nos últimos anos. A sua grande vantagem, em relação à computação cliente/servidor, é possibilitar a colaboração directa entre os utilizadores, sem depender de servidores centralizados administrados por terceiros, que são susceptíveis a falhas como ponto único de estrangulamento ou ponto único de falha. As redes P2P permitem a partilha de recursos computacionais pelos utilizadores da Internet, mesmo que as máquinas estejam escondidas atrás de *firewalls* e NATs.

A inexistência de um servidor centralizado significa que é possível cooperar para a formação de uma rede P2P, onde todas as máquinas fazem o papel de cliente e o papel de servidor, sem qualquer investimento adicional em hardware de alto desempenho para coordená-la. Outra vantagem é a possibilidade de agregar e utilizar a capacidade de processamento e armazenamento que fica subutilizada em máquinas ociosas. Além disso, a natureza descentralizada e distribuída dos sistemas P2P torna-os inerentemente robustos a certos tipos de falhas muito comuns em sistemas centralizados. Finalmente, o modelo P2P apresenta o benefício da escalabilidade, para tratar de crescimentos incontroláveis no número de utilizadores e equipamentos ligados, da capacidade de rede, das aplicações e da capacidade de processamento.

Os sistemas P2P podem, assim, ser definidos como um conjunto de nós interligados entre si, com a capacidade de se auto-organizarem em topologias de rede, capazes de se adequarem às constantes entradas e saídas de nós e de se adaptarem a eventuais falhas, mantendo o funcionamento aceitável da rede, em termos de conectividade e desempenho, sem necessitarem do suporte ou intermediação de uma entidade centralizada. Este tipo de sistemas destina-se principalmente à partilha de recursos entre os participantes, desde partilha de conteúdos, ciclos da CPU, armazenamento e largura de banda.

No entanto, existem sistemas considerados P2P que não são totalmente descentralizados, englobando graus de centralização, desde sistemas totalmente descentralizados, como *Gnutella* [6] ou *FreeHaven* [7], até sistemas parcialmente centralizados, como o *Kazaa* [8] ou *Morpheus*[9].

Em seguida são apresentadas aplicações, características e modelos de arquitecturas de sistemas P2P onde se podem verificar as propriedades deste tipo de sistemas para posterior análise e comparação (secção 5), orientadas ao enquadramento e desenvolvimento da solução proposta.

2.1.1 Aplicações

As arquitecturas P2P têm vindo a ser implementadas em várias categorias de aplicações: comunicação (troca de mensagens), distribuição de conteúdos (partilha de ficheiros) e computação distribuída são alguns exemplos.

Troca de mensagens: Esta categoria engloba os sistemas que fornecem uma infra-estrutura para o suporte de aplicações, normalmente de tempo-real, de comunicação e colaboração entre computadores. O ICQ (www.icq.com) lançou o modelo que é agora utilizado por todos os sistemas populares de troca de mensagens. O sistema ICQ é baseado num directório central, num protocolo proprietário e um serviço e aplicação cliente gratuitos. Sistemas actuais, como o Yahoo! Messenger, AOL Instant Messenger, e MSN Messenger suportam um modo P2P para a troca directa de mensagens entre clientes.

Ao contrário do correio electrónico, em que uma mensagem é armazenada numa conta de correio e entregue ao utilizador quando este verifica a sua conta, os sistemas de mensagens instantâneas fornecem a entrega imediata das mensagens ao utilizador. Se o utilizador não estiver disponível, a mensagem pode ser armazenada, até que o mesmo se torne activo, ou pode ser simplesmente descartada. Para evitar esta incerteza na entrega, os sistemas de mensagens instantâneas fornecem uma lista de contactos com um mecanismo capaz de identificar um utilizador e determinar o seu estado, por exemplo, activo, inactivo ou ocupado. Algumas das aplicações de troca de mensagens são apresentadas a seguir:

Distribuição de conteúdos: Engloba sistemas e infra-estruturas desenhadas para a partilha de todo o tipo de dados entre os utilizadores, desde partilha de ficheiros até sistemas mais sofisticados de armazenamento distribuído para publicação, organização, indexação, pesquisa, actualização e recuperação de dados de uma forma segura e eficiente. À medida que os sistemas ficam mais sofisticados, asseguram requisitos não funcionais como anonimato, justiça, aumento da escalabilidade e desempenho.

Os sistemas de armazenamento distribuído baseados na tecnologia P2P utilizam as vantagens da infra-estrutura existente para oferecer as seguintes características: áreas de armazenamento e publicação de conteúdos - sistemas como o Freenet [10] fornecem ao utilizador uma área potencialmente ilimitada para o armazenamento de dados; alta disponibilidade dos conteúdos armazenados - para garantir alta disponibilidade dos dados armazenados, alguns sistemas adoptam uma política de replicação múltipla dos conteúdos, isto é, o mesmo ficheiro pode estar armazenado em mais vários nós da rede, ex: Chord; anonimato - alguns sistemas P2P, como o Publius [11], garantem matematicamente que o documento publicado será mantido no anonimato dos seus autores e publicitários; localização e encaminhamento - a maioria dos sistemas P2P fornecem mecanismos eficientes de localização e recuperação dos conteúdos armazenados na rede; gestão da reputação - em P2P não existe um gestor de reputação centralizado, pelo que cabe aos nós guardar essa informação.

As principais questões técnicas em sistemas de partilha de ficheiros são o consumo de largura de banda da rede, a segurança e a sua capacidade de pesquisa [12]. Algumas das aplicações de partilha de ficheiros são apresentadas a seguir:

Computação distribuída: Inclui sistemas cujo objectivo é aproveitar ciclos da CPU disponíveis em outros computadores. A metodologia utilizada consiste na segmentação da tarefa a realizar, em pequenas unidades de trabalho, sendo distribuídas pelos computadores da rede, que vão executar as unidades de trabalho e devolver os resultados. Vários sistemas que abordam este método utilizam um coordenador central que divide as tarefas, distribui-as pelos nós e recolhe os resultados. Exemplos desses sistemas incluem projectos como o SETI@home, Genome@home e Folding@home. Alguns projectos como o MOSIX [13] e Condor [14] também têm feito uso da computação distribuída, delegando as tarefas computacionais entre as máquinas ociosas.

2.1.2 Características dos sistemas P2P

Escalabilidade: Um dos benefícios imediatos da descentralização das redes P2P é a melhor escalabilidade dos sistemas. A escalabilidade é limitada por factores relacionados com quantidade de operações centralizadas, como coordenação e sincronização, manutenção de estados, o paralelismo inerente das aplicações ou o modelo de programação utilizado na construção do sistema. Em geral, sistemas P2P tendem a ter maior escalabilidade do que os sistemas que utilizam o modelo cliente-servidor, onde os servidores são os únicos responsáveis por toda a carga do sistema. Num sistema P2P, quando o número de clientes na rede aumenta, cresce também o número de servidores, uma vez que todos podem actuar como clientes e servidores, aumentando na mesma proporção a quantidade de recursos disponíveis para partilha. Desta forma, o sistema aumenta de tamanho sem perder escalabilidade.

O Napster trata o problema da escalabilidade permitindo que os nós realizem o download de ficheiros de músicas directamente dos nós que possuem o ficheiro solicitado. Deste modo, o Napster chegou a suportar até 6 milhões de utilizadores no seu auge.

A escalabilidade também depende do raio de comunicação (grau de conectividade) para computação entre os nós de um sistema P2P. As aplicações de criptografia e de procuras de grandes números primos têm melhor desempenho quando o raio de comunicação é mais próximo de zero.

Os sistemas P2P mais antigos como Gnutella e Freenet são essencialmente ad-hoc. Nestes sistemas, um nó envia os seus pedidos para outros nós e estes reencaminham os pedidos adiante para os nós vizinhos. Esse comportamento faz com que o tempo de resposta do pedido tenha características não determinísticas. Algoritmos mais recentes, como CAN [15], Chord [3] e Pastry [4], implementam um mapeamento consistente entre a chave do objecto e o nó que contém esse objecto. Consequentemente, um objecto pode ser recuperado logo que os nós responsáveis possam ser alcançados. Cada nó mantém apenas informações acerca de um pequeno número de outros nós no sistema. Esta característica limita a quantidade de estados

do sistema que precisam de ser mantidos, melhorando desta forma a escalabilidade e proporcionando o balanceamento de carga.

Encaminhamento: Os algoritmos de procura e de encaminhamento tentam geralmente optimizar o encaminhamento de uma mensagem de um nó para outro. Os três modelos mais comuns são: centralizado, por inundação (flooding) e tabela de hash distribuída (DHT). O modelo de inundação pode ser também classificado como modelo "descentralizado não estruturado", enquanto o modelo DHT pode ser classificado como modelo "descentralizado estruturado". Isto deve-se aos mecanismos de entrada de um nó na rede e da procura de informações serem estruturados ou não.

Modelo Centralizado: Popularizado pelo Napster, este modelo caracteriza-se pela ligação dos nós a um directório central onde publicam informações sobre os conteúdos que disponibilizam para partilha. Como ilustra a Figura 1, ao receber um pedido, o índice (directório) central escolhe o nó no directório que for mais adequado. Esse nó pode ser o mais rápido e disponível, dependendo das necessidades do utilizador. A troca de ficheiros é realizada directamente entre os dois nós. Este modelo requer uma infra-estrutura centralizada (o servidor de directórios), que armazena as informações sobre todos os participantes da comunidade. Tal aspecto pode gerar limites de escalabilidade ao modelo, uma vez que requer servidores maiores quando o número de pedidos aumenta e mais espaço para armazenamento à medida que a quantidade de utilizadores cresce. Contudo, a experiência do Napster mostrou que, excepto por questões legais, este modelo era bastante robusto e eficiente.

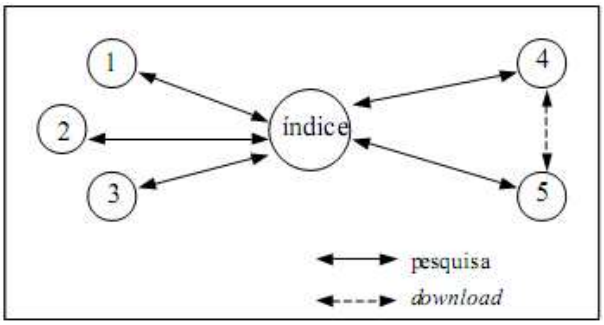

Figura 1: Modelo centralizado

Modelo por Inundação: O modelo por inundação de pedidos consiste no envio de cada pedido para todos os nós directamente ligados ao nó que efectuou o pedido, os quais enviam para os nós directamente ligados a eles, e assim sucessivamente até que o pedido seja respondido ou que ocorra o número máximo de encaminhamentos (tipicamente 5 a 9). Este modelo (Figura 2) é utilizado pelo Gnutella e exige alta capacidade de comunicação pelos nós para proporcionar um desempenho razoável, apresentando problemas de escalabilidade quando o objectivo é alcançar todos os nós de uma rede. Contudo, o modelo é eficiente em comunidades limitadas e em redes corporativas.

Manipulando o número de conexões de cada nó e configurando apropriadamente o valor do parâmetro TTL (time to live) das mensagens de pedido, o modelo por inundação pode ser utilizado por centenas de milhares de nós. Além disso, algumas empresas têm desenvolvido softwares para clientes “super-nós”, os quais concentram vários pedidos, alcançando um bom desempenho nas ligações entre os nós de menor capacidades de transmissão a custo de alto consumo da CPU. O armazenamento de pesquisas recentes (caching) também é usado para melhorar a escalabilidade.

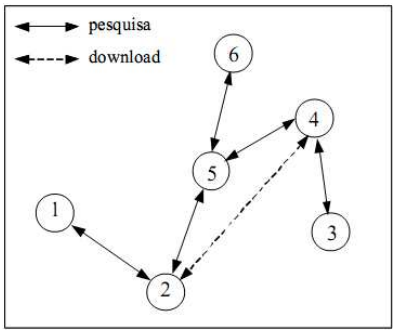

Figura 2: Modelo por inundação

Modelo DHT: O modelo de tabelas de hash distribuídas (DHT) é o mais recente. Neste modelo, é atribuído um ID aleatório a cada nó da rede, que conhece um determinado número de nós, como pode ser visualizado na Figura 3. Quando um ficheiro é publicado (partilhado), é-lhe associado um ID baseado numa função de hash do seu conteúdo e do seu nome. Cada nó encaminha o ficheiro ao nó cujo ID é mais próximo do ID do ficheiro. Este processo é repetido até que o ID do nó actual seja o mais próximo do ID do ficheiro. Quando um nó solicita um ficheiro à rede, o pedido será encaminhado até ao nó com o ID mais semelhante ao ID do ficheiro, até que uma cópia do ficheiro seja encontrada [3].

Apesar do modelo DHT ser eficiente para comunidades grandes e globais, apresenta um problema relacionado com o ID do ficheiro que precisa de ser conhecido antes da realização de um pedido. Assim, é mais difícil implementar uma pesquisa neste modelo que no modelo de inundação. Além disso, o particionamento da rede pode originar a formação de “ilhas”, onde a comunidade se divide em sub-comunidades independentes quando deixam de possuir ligações entre si (saídas de nós ou quebra de ligações).

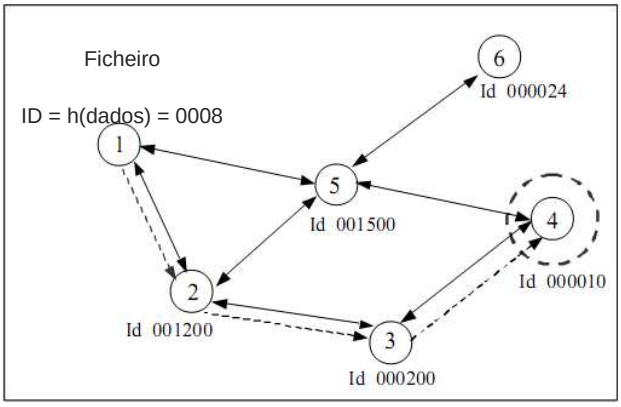

Figura 3: Modelo DHT

2.1.3 Modelos de arquitecturas P2P

O funcionamento de um sistema de distribuição de conteúdos em P2P assenta numa rede de nós ligados entre si, referida como uma rede *overlay* (Figura 4). Uma rede *overlay* é uma rede "virtual" criada sobre uma rede física existente, por exemplo: a Internet, com infra-estrutura IP. Os nós de uma rede *overlay* podem ser considerados como ligações lógicas ou virtuais, que correspondem a um caminho através de uma ou mais ligações físicas da rede subjacente. A rede *overlay* cria uma arquitectura com um alto nível de abstracção, de modo a poder solucionar vários problemas que, em geral, são difíceis de serem tratados ao nível dos *routers* da rede subjacente [16]. A própria Internet pratica o paradigma *overlay* quando utiliza o protocolo IP como solução de interligação sobre tecnologias de diversas redes como ATM, *Frame Relay*, PSTN, LANs, etc.

O funcionamento do sistema depende da topologia, estrutura, grau de centralização e mecanismos de localização e encaminhamento da rede, na medida em que estes afectam a tolerância a faltas, auto-sustentabilidade, adaptação à faltas, desempenho, escalabilidade e segurança do sistema.

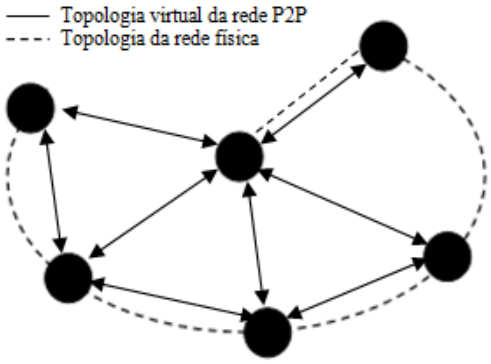

Figura 4: Rede *overlay*

As arquitecturas das redes *overlay* podem ser distinguidas no que diz respeito ao seu grau de centralização, 2.1.1, 2.1.2 e 2.1.3, e a sua estrutura, 2.1.4 e 2.1.5. A tabela 1 classifica alguns exemplos de sistemas.

	Centralização		
	Híbrida	Parcialmente	Nenhuma
Não estruturada	*Napster* *Publius*	*KaZaA* *Gnutella* *Morpheus* *Edutella*	*Gnutella* *FreeHaven*
Estruturada			*Chord,* *Pastry,* *CAN,* *Tapestry*

Tabela 1: Exemplos de sistemas P2P. No que diz respeito aos seus graus de centralização e estruturação

2.1.3.1 Totalmente descentralizadas

Todos os nós da rede efectuam as mesmas funções e tarefas, não existindo diferenciação entre os mesmos. Não existe coordenação centralizada nas actividades dos nós.

Freenet: O *Freenet* [17] (http://freenet.sourceforge.net/) é um sistema totalmente descentralizado e estruturado de distribuição de conteúdos. Enquanto o *Gnutella* é apenas um sistema aplicativo, o *Freenet* tem objectivos sociopolíticos, tais quais: permitir a distribuição do material anonimamente; permitir a consulta do material de forma anónima; garantir que seja praticamente impossível a retirada completa do material na rede; operar sem controlo centralizado.

Arquitectura: O foco principal é o armazenamento de ficheiros. A rede *Freenet* para além do encaminhamento de mensagens, armazena também vários ficheiros em máquinas não proprietárias dos ficheiros, construindo um sistema de armazenamento distribuído e tolerante a falhas. Para além disso, a codificação dos dados armazenados noutras máquinas permite garantir a confidencialidade.

No *Freenet*, os ficheiros são identificados por chaves binárias únicas, podendo suportar três tipos de chaves, sendo a mais simples baseada na aplicação de uma função de *hash* a um pequeno texto descritivo que vem com cada ficheiro. Cada nó mantém os seus dados locais, que disponibiliza para leitura e escrita, e uma tabela de encaminhamento que guarda os endereços de outros nós e as chaves que eles devem possuir.

O *Freenet* utiliza as seguintes mensagens, que incluem o identificador do nó (para detecção do *loop*), um valor de *hops-to-live* (semelhante ao TTL do *Gnutella*) e os identificadores dos nós de origem e destino: *data insert* – um nó insere novos dados na rede, incluindo a chave e o próprio ficheiro; *data request* – pedido de um certo ficheiro, contendo a chave do ficheiro; *data reply* – mensagem de resposta quando o ficheiro é encontrado, contendo o

ficheiro pedido; *data failed* – falha na localização do ficheiro, contendo o local (nó) da falha e a razão.

Inserção de novos ficheiros : Os nós estão ligados via TCP/IP. Quando um novo nó entra na rede tem que conhecer o endereço de outro nó pertencente à rede e começar depois a enviar mensagens de *data insert*. Para inserir novos ficheiros na rede o nó calcula a chave binária (função de *hash* do conteúdo) para o ficheiro e envia uma mensagem de *data insert* para ele mesmo.

Qualquer nó que receba uma mensagem de *data insert* verifica se a chave já está a ser usada. Se a chave não for encontrada, o nó procura pela chave mais próxima na sua tabela de encaminhamento e envia a mensagem de *insert* para o nó correspondente (através deste mecanismo os novos ficheiros inseridos serão armazenados em nós com ficheiros de chaves semelhantes). Este mecanismo contínua até ser alcançado o limite de *hops-to-live*. Desta forma, o novo ficheiro será guardado em vários nós e, ao mesmo tempo, os nós participantes irão actualizar as suas tabelas com a nova informação (este é também o mecanismo usado pelos novos nós para anunciarem a sua presença ao resto da rede).

Se o limite de *hops-to-live* é atingido sem que haja colisões de chaves, será propagado um resultado de "*all-clear*" até ao nó que enviou a mensagem original, informando que a inserção foi bem sucedida. Se a chave já estiver a ser utilizada, o nó retorna o ficheiro pré-existente como se de um pedido se tratasse. Desta forma, tentativas maliciosas de substituir ficheiros inserindo lixo irão resultar numa maior distribuição dos ficheiros já existentes.

Procura : Cada nó do *Freenet* envia a mensagem de *data request* para nós próximos ao que eles acham que possui a informação, de acordo com as suas tabelas de encaminhamento. Se um nó recebe um *request* para um ficheiro que possui, a procura cessa e o ficheiro é enviado de volta ao solicitador. Se o nó não possui o ficheiro, reencaminha a mensagem de *request* para o vizinho com maior probabilidade de ter o ficheiro, consultando, na sua tabela de encaminhamento, a chave do ficheiro com valor mais próximo à chave solicitada. Para evitar grandes propagações das mensagens de *request*, estas são descartadas depois de passarem por um certo número (*hops-to-live*) de nós. Os nós guardam também o ID e outras informações das mensagens de *request* para tratamento de mensagens de *data reply* e *data failed*.

Se um nó recebe uma mensagem de *data failed* de um nó subjacente, este escolhe o próximo nó mais provável da sua lista de encaminhamento e reencaminha o pedido para ele. Se todos os nós da tabela de encaminhamento tiverem sido explorados e enviado uma mensagem de *failed*, o nó envia de volta, ao nó que lhe enviou a mensagem de pedido original, a mensagem de *data failed*.

Se o ficheiro pedido é encontrado num certo nó, este envia uma mensagem de *reply* de volta para cada nó que encaminhou o pedido, contendo o ficheiro solicitado. Os nós intermediários podem armazenar o ficheiro localmente, passando a ser fonte da informação

para consultas futuras. Pedidos com chaves semelhantes serão reencaminhados para o nó que anteriormente forneceu os dados.

Transferência de ficheiros : Um dos objectivos do *Freenet* é manter anónimos os nós que possuem um ficheiro. Ao contrário do *Gnutella*, a recuperação de um ficheiro é realizada através de vários nós, o que dificulta a identificação do nó que forneceu o ficheiro, tornando anónimas as fontes que partilham informações.

Como resultado dos algoritmos de encaminhamento e procura, uma rede *Freenet* destaca as seguintes propriedades: os nós tendem a especializar-se na procura de chaves semelhantes ao longo do tempo; os nós armazenam chaves semelhantes ao longo do tempo, devido ao *caching* de ficheiros como resultado de pedidos bem sucedidos; semelhança de chaves não reflecte a semelhança de ficheiros; o encaminhamento não reflecte a topologia da rede subjacente.

2.1.3.2 Parcialmente centralizadas

Semelhante à arquitectura totalmente descentralizada, com a excepção de que existem nós com funcionalidade acrescida. Estes super-nós são nós normais aos quais lhe foram atribuídas responsabilidades extra: indexação dos ficheiros partilhados pelos nós ligados ao super-nó e funciona como *proxy* aos pedidos de procura dos outros nós. A escolha destes é feita de forma automática e dinâmica para nós que tenham largura de banda suficiente e poder de processamento. Em caso de falha de um super-nó a rede toma medidas para proceder á sua substituição, pelo que não representam pontos de estrangulamento único. O *Kazaa* é um sistema típico da implementação desta arquitectura.

KaZaA: O *KaZaA* (http://www.kazaa.com – versão actual 3.2.5) é o sistema de partilha de ficheiros que utiliza o conceito de super-nós para melhorar o desempenho da rede. Os super-nós mantêm uma lista contendo os ficheiros disponibilizados por outros utilizadores e o local onde eles estão armazenados. Quando é efectuada uma procura, a aplicação *KaZaA* procura primeiro no super-nó mais próximo do utilizador que iniciou a consulta, que retorna um conjunto de respostas para o utilizador e encaminha a consulta para outros super-nós. Uma vez localizado o utilizador que possui o ficheiro, é estabelecida uma ligação directa entre os nós para que seja efectuado o download.

Os super-nós são nós simples que foram promovidos com responsabilidades extra. Normalmente, a escolha dos nós a promover é feita baseando-se na sua capacidade de processamento e largura de banda, sendo escolhidos os nós que suportam a transmissão de tráfego de sinalização com débitos *upstream* e *downstream* de 161 *kbps* e 191 *kbps*, respectivamente [8].

2.1.3.3 Descentralizadas híbridas

Os clientes entram na rede ao estabelecerem ligação com um servidor centralizado que guarda, numa tabela dos clientes registados, as informações da ligação (endereço IP,

largura de banda da ligação, etc.). Cada cliente indica ao servidor os ficheiros que contém e disponibiliza para partilha na rede. O servidor guarda esses registos juntamente com descrições dos ficheiros (nome do ficheiros, data de criação, etc.) numa tabela de listagem dos ficheiros em cada utilizador.

Napster: O *Napster* [18] (http://free.napster.com) é o exemplo de um sistema P2P dedicada à procura de ficheiros MP3, que depende de servidores centrais. É utilizado um servidor central para armazenar uma lista com as músicas disponibilizadas pelos utilizadores e as suas respectivas localizações. O servidor não consiste apenas numa máquina, mas sim em várias, de modo a gerir a carga de trabalho do elevado número de clientes. Desta forma, o redimensionamento do servidor consiste apenas na adição de novas máquinas garantindo ao mesmo tempo redundância em caso de falhas pontuais de um número limitado de máquinas constituintes deste "servidor virtual".

O programa cliente *Napster*, instalado no computador do utilizador, faz uma consulta ao servidor *Napster* para obter informações sobre o ficheiro desejado. O servidor *Napster* responde se o ficheiro desejado existe e, em caso positivo, devolve a sua localização para ser estabelecida uma ligação directa entre os computadores para que seja efectuado o download. Esta arquitectura é ilustrada pela Figura 6.

As interacções ponto a ponto e a troca de ficheiros são realizadas directamente entre os nós. O servidor central executa as tarefas de procura e identificação dos nós que guardam os ficheiros, pelo que representa um ponto de estrangulamento único, tornando este tipo de arquitecturas pouco escaláveis e susceptíveis a falhas e a ataques.

Figura 5: Arquitectura do *Napster*

De notar que existem sistemas que não se classificam na categoria de arquitecturas descentralizadas híbridas, mas que utilizam este tipo de administração centralizada até um certo ponto, como para inicialização do sistema (ex: *MojoNation*) ou para permitir a entrada de novos utilizadores na rede (rede *Gnutella*).

2.1.3.4 Redes não estruturadas

Em arquitecturas não estruturadas, a colocação dos conteúdos não está relaciona com a topologia da *overlay*. Normalmente, a localização dos conteúdos é obtida através de algoritmos de procura, como o método por inundação, através da propagação dos pedidos

pelos nós da rede até o conteúdo ser localizado. Os mecanismos de procura utilizados nas arquitecturas não estruturadas têm implicações, nomeadamente no que diz respeito à disponibilidade, escalabilidade e persistência. Este tipo de arquitecturas é, normalmente, mais apropriado para populações de nós altamente dinâmicos.

Gnutella: A rede *Gnutella* [6] é uma rede virtual *overlay* sobre a internet, não estruturada, totalmente descentralizada. A comunicação entre os clientes é feita através de 4 tipos de mensagens: *ping* – um pedido para que um dado *host* se apresente; *pong* – resposta a um pedido Ping contendo o endereço IP e o porto do *host* remetente e o número e os tamanhos dos ficheiros a partilhar; *query* – pedido de procura contendo a frase a procurar e os requisitos de velocidade mínima do *host* remetente; *query hits* – resposta a uma mensagem *query* contendo o IP, porto e a velocidade do *host* remetente, o número de ficheiros encontrados e a indexação do conjunto de resultados.

Arquitectura : Um cliente liga-se a uma rede *Gnutella* estabelecendo ligação com outros nós já existentes na rede que podem ser descobertos em bases de dados na internet. Posteriormente, o novo cliente envia uma mensagem de *Ping* a todos os nós a que está ligado e estes propagam a sua mensagem de *Ping* aos nós vizinhos que respondem com a mensagem de *Pong*. A propagação das mensagens *Ping* pelos nós, bem como das mensagens *Query* para localização de ficheiros, é feita através de um mecanismo por inundação (ou *broadcast*). Este mecanismo consiste no reencaminhamento das mensagens recebidas, por parte dos nós, para os seus vizinhos e o reencaminhamento das mensagens de resposta pelo caminho contrário de onde o pedido original chegou.

Cada mensagem possui um identificador único e cada nó mantém uma tabela dinâmica de encaminhamento, de forma a encontrar o caminho de volta para a mensagem de resposta, cujo ID é o mesmo da mensagem original. Os nós utilizam os identificadores únicos de cada mensagem para detectarem e descartarem mensagens duplicadas.

Procura e transferência : As procuras no *Gnutella* contêm uma frase com as palavras-chave da procura. Um ficheiro será encontrado com sucesso se o seu nome conter todos os termos procurados. As mensagens de procura possuem um campo que determina a velocidade mínima que um nó deve possuir para que esteja apto a retornar uma consulta, evitando que máquinas com baixa taxa de transmissão forneçam ficheiros. São limitadas a um tamanho de 256 bytes. Para além disso, de forma a limitar a dispersão das mensagens pela rede, cada cabeçalho das mensagens contém um campo TTL (*time-to-live*), sendo diminuído em cada retransmissão, que, ao chegar a zero, desencadeia a eliminação da mensagem.

Quando um ficheiro é identificado, em resposta a uma mensagem *Query*, o nó que desencadeou a procura recebe uma mensagem de *QueryHit* contendo o endereço IP e a porta onde o nó que tem o ficheiro pode ser acedido, iniciando o download através de uma ligação directa.

No geral, sistemas de distribuição de conteúdos em P2P não estruturado são apropriados para aplicações onde os conteúdos estejam replicados equitativamente pelos participantes, com populações altamente dinâmicas, onde os utilizadores aceitem uma abordagem de recuperação dos conteúdos do tipo *"best effort"* e com uma rede não muito grande de forma a evitar problemas de escalabilidade.

2.1.3.5 Redes estruturadas

Este tipo de redes emergiu como uma tentativa de tratar os problemas de escalabilidade que os sistemas originais de redes não estruturadas enfrentavam. Em redes estruturadas, a topologia do *overlay* é rigorosamente controlada e os ficheiros (ou ponteiros para eles) são colocados em locais específicos. Sistemas com este tipo de arquitectura fornecem um mapeamento entre os conteúdos e a sua localização na forma de tabelas de encaminhamento distribuído, para que os pedidos de procuras sejam encaminhados de forma eficiente.

Uma desvantagem dos sistemas estruturados deve-se ao facto de ser difícil de manter a estrutura necessária para um encaminhamento das mensagens eficiente perante populações de nós muito dinâmicas.

Chord: O sistema *Chord* [3] é uma infra-estrutura de localização e encaminhamento em P2P que realiza um mapeamento de ficheiros através de identificadores dos nós. A localização de dados implementada no *Chord* passa pela identificação dos dados (ficheiros) com chaves e guardando os pares (chave, dados) nos nós mapeadas pelas chaves.

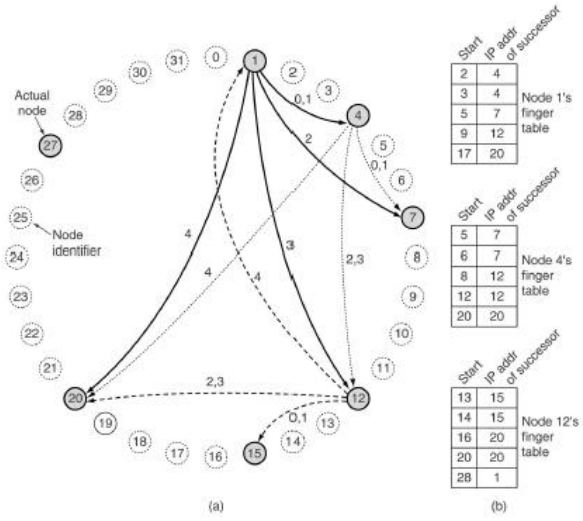

Figura 6: Envio de uma mensagem pela rede. (a) Conjunto de 32 identificadores de nós organizados em círculo. (b) Tabelas de *finger*.

Arquitectura: A rede *Chord* pode ter *n* utilizadores participantes, cada um dos quais irá armazenar índices e estará disponível para armazenar dados que podem ser acedidos por outros utilizadores. O endereço IP de cada utilizador pode ser mapeado para um número de

m bits através de uma função de *hash* consistente, como o SHA-1 [19]. Deste modo, é possível converter qualquer endereço IP num número de 160 *bits*, designado de identificador do nó. Todos os 2^{160} identificadores estão organizados num espaço de chave circular. A Figura 7 mostra o círculo de identificadores de nós para m = 5, ou seja, 2^5 identificadores. Os nós 1, 4, 7, 12, 15, 20 e 27 correspondem a nós reais e estão sombreados na Figura 7. Apenas esses nós fazem parte da rede.

Os identificadores são representados como um círculo de números de 0 a 2^m – 1. O sucessor(*k*) é definido como sendo o identificador do nó do primeiro nó real seguinte a *k*, no sentido horário. Por exemplo, o sucessor(5) = 7, sucessor(9) = 12 e sucessor(20) = 27. Os nomes dos dados (ficheiros) também são mapeados para números através da função de *hash* (SHA-1) para gerar um número de 160 *bits*, denominados chaves.

O *Chord* disponibiliza uma API que consiste em 5 funções principais: *insert(key, value)* – insere o par (chave, valor) em *r* nós específicos. O número *r* é um parâmetro do sistema que depende do grau de redundância desejado; *lookup(key)* – devolve o valor associado à chave *key*; *update(key, newval)* – actualiza o valor da chave *key* pelo *newval* (esta função só pode ser chamada pelo criador do valor com a chave *key*); *join()* – entrada de um novo nó na rede; *leave()* – saída de um nó da rede.

O sistema do *Chord* não possui uma operação explícita para eliminação de dados na rede. Uma forma utilizada consiste na chamada da função *update(key, value)*, em que o *value* corresponde à operação de eliminação. Se um utilizador deseja disponibilizar dados, primeiro deverá criar um par do tipo (*hash*('nome'), dados) e depois pedirá ao sucessor(*hash*('nome')) para armazenar o par, através da função *insert(hash*('nome'), dados). Se mais tarde algum utilizador quiser procurar por 'nome', utilizará o *hash* do 'nome' para obter a chave e chama a função *lookup*(chave).

Delegação de chaves **:** Quando um nó *x* entra na rede, algumas das chaves que estavam associadas ao sucessor do nó *x* vão ser delegadas ao novo nó *x*. Quando um nó sai da rede, todas as chaves que lhe estavam associadas vão ser delegadas ao seu sucessor. Desta forma é mantido um balanceamento da carga dos nós.

Localização simples de chaves **:** De um modo geral, a função de *lookup* funciona da seguinte forma: o nó solicitante envia um pacote ao seu sucessor contendo o seu endereço IP e a chave que procura. O pacote é propagado pelo anel até ser localizado o sucessor para o identificador que se procura. Cada nó contém o endereço IP do seu sucessor e do seu predecessor, de forma a que as procuras possam ser enviadas no sentido horário ou anti-horário, dependendo do percurso mais curto. Mesmo com duas opções de sentido, a pesquisa linear de todos os nós é muito ineficiente num sistema P2P de grande porte, pois o número médio de nós exigidos por pesquisa é N/2.

Localização escalável de chaves : Para tornar a busca mais rápida, o *Chord* mantém informações adicionais de outros nós. Estas informações são armazenadas numa tabela de

m entradas, denominada de tabela de *fingers*. Cada uma das entradas tem dois campos: início e o endereço IP do seu sucessor, como mostram os exemplos apresentados na Figura 7 (b). Os valores dos campos correspondentes à entrada i no nó k são: início = $k + 2^i$ (módulo 2^m); endereço IP de sucessor(inicio[*i*]).

A tabela de *fingers* de um nó não contém informações suficientes para determinar directamente o sucessor de uma chave *k* aleatória. Por exemplo, o nó 1 na Figura 7 (a) não pode determinar o sucessor da chave 14, uma vez que o sucessor (nó 15) não aparece na sua tabela de *fingers*. Utilizando a tabela de *fingers*, a pesquisa da chave no nó *k* prossegue da seguinte maneira: se a chave estiver entre *k* e o sucessor(*k*), então o nó que contém a informação sobre a chave é o sucessor(*k*). Caso contrário, a tabela de *fingers* é consultada para determinar a entrada cujo campo início é o predecessor mais próximo da chave. A seguir, uma solicitação é enviada directamente ao endereço IP contido nessa entrada da tabela de *fingers*, solicitando que ele continue a pesquisa. Tendo em vista que cada pesquisa reduz para metade a distância restante até o destino, é possível mostrar que o número médio de pesquisa é $\log_2 n$.

O sistema *Chord* permite fornecer as seguintes propriedades: escalabilidade, disponibilidade, balanceamento de carga, dinamismo, actualização dos dados e localização de acordo com a proximidade (caso o resultado de uma procura se encontre próximo do nó, este não tem de contactar nós distantes). No entanto, existem alguns aspectos negativos nestes sistemas, no que diz respeito ao desempenho que se degrada à medida que as informações de encaminhamento vão ficando inválidas, devido a entradas e saídas de nós; a disponibilidade apenas se mantém desde que as falhas dos nós sejam independentes; uma vez que a topologia *overlay* não é baseada na topologia da rede física subjacente, uma falha da rede IP pode manifestar-se como múltiplas falhas de ligações espalhadas pela *overlay*.

Pastry: O *Pastry* [4] é uma base escalável de encaminhamento e localização de objectos distribuídos para aplicações P2P de grande escala. O *Pastry* desempenha o encaminhamento a nível da aplicação e a localização de objectos numa vasta rede *overlay* de nós ligados através da internet. Este pode ser usado para suportar uma variedade de aplicações P2P, incluindo armazenamento de dados, partilha de dados e comunicação entre grupos. Uma das propriedades do *Pastry* é a compreensão de proximidade de localização; este procura minimizar a distância percorrida pelas mensagens, de acordo com uma métrica escalar de proximidade, como o número de saltos do encaminhamento IP.

Arquitectura: No *Pastry*, a cada nó da rede é atribuído um identificador (*nodeId*) de 128 *bits* que pode ser gerado a partir de uma função de *hash* aplicada ao seu endereço IP ou à sua chave pública. O *nodeId* é usado para indicar a posição do nó num espaço de chave circular de 0 à $2^{128} - 1$. Uma chave é mapeada para um nó cujo *nodeId* está numericamente mais próximo da identificação da chave. Assumindo que uma rede consiste em *N* nós, o *Pastry*

pode encaminhar qualquer mensagem em O($log_2^b N$) *hops* (*b* é um parâmetro de configuração).

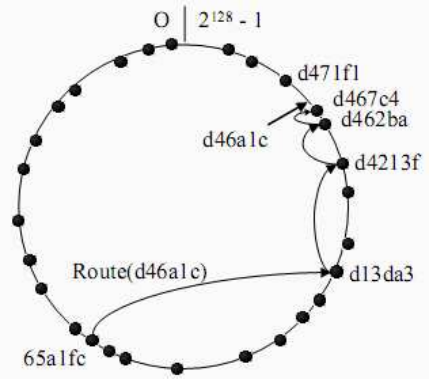

Figura 7: Encaminhamento de uma mensagem. Envio do nó 65a1fc com a chave d46a1c. Os pontos definem os nós ligados. [20]

Para efeitos de encaminhamento, o *nodeId* e as chaves são considerados como uma sequência de dígitos com base 2^b. O *Pastry* encaminha as mensagens para o nó cujo *nodeId* está numericamente mais próximo da chave pesquisada. Isto é feito da seguinte forma: a cada etapa do encaminhamento, um nó normalmente encaminha as mensagens para outro nó cujo *nodeId* partilhe com a chave pelo menos um dígito (ou *b bits*) a mais do que é partilhado com o nó actual. Se nenhum nó é conhecido, a mensagem é encaminhada para um nó cujo *nodeId* partilha um prefixo com a chave e está numericamente mais próximo da chave do que o presente nó, como pode ser visto na Figura 8.

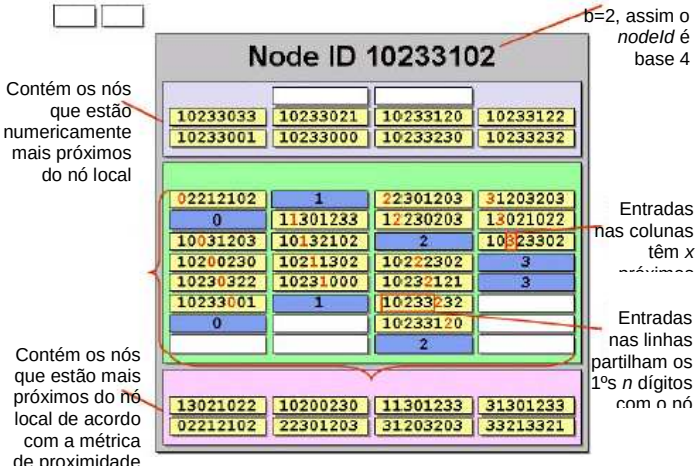

Figura 8: Tabela de encaminhamento do Pastry. As células a azul mostram o dígito correspondente ao *nodeId* do nó actual. Os *nodeIds* em cada entrada estão divididos de forma a mostrarem: [prefixo comum prók. dígito resto].

Para suportar este procedimento de encaminhamento, cada nó mantém uma tabela de encaminhamento, um conjunto de vizinhanças e um conjunto de folhas (*leafs*). Uma tabela

de encaminhamento é formada por | $log_2^b N$ | linhas, cada uma com 2^b-1 entradas. Cada entrada na tabela de encaminhamento contém o endereço IP de potenciais nós cujo *nodeId* tem um prefixo apropriado. O conjunto de vizinhança *M* contém os *nodeIds* de |*M* | nós que estão mais próximos (de acordo com uma métrica de proximidade) do nó local. O conjunto de vizinhança é normalmente utilizado no encaminhamento das mensagens. O conjunto de folhas, *L*, é formado pelos |*L*|/2 nós sucessores e |*L*|/2 nós predecessores mais próximos de um dado nó. A Figura 9 apresenta um estado hipotético de um nó *Pastry* com o *nodeId* 10233102 (base 4), num sistema de 16 *bits* para identificação e um valor de *b* = 2.

Cada 2^b-1 entradas na linha *n* da tabela de encaminhamento referem o nó cujo *nodeId* partilha os primeiros *n* dígitos do *nodeId* do nó actual, mas cujo *n*+1º dígito é um dos 2^b-1 valores possíveis, excepto o *n*+1º dígito do *nodeId* do nó actual.

Encaminhamento: Dada uma mensagem, o nó verifica se a chave está dentro da faixa de endereços coberto pelo conjunto de folhas. Em caso afirmativo, a mensagem é encaminhada directamente para o nó destinatário. Isto quer dizer que existe um nó no conjunto de folhas que está mais próximo da chave procurada (possivelmente no presente nó). Se a chave não é encontrada no conjunto de folhas, então é usada a tabela de encaminhamento e a mensagem é encaminhada para o nó que partilha um prefixo comum com a chave (pelo menos um ou mais dígitos). Em certos casos, é possível que o nó associado não esteja alcançável, sendo a mensagem encaminhada para um nó que partilhe um prefixo com a chave e que esteja numericamente mais próximo da chave do que o nó actual.

Pastry API : Nesta secção é descrita a API exportada pelo *Pastry* para aplicações. O *Pastry* exporta as seguintes operações:
nodeId = pastryInit(Credentials) insere um novo nó numa rede *Pastry* existente (ou cria uma nova) e inicializa todos os estados necessários, retornando o *nodeId* do nó. As *Credentials* são fornecidas pela aplicação e contêm informação necessária para autenticar o nó local e para o associar de forma segura à rede;
route(msg, key) encaminha a mensagem *msg* para o nó com o *nodeId* numericamente mais próximo da chave *key*;
send(msg, IP-addr) envia a mensagem *msg* para o nó com o endereço *IP-addr*. A mensagem é recebida pelo nó através do método de *deliver*.

Aplicações sobre o *Pastry* devem exportar as operações:
deliver(msg, key) chamado pelo *Pastry* quando é recebida uma mensagem e o nó local possui o *nodeId* mais próximo numericamente que qualquer outro *nodeId*, ou quando é recebida uma mensagem que foi transmitida pelo método *send*, utilizando o endereço IP do nó local;
forward(msg, key, nextId) chamado pelo *Pastry* no momento anterior a uma mensagem ser encaminhada para o nó com o *nodeId* = *nextId*. A aplicação pode alterar os conteúdos da mensagem *msg* ou do valor do *nextId*. Alterando o *nextId* para NULL irá parar a mensagem

no nó local;
newLeafs(leafSet) chamado pelo *Pastry* quando há alterações no conjunto de folhas (*leafset)*.

Adição e falha de nós **:** Um novo nó com um novo *nodeId X* pode inicializar o seu estado contactando com um nó *A* próximo e pedir a *A* que encaminhe uma mensagem especial usando *X* como chave. Esta mensagem é encaminhada para o nó *Z* que possui um *nodeId* numericamente mais próximo ao de *X* (no caso incomum de *X* ser igual ao *nodeId* de *Z*, o novo nó deverá obter um novo *nodeId*). Posteriormente, o novo nó obtém o conjunto de folhas de *Z* e a linha *i* da tabela de encaminhamento do nó *i* encontrado no caminho entre *A* e *Z*. Com esta informação, o novo nó pode inicializar o seu estado e notificar os nós que precisam de saber da sua chegada.

Para tratar da falha de nós, nós vizinhos no espaço de *nodeIds* (que se apercebem de cada um pelo facto de pertencerem ao conjunto de folhas de cada um) trocam periodicamente mensagens de *keep-alive*. No caso de um nó não enviar uma dessas mensagens num período *T,* é presumido como falhado. Todos os membros do conjunto de folhas do nó falhado são notificados e actualizam os seus conjuntos. Um nó que recupere de uma falha contacta os nós do seu último conhecido conjunto de folhas, obtém os seus conjuntos de folhas, actualiza o seu conjunto e notifica os membros do seu novo conjunto da sua presença.

No geral, este tipo de sistemas estruturados apresentam soluções relativamente semelhantes no encaminhamento e localização em ambientes *peer-to-peer* distribuídos, focando, contudo, em diferentes aspectos e dando prioridade a diferentes problemas de desenho. Uma das grandes limitações dos sistemas estruturados *peer-to-peer* deve-se ao facto de apenas suportarem procuras exactas. É necessário conhecer a chave exacta dos dados pretendidos para localizar o nó que os guarda.

2.2 GRIDS INSTITUCIONAIS

Computação em *Grid* é o termo utilizado para referir uma técnica computacional que utiliza os recursos de diferentes computadores, com o intuito de resolver problemas de grande complexidade e/ou volume, estando portanto, não limitada apenas à execução distribuída de algoritmos de processamento, mas também à gestão de grandes quantidades de dados distribuídos. Explorando mais a definição de *Grid*, é possível utilizar-se da definição de *Buya*: um tipo de sistema distribuído e paralelo que possibilita a partilha, selecção e agregação dinâmica, em tempo de execução, de recursos autónomos geograficamente distribuídos, de acordo com a sua disponibilidade, capacidade, performance, custo e requisitos do utilizador na qualidade de serviço [21]. A *Grid* é distinguida da computação distribuída convencional pelo seu foco na partilha de recursos em grande escala, aplicações inovadoras e, em alguns casos, orientação de alto desempenho [22]. De certa forma, a computação em *Grid* encontra-se na mesma área de actuação dos clusters e super-computadores (muitas vezes actuando em conjunto com ambos).

Com o poder de processamento dos computadores actuais a aumentar regularmente, aumenta a complexidade (e o conjunto) de problemas que podem ser resolvidos com a utilização de *Grids*.

Com a disseminação da Computação em *Grid*, surgiram diversos sistemas desenvolvidos pela indústria e pela comunidade académica. Nas próximas secções são apresentados alguns dos sistemas de Computação em *Grid* existentes.

2.2.1 Globus

O projecto *Globus* [5] é um projecto que provocou grande impacto na área de Computação em *Grid*. O seu sistema de Computação em *Grid* é denominado *Globus Toolkit* e fornece uma série de funcionalidades que permitem a implementação de sistemas de Computação em *Grid*, assim como o desenvolvimento de aplicações para tais sistemas. A abordagem da caixa de ferramentas (*toolkit*) permite a personalização das *Grids* e aplicações, permitindo a criação incremental de aplicações que, a cada nova versão, utilizem mais recursos da *Grid*. Por exemplo, pode-se inicialmente utilizar o *Globus* apenas para coordenar a execução de aplicações sem que estas sejam alteradas, e posteriormente, é possível modifica-las para que usem serviços da *Grid*, como por exemplo transferência de ficheiros.

Conceitos: É utilizado o termo *metacomputer* para designar uma rede virtual de super-computadores, construída dinamicamente a partir de recursos distribuídos geograficamente, com ligações de grandes débitos. Um *metasystem* caracteriza-se pela selecção de recursos para as diferentes aplicações, de acordo com os seus critérios de conectividade, custo, segurança e fiabilidade; heterogeneidade em vários níveis; estrutura imprevisível, isto é, o ambiente de execução é construído dinamicamente a partir dos recursos disponíveis; comportamento dinâmico e imprevisível num ambiente onde os recursos são partilhados e consequentemente susceptíveis de variações de desempenho e comportamento durante o tempo; existência de múltiplos domínios administrativos.

The Globus Toolkit: O *Globus Toolkit* é caracterizado como um conjunto de serviços para a construção de sistemas e aplicações em *Grid*. São descritos a seguir alguns dos principais serviços disponíveis no *toolkit*.

Alocação e Descoberta de Recursos: Quando um utilizador submete uma aplicação para execução no *Grid* utiliza um escalonador de aplicação (*application scheduler*) para escolher os recursos a utilizar, particiona o trabalho entre os recursos e envia as tarefas para os escalonadores de recursos. Os escalonadores de recursos são acedidos através do serviço GRAM (*Globus Resource Allocatin Manager*) que fornece uma interface única que permite submeter, monitorizar e controlar tarefas de forma independente do escalonador de recursos. Para facilitar ainda mais a tarefa dos escalonadores de aplicação, é disponibilizado o MDS (*Metacomputing Directory Service*), um serviço de informação sobre o *Grid* que contém informações sobre os recursos que formam a *Grid* (como quantidade de memória, velocidade da CPU, número de nós ligados em paralelo ou número e tipo de interfaces de rede

disponíveis), informações sobre o seu estado (desempenho, disponibilidade e carga da rede) e informações específicas às aplicações (como requisitos de memória).

Segurança e Autenticação **:** Um aspecto que complica o uso das *Grids* na prática é a autenticação de utilizadores em diferentes domínios administrativos. A GSI (*Globus Security Infrastructure*) é o serviço *Globus* que ataca este problema. Permite efectuar um login único na *Grid*. A GSI utiliza criptografia de chave pública, certificados X.509 e comunicação SSL (*Secure Sockets Layer*) para estabelecer a identidade *Globus* do utilizador. Depois do utilizador se identificar perante a GSI, todos os restantes serviços *Globus* saberão, de forma segura, que o utilizador é de facto quem diz ser.

O *Globus Toolkit* é uma das mais significantes implementações baseada em infra-estruturas de *Grids*. De uma forma geral, o modelo de programação do *Globus* é baseado na utilização de bibliotecas especializadas para a submissão remota de tarefas e controlo e implementa serviços de autenticação e autorização dos utilizadores, submissão de tarefas, transferência de dados, etc.

2.2.2 MyGrid

A motivação para a construção do *MyGrid* [23] surgiu do facto de que, embora tenha sido realizada muita pesquisa para o desenvolvimento dos *Grids* Computacionais, são poucos os utilizadores que executam as suas aplicações paralelas sobre essa infra-estrutura. Assim, o projecto *MyGrid* ataca apenas aplicações *Bag of Tasks*, isto é, aplicações cujas tarefas são independentes, podendo ser executadas por qualquer ordem. As aplicações *Bag of Tasks* são um alvo interessante porque se adequam melhor à ampla distribuição, heterogeneidade e dinamismo da *Grid*.

Arquitectura**:** O *MyGrid* define 2 tipos de máquinas, Máquina Base (*Home Machine*) e Máquina de *Grid* (*Máquina de Grids*). No *MyGrid*, a Máquina Base é aquela que controla a execução da aplicação. Tipicamente contém os dados de entrada e recolhe os resultados da computação. A Máquina Base é normalmente usada pelo utilizador directamente no seu dia-a-dia. Todas as máquinas usadas via *MyGrid* para executar tarefas são definidas como máquinas de *Grid*. As máquinas de *Grid*, tipicamente, não possuem o mesmo sistema de ficheiros ou os mesmos softwares instalados, pelo que é necessário manipular estas máquinas, através de abstracções criadas pelo *MyGrid*.

O *MyGrid* define o *Máquina de Grid Interface* como sendo um conjunto mínimo de serviços necessários para que uma dada máquina possa ser adicionada ao *Grid* do utilizador. Esses serviços devem possibilitar a criação e cancelamento de processos e transferência de ficheiros entre a Máquina de *Grid* e a Máquina Base, em ambas direcções. Uma das formas de implementação do *Máquina de Grid Interface* é fornecer ao sistema scripts que implementam os serviços listados anteriormente. Neste caso, o *MyGrid* utiliza o módulo *Grid Script* que utiliza ferramentas de linha de comandos para implementar as 4 operações necessárias nas Máquinas de Grid: *ftp* ou *scp* para a troca de ficheiros, e *ssh* para a criação e cancelamento de

tarefas. Tal método deve ser utilizado apenas em último caso, pois apresenta problemas de desempenho. Outra forma passa pela implementação de uma *Globus Proxy* que permite ao *MyGrid* aceder a máquinas geridas pelo *Globus* através de *Grid Services*. A *Globus Proxy* redirecciona as operações necessárias às Máquinas de *Grid* para os *Grid Services* adequados. Utiliza-se o *Grid Service* para execução de tarefas e o *GridFTP* para a transferência de ficheiros. Finalmente, o *MyGrid* fornece também um mecanismo de acesso a Máquinas da *Grid*, chamado *User Agent*. O *User Agent* é um pequeno *daemon* escrito em Java que implementa as operações definidas pela Máquina de *Grid*. É a implementação ideal a ser utilizada quando o utilizador pode instalar software nas máquinas remotas. O *User Agent* não necessita de nenhum acesso especial à máquina, sendo necessário apenas uma área de disco para escrita e leitura de ficheiros.

Outro componente fundamental da arquitectura é o *Scheduler*, que recebe do utilizador a descrição das tarefas a executar, escolhe qual processador a utilizar para cada tarefa e submete e monitoriza a execução da tarefa. O *MyGrid* possui actualmente duas heurísticas de escalonamento: *Workqueue with Replication* (WQR) [24] e *Storage Affinity* [25]. Ambas conseguem obter um bom desempenho, mesmo sem utilizar informações sobre o estado da *Grid* ou o tamanho de cada tarefa. O WQR foi definido para aplicações de intenso processamento, enquanto o *Storage Affinity* foi desenvolvido para melhorar o desempenho de aplicações que processam grandes quantidades de dados.

2.2.3 OurGrid

O *MyGrid* é uma solução direccionada para o utilizador utilizar os recursos dos quais dispõe. No entanto, não há nenhuma forma directa que permita que um utilizador utilize os recursos de terceiros, a menos que o utilizador explicitamente negoceie o acesso aos recursos com os seus proprietários, o que costuma ser difícil. Desta forma, os programadores do *MyGrid* arquitectaram o *OurGrid* [26], uma comunidade *peer-to-peer* para partilha de recursos. Ao contrário do *Globus*, a solução *OurGrid* tem um objectivo diferente, todavia complementar. O objectivo é prover uma solução efectiva para a execução de aplicações *Bag-of-Tasks* em *Grids* Computacionais.

Arquitectura: O *OurGrid* é formado por três componentes: *MyGrid Broker* [23], *OurGrid Peer* [26] e uma solução de *Sanboxing* baseado no *Xen* [27]. O *OurGrid* explora a ideia de que uma *Grid* é composta por vários aglomerados de máquinas que têm o interesse em trocar favores computacionais entre si. Portanto, existe uma rede *peer-to-peer* de troca de favores que permite que os recursos ociosos de um site sejam disponibilizados para outro quando solicitado. Para manter o equilíbrio do sistema, numa situação de contenção de recursos, os sites que doaram mais recursos (quando estes estavam ociosos) deverão ter prioridade quando solicitarem recursos.

A Figura 10 ilustra a ideia da rede de favores, onde cada nó controla um conjunto de recursos de um site. Ao surgir uma procura interna por recursos que o nó de um determinado site não

consegue fornecer, este nó irá fazer solicitações à comunidade. A ideia é que os nós utilizem um esquema de prioridades baseado nos seus consumos em relação aos outros [28].

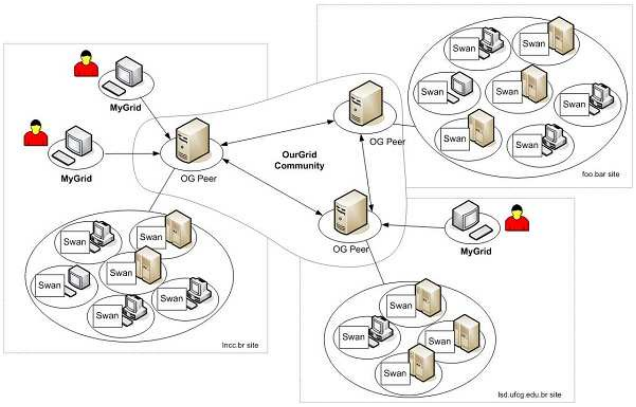

Figura 9: Arquitectura *OurGrid* [29]

Segurança e Autenticação: O utilizador pode ter acesso directo a alguns recursos (isto é, *Grid Machines* - GUMs - na sua rede local). Neste caso o utilizador utiliza o esquema de autenticação tradicional, que implica, no geral, a utilização de um nome de utilizador e uma palavra-chave. Contudo, para além das GUMs a que o utilizador tem acesso directo, o *OurGrid* permite (e promove) a obtenção de acesso a GUMs de outros sites. Assim, para as GUMs obtidas da comunidade, há uma autenticação com duas vias (cliente e servidor), baseada em certificados digitais no formato X.509.

Descoberta e Alocação de Recursos: Para executar uma aplicação, usando o *OurGrid*, o utilizador deve descrever sua aplicação. A descrição da aplicação consiste: num conjunto tarefas, nos ficheiros necessários à execução, ficheiros de saída e os seus requisitos (por exemplo, sistema operacional necessário, mínimo de memória, arquitectura do processador, etc.). Em seguida, o utilizador submete a sua aplicação para execução na *Grid* através do *MyGrid Broker*. O componente interno do *MyGrid Broker* que recebe a submissão é o *Scheduler*. Por sua vez, o *Scheduler* solicita, aos fornecedores de GUMs, recursos para executar a aplicação submetida pelo utilizador. Esses fornecedores podem responder com recursos locais ou recursos obtidos na rede de favores. Após ter sido descoberto um recurso que possui atributos compatíveis com os requisitos da aplicação, o recurso é alocado e repassado para o *Scheduler* que o solicitou. Contudo, caso o recurso tenha sido descoberto através da rede de favores, o recurso pode ser reivindicado pelo nó que o forneceu. A alocação dos recursos é efectuada ao nível do *MyGrid Broker*, pelo que os recursos não estarão dedicados de forma exclusiva, permitindo a outras aplicações, que não usam a infra-estrutura do *OurGrid*, sejam executadas concorrentemente com a aplicação submetida pelo utilizador.

A característica mais importante do *OurGrid* é conseguir fornecer uma solução útil e eficiente para uma comunidade de utilizadores em produção, apesar de se basear em soluções simples e de intento limitado (apenas para aplicações do tipo *Bag-of-Tasks*). De uma forma geral, este projecto controla agrupamentos de *grid* de uma forma *peer-to-peer*, através do escalonamento

de aplicações para serem executadas em nós remotos e gestão de recursos utilizando uma rede de favores.

2.2.4 Conclusões

Nesta secção foram apresentados os principais aspectos das *Grids* Institucionais, desde a apresentação de um conceito que classifica uma *Grid* Institucional, até o estado actual do desenvolvimento da tecnologia para a construção de infra-estruturas *Grid* (alguns exemplos de sistemas *Grid* na tabela 2).

Projecto	Objectivos e tecnologias desenvolvidas
Globus	Infra-estrutura de software para computações que integram recursos de computação e informação distribuídos geograficamente
Legion	*Metasystem* baseado em objectos. Suporta escalonamento transparente, gestão de dados, tolerância a faltas, autonomia de sites e opções de segurança
InteGrade	*Middleware* que constrói uma hierarquia de aglomerados para melhorar o desempenho de aplicações MPI, sobre uma implementação CORBA
OurGrid	Coliga sites com aglomerados *(clusters) de Grids* de uma forma P2P, através do escalonamento de aplicações para nós remotos e gestão de recursos utilizado uma Rede de Favores
XtremWeb	*Toolkit* baseado em Java para o desenvolvimento de uma infra-estrutura de proveito de ciclos para aplicações em larga escala

Tabela 2: Descrição de alguns sistemas de *Grid*

Vimos que as *Grids* Institucionais levantam questões em várias áreas de pesquisa. Contudo, os seus benefícios vão além de uma simples plataforma para execução de aplicações em larga escala. A ideia é facilitar a colaboração de grupos ou instituições distribuídos geograficamente.

Quanto aos aspectos técnicos propriamente ditos, há diversas questões em aberto na área. Em particular, a urgente necessidade de progresso (i) na criação de modelos de programação que melhor exponham a natureza das *Grids* ao programador, e (ii) em melhores formas de lidar com grandes quantidades de dados numa *Grid*. As *Grids* são mais complexas e dinâmicas que outras plataformas para execução de aplicações paralelas. Para garantir o desempenho é necessário conceber abstracções úteis, que permitam ao programador codificar estratégias de adaptação à complexidade das *Grids*.

Finalmente, a discussão sobre os padrões emergentes para o desenvolvimento de infra-estruturas de *Grids* Institucionais, mostra que os esforços têm amadurecido, fomentado a

pesquisa e o desenvolvimento de ferramentas, que contribuem para a concretização de um ambiente amplamente distribuído onde será possível consumir serviços computacionais de forma transparente.

2.3 CYCLE SHARING / DESKTOP GRIDS

Actualmente, é possível encontrar uma vastidão de poder computacional nas centenas de milhões de computadores pessoais espalhados pelo mundo. A computação de recursos públicos obtém imensas computações distribuídas através da recolha de recursos ociosos em computadores ligados à internet. A ideia de utilizar os ciclos de processamento ociosos na computação distribuída foi proposta em 1978 pelo projecto de computação *Worm*, na Xerox PARC, envolvendo cerca de 100 máquinas para medirem o desempenho da *Ethernet* [30]. Esta ideia foi mais tarde explorada por projectos académicos, como o *Condor*, em 1988. Com o crescimento da internet, nos meados dos anos 90, apareceram dois projectos de computação de recursos públicos, o GIMPS (para procura de números primos) e Distributed.net (para demonstração de decifra utilizando *brute-force*). Em 17 de Maio de 1999, o aparecimento do SETI@*home* atraiu milhões de participantes pelo mundo. Actualmente, com mais de 1.6 milhões de computadores no sistema, o SETI@*home* tem a capacidade de computar a mais de 380 *TeraFLOPS*. A experiência mostra que não só existem recursos ociosos disponíveis na Internet, como também muitos utilizadores estão dispostos a partilhar os seus ciclos de processamento (*cycle sharing*).

Um maior interesse e atracção, no que diz respeito à área das *Desktop Grids*, têm vindo a aumentar, graças ao sucesso de vários exemplos populares como o SETI@*home*. Em seguida, são analisados vários exemplos de projectos para partilha e aproveitamento de recursos de outras máquinas.

2.3.1 Condor

O *Condor* teve a sua origem em 1988, derivado de um sistema ainda mais antigo, *RemoteUnix* [31], que possibilitava a integração e o uso remoto de estações de trabalho. O sistema *Condor* [14] possibilita a integração de diversos computadores em aglomerados, de forma a permitir uma utilização eficaz dos recursos computacionais de tais máquinas. Grande parte da capacidade de processamento dos computadores, especialmente estações de trabalho, permanece inutilizada durante grande parte do tempo. Assim, o principal objectivo do *Condor* é utilizar tal capacidade para executar programas, preservando o proprietário do recurso de perdas de desempenho. Contudo, este sistema garante apenas que o trabalho será eventualmente concluído. O utilizador submete ao *Condor* tarefas independentes, isto é, que não necessitam de comunicação entre tarefas durante a sua execução (aplicações *Bag of Tasks*).

Arquitectura: A organização das máquinas participantes no sistema *Condor* consiste na formação de um aglomerado de máquinas (*Condor Pool*). Normalmente as máquinas pertencentes a um aglomerado situam-se num mesmo domínio administrativo, como um

departamento de uma empresa ou uma rede local. No entanto, tal organização não é obrigatória. Dentro do aglomerado, existem diversos módulos:

Schedd: permite que um utilizador solicite execuções ao sistema *Condor*, devendo por isso estar presente em todas as máquinas das quais se deseja submeter aplicações para execução. Permite também que o utilizador monitorize e controle remotamente a execução da aplicação;

Startd: permite que o sistema *Condor* execute aplicações na máquina local. Assim, este módulo deve ser executado em cada máquina para a qual se deseja submeter aplicações para execução. Além de iniciar aplicações, o *Startd* também publica periodicamente informações sobre o uso e disponibilidade de recursos da máquina em questão e pode ser configurado de maneira a fazer valer a política de partilha de recursos estabelecida pelo proprietário.

Collector: módulo responsável por manter informações do aglomerado e receber solicitações de execução. Periodicamente, cada *Startd* envia para o *Collector* informações sobre a disponibilidade de recursos. Da mesma forma, quando um utilizador solicita uma execução através do *Schedd*, este envia para o *Collector* uma solicitação a informar as necessidades da aplicação. O *Collector* pode também ser acedido por ferramentas externas de modo a fornecer informações sobre o funcionamento do aglomerado. Tipicamente existe um *Collector* em cada aglomerado;

Negotiator: é o escalonador do aglomerado. Periodicamente, o *Negotiator* consulta o *Collector* com o objectivo de determinar se existem solicitações de execução. Caso existam, o *Negotiator*, baseado nas informações fornecidas pelo *Collector*, emparelha solicitações de execução com máquinas que possam atender tais solicitações.

No *Condor*, tanto as solicitações de execução de uma aplicação como as ofertas de recursos são definidas em termos de uma estrutura de dados, *ClassAds* [32]. Um *ClassAd* é um conjunto de expressões que representam a disponibilidade e a necessidade de recursos. As expressões são pares do tipo (atributo, valor) e podem incluir números, *strings* e intervalos, entre outros. Uma característica dos *ClassAds* é que eles não possuem um esquema fixo, ou seja, nem todos os *ClassAds* precisam de possuir os mesmos campos.

O processo de execução remota de aplicações ocorre da seguinte forma: um utilizador submete ao *Schedd* da sua máquina uma aplicação para execução remota, associada a um anúncio de pedido de recursos. Esse anúncio é enviado ao *Collector*. De forma análoga, cada máquina que cede recursos anuncia periodicamente a disponibilidade dos mesmos, assim como quaisquer restrições à sua utilização. Periodicamente, o *Negotiator* associa anúncios de solicitações e ofertas (*Matchmaking*) [32] de acordo com um algoritmo e notifica o *Schedd* da máquina que realizou a solicitação. Neste ponto, o *Schedd* contacta o *Startd* da máquina que ofereceu os recursos, de forma a determinar se os recursos da oferta continuam válidos.

Checkpointing: O *Condor* aborda esta questão fazendo o *checkpoint* da tarefa (isto é, salvando transparentemente o estado da sua execução). Isto permite que a tarefa seja executada noutra

máquina a partir do ponto em que parou. A abordagem para *checkpointing* do *Condor* [33] possui a grande vantagem de não exigir alterações nas aplicações. Entretanto, a capacidade do mecanismo é limitada: por exemplo, não há nenhum suporte para lidar com comunicação entre processos, processos que façam uso de chamadas de sistema *fork*() ou *exec*(), entre outros. Também não há *checkpointings* para aplicações paralelas.

Aplicações paralelas: O *Condor* permite a execução de aplicações paralelas do tipo MPI (aplicações que requerem comunicação com outras aplicações durante a sua execução) e do tipo PVM (aplicações que possibilitam a sua execução em máquinas heterogéneas). Entretanto, o suporte a aplicações do tipo MPI é limitado [34] pois na prática impede a execução de aplicações MPI sobre recursos partilhados. Em contrapartida ao MPI, o *Condor* permite a execução de aplicações PVM em recursos partilhados, uma vez que o próprio modelo PVM permite que os nós sejam adicionados ou removidos de uma aplicação em execução.

O *Condor* é outra importante infra-estrutura para a partilha de recursos, suportando um conjunto de serviços, destacando-se o *checkpointing*, que permite suspender uma computação e o seu estado para ser posteriormente executada noutro nó. No entanto, o *checkpointing* é dependente da máquina onde se executa, pelo que representa uma grande limitação já que os clientes são muito heterogéneos.

2.3.2 SETI@home

Antes do SETI@*home*, os projectos do SETI utilizavam super-computadores para a análise dos dados. Em 1995, foi proposta a utilização de um super-computador virtual que consistia num grande número de computadores ligados à internet, o que levou ao desenvolvimento do projecto SETI@*home*. Este projecto utiliza os dados recolhidos pelos radiotelescópios, dividindo-os em pequenas unidades de trabalho que possam ser analisados por computadores pessoais comuns. Para isso, o projecto conta com a participação voluntária de utilizadores que cedem os seus ciclos de processamento para a análise dos sinais de rádio.

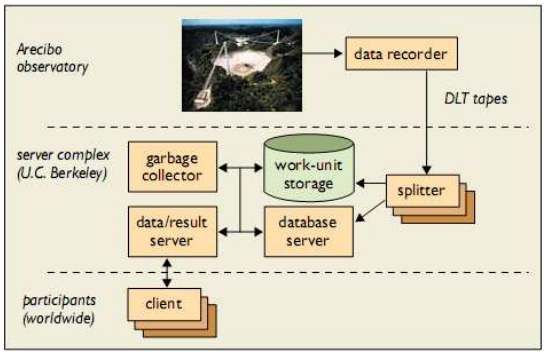

Figura 10: Arquitectura do SETI@*home* [1]

Arquitectura**:** O modelo computacional do SETI@*home* é simples. Os dados do sinal de rádio são divididos em unidades de trabalho de tamanho fixo e distribuídos por um complexo de

servidores (Figura 11), via internet, a um programa cliente que corre nos computadores voluntários. O programa cliente processa um resultado, devolve-o ao servidor e obtém uma nova unidade de trabalho. Não existe comunicação entre os clientes.

O SETI@*home* realiza computação redundante, com dois a três níveis de redundância. Cada unidade de trabalho é processada múltiplas vezes de forma a serem detectados e descartados os resultados errados devido a erros de processamento ou utilizadores maliciosos.

Os componentes do complexo de servidores são a seguir descritos: é utilizada uma base de dados relacional (*Informix*) para armazenar informações sobre as unidades de trabalho, resultados, utilizadores entre outros; um servidor multi-tarefas de dados/resultados distribui as unidades de trabalho para os clientes através de um protocolo baseado em HTTP, para que os clientes atrás de *firewalls* ou NATs possam contactar o servidor; um programa de *garbage collector* remove unidades de trabalho do disco. A política usada consiste na eliminação das unidades de trabalho que foram enviadas *M* vezes, sendo *M* um valor ligeiramente superior ao nível de redundância aplicado. Este método elimina o ponto de estrangulamento da produção das unidades de trabalho.

Para evitar a ocorrência de falhas, a arquitectura foi desenvolvida com vista a minimizar as dependências entre o servidor e subsistemas, como por exemplo, o servidor de dados/resultados pode ser executado num modo que em vez de usar a base de dados para enumerar as unidades de trabalho a enviar, o servidor obtém a informação de um ficheiro em disco, permitindo a distribuição de dados quando a base de dados se encontra desactivada.

Funcionamento do lado do cliente: O programa cliente recebe repetidamente uma unidade de trabalho do servidor de dados/resultados, procede á sua análise, podendo ser configurado para processar apenas quando a máquina estiver disponível ou continuamente com baixa prioridade, e retorna os resultados ao servidor. A ligação à internet, apenas é necessária quando se comunica com o servidor. O programa cliente escreve periodicamente o seu estado num ficheiro em disco, que será aberto ao iniciar o computador, possibilitando a retoma do progresso do trabalho quando o computador é desligado.

Tratamento dos resultados: Os resultados são enviados para o complexo de servidores onde vão ser armazenados e analisados. O seu tratamento consiste em duas tarefas: científica – o servidor escreve o resultado para um ficheiro no disco. Um programa lê os ficheiros, originando registos dos resultados na base de dados. Para efeitos de optimização, várias cópias do programa são executadas em paralelo; contabilização – para cada resultado, o servidor escreve uma entrada num ficheiro de *log* descrevendo o utilizador do resultado, o tempo de CPU consumido, etc. Os ficheiros de *log* são lidos por um programa que acumula numa cache as actualizações para todos os registos relevantes da base de dados (como o utilizador, a equipa, o país e o tipo de CPU). Os dados desta cache são enviados para a base de dados de alguns em alguns minutos. Desta forma, as actualizações são mantidas no disco durante algum tempo, diminuindo os períodos de grande utilização e interrupção da base de dados.

O SETI@*home* foi um dos grandes pioneiros no aproveitamento de ciclos de processamento em *peer-to-peer*, permitindo a resolução de problemas científicos complexos através da sua partição em unidades de trabalho e distribuição por vários clientes. No entanto, a sua implementação apresenta limitações como a existência de um ponto único de falha e de estrangulamento, a ausência de comunicação directa entre os clientes e apenas o administrador tem acesso aos recursos partilhados.

2.3.3 BOINC

O BOINC [35] é uma plataforma de computação distribuída desenvolvida em Berkeley. Ultrapassou o seu projecto original, o SETI@*home*, e incorpora agora um vasto número de projectos relacionados. O seu funcionamento assenta na noção de unidades de trabalho mas não é flexível. Todas as unidades de trabalho são definidas como tendo o mesmo custo computacional e de largura de banda, determinada em cada projecto. É impossível comparar antecipadamente, por exemplo, se servir um bloco do SETI@*home* ou do Folding@*home* tem maior custo de CPU ou de largura de banda. Para além disso, os utilizadores não podem submeter as suas próprias unidades de trabalho sem terem desenvolvido um cliente BOINC totalmente apto e executado o seu servidor BOINC.

Arquitectura: A arquitectura do BOINC é simples e foi inspirada no seu predecessor. No lado servidor, existe um banco de dados relacional, que armazena diversas informações referentes a um projecto, como utilizadores registados, unidades de trabalho disponíveis, enviadas e processadas, etc. Cada projecto possui também um *back-end*, responsável por distribuir as unidades de trabalho e tratar os resultados recebidos. Os servidores de dados são responsáveis pela distribuição dos ficheiros de dados e pela recolha de ficheiros de saída, quando presentes. Os escalonadores controlam o fluxo de entrega das unidades de trabalho aos clientes conforme a produtividade de cada um. Finalmente, são disponibilizadas interfaces Web para a interacção com os programadores e utilizadores. O lado do cliente é composto pelo núcleo, que se mantém comum como fundação do sistema, e pelo código cliente específico de um determinado projecto.

Segurança: Uma vez que o cliente pode executar diversas aplicações, ao contrário do SETI@*home*, mecanismos de segurança tornam-se necessários. Por exemplo, para impedir falsificação de resultados, o BOINC utiliza redundância para diminuir a probabilidade de ocorrência desse problema: cada unidade de trabalho é distribuída para vários clientes e os resultados são verificados em busca de eventuais discrepâncias. Para eliminar a distribuição forjada de aplicações, ou seja, a possibilidade de um atacante distribuir um código cliente como se pertencesse a um projecto no qual o utilizador participa, o BOINC usa assinatura de código, isto é, cada projecto possui um par de chaves criptográficas que são usadas para autenticar os programas que distribui. Finalmente, para impedir ataques de DoS (*Denial of Services*) ao servidor de dados, nos quais um mal-intencionado envia ficheiros de saída gigantescos de forma a ocupar todo o disco do servidor, o BOINC possibilita que os programadores da aplicação especifiquem um tamanho máximo esperado dos ficheiros de saída, impedindo

assim tal ataque. No entanto, o BOINC não toma medidas para impedir que uma aplicação distribuída por um projecto cause danos intencionais ou acidentais ao sistema dos utilizadores: não há nenhum tipo de protecção do estilo *sandbox* que limite as acções de aplicações pertencentes a um projecto, o que implica que o participante confie plenamente nos responsáveis por um projecto.

Assim, tal como no SETI@*home*, o BOINC não se preocupa apenas na construção de uma infra-estrutura para a computação distribuída, mas também na criação de características que atraiam utilizadores aos eventuais projectos que utilizarão tal estrutura. Assim, o BOINC oferece a possibilidade aos programadores de projectos de gerar gráficos em *OpenGL* que serão apresentados aos utilizadores fornecedores de recursos, servindo como um atractivo.

2.3.4 Cluster Computing on the Fly

O *Cluster Computing on the Fly* (CCOF) [36] é um projecto que procura utilizar os ciclos de processamento de utilizadores com acesso disponível, em ambientes não institucionais. O sistema de partilha de ciclos do CCOF engloba todas as actividades envolvidas na gestão de ciclos ociosos: construção de uma *overlay* para os participantes oferecerem os seus ciclos, descoberta de recursos na *overlay* e os diferentes tipos de escalonamento.

O CCOF suporta o escalonamento automático de diversas aplicações num modelo distribuído de forma a permitir a qualquer nó a partilha ou o consumo de ciclos ociosos. Assume também justiça entre os nós, garantindo que todos terão acesso ao sistema de forma justa, sem necessitar de monitorização dos ciclos oferecidos e consumidos.

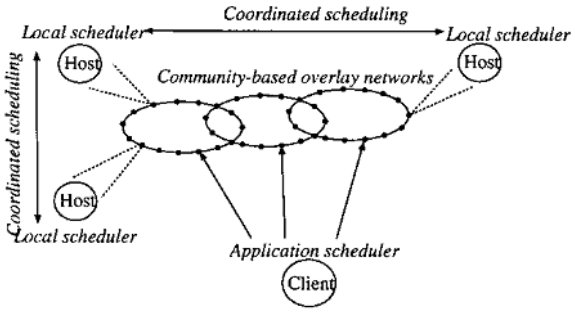

Figura 11: Arquitectura do CCOF [36]

Arquitectura**:** Na arquitectura deste sistema, os clientes associam-se a uma variedade de comunidades em redes *overlay*, dependendo da forma como gostariam de oferecer os seus ciclos ociosos. Em seguida, os clientes formam um grupo de computação através da descoberta e escalonamento de um conjunto de máquinas das redes *overlays*. Os componentes da arquitectura, como mostra a Figura 12, consistem: gestor da comunidade – organização das comunidades através da criação de redes *overlay* de acordo com certos factores comuns, como o interesse, a localização geográfica, o desempenho, a confiança ou as afiliações institucionais; descoberta de recursos (ver a seguir) [37]; escalonador da aplicação – é responsável pela selecção de um conjunto de nós para a realização de computação P2P,

negociação do acesso, exportação da aplicação e recolha e verificação dos resultados; escalonador local – rastreia ciclos ociosos e negoceia com o escalonador da aplicação as tarefas a executar, utilizando critérios locais no que diz respeito ao nível de confiança, justiça e desempenho; mecanismos e políticas de incentivos e justiça; mecanismos de reputação e confiança (ver a seguir); mecanismos de segurança – a execução local de código externo é feita através de uma máquina virtual que cria uma "*sandbox*" para a execução, protegendo o utilizador e controlando a utilização dos recursos. Para evitar o uso malicioso dos recursos, os nós terão que negar o acesso a clientes que não são de confiança, de acordo com os sistemas de confiança e reputação; monitorização do desempenho.

Wave Scheduling**:** O *Wave Scheduling* tem como objectivo capturar ciclos de computadores que durante a noite estão desocupados. Seguindo os diferentes fusos horários, é possível delegar tarefas sem que haja interrupções, pelos utilizadores, das máquinas. Para organizar os nós dos diferentes fusos horários, é utilizado o DHT *overlay* baseado no CAN [15] (Figura 13):

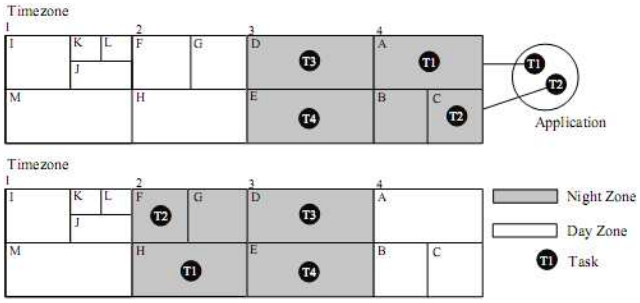

Figura 12: Iniciação e migração de uma tarefa em *Wave Scheduling* [36]

É seleccionada uma dimensão da rede d-dimensional para representar os fusos horários, por exemplo, 1x24 espaço do CAN pode ser dividido em zonas de 1 até 23 horas. O escalonador da aplicação escolhe então um fuso horário nocturno inicial (de acordo com um critério de selecção) e decide o número de nós alvo superior ao necessário para obter mais flexibilidade. Estes nós alvo são retirados aleatoriamente duma lista de nós na zona de fuso horário nocturno e é-lhes enviada uma mensagem de pedido utilizando o encaminhamento do CAN.

Após as negociações dos recursos, o escalonador escolhe um subconjunto desses nós para receberem as tarefas. Quando amanhece na zona de um nó, é escolhida outra zona e a tarefa é transferida para um novo nó nessa zona. Quando uma tarefa termina, os resultados são enviados directamente para a aplicação. Se a aplicação não estiver a ser executada, os resultados são guardados no sistema de ficheiros distribuídos do CAN e posteriormente recolhidos através de uma procura baseada em DHT.

Reputação e Confiança: A validação dos resultados recolhidos é feita utilizando um mecanismo de problemas a resolver. A aplicação do nó envia um conjunto de problemas cujas soluções são já conhecidas. Dependendo do desempenho dos nós na resolução dos problemas, a aplicação decide se os resultados desse nó serão aceites ou rejeitados. São usados 2 métodos para a realização deste mecanismo. Um método utiliza problemas que são distintos do código

da actual aplicação. Estes são encapsulados para que os nós (maliciosos) não possam distinguir o que são os problemas e o que é o código da aplicação. Se o nó resolver o problema, ser-lhe-á enviada outra tarefa, esta contendo o código da aplicação. O segundo método insere pequenos problemas no código da aplicação. As resoluções e resultados da aplicação são periodicamente enviados para a aplicação. Se a aplicação receber respostas erradas, a tarefa é de imediato reencaminhada para outro nó. Em ambos os métodos, quando um nó envia resultados correctos (ou incorrectos), estes podem ser usados para actualizar o nível de confiança de um nó.

Descoberta de recursos: A descoberta de recursos é uma das funcionalidades mais delicadas de implementar num sistema de partilha de ciclos de processamento devido às constantes entradas e saídas dos nós participantes. O CCOF serviu de simulador para comparar quatro métodos de procura de recursos, *expanding ring search*, *advertisement based search*, *random walk* e *rendezvous point*, sendo este último, o que revelou maior desempenho global em situações de grande e pouca carga computacional, graças ao seu constante e reduzido tráfego de mensagens.

Rendezvous Point – este método utiliza um grupo dinâmico de nós classificados como 'pontos de encontro' no sistema, com o objectivo de oferecer eficiência nos pedidos e na recolha de informação. Os nós anunciam os seus perfis aos 'pontos de encontro' mais próximos e os clientes contactam os 'pontos de encontro' para obterem a localização de nós com recursos disponíveis. Quando um nó é seleccionado como 'ponto de encontro', este informa a sua nova função a todos os nós a que está ligado dentro de um alcance limitado.

Estratégias de escalonamento: Depois de o escalonador da aplicação obter um conjunto de nós candidatos para a realização das suas tarefas, este escolhe os nós alvo de acordo com um critério de selecção como o nível de confiança desse nó, classificação do desempenho, etc. Quando um cliente não consegue encontrar recursos suficientes para poder iniciar a computação, o escalonador da aplicação pode desistir de procurar ou tentar agendar a tarefa para a noite, onde poderão existir mais recursos disponíveis. Após a descoberta dos recursos necessários para iniciar a tarefa, estes são reservados durante um período de tempo de forma a evitar a competição e migração das tarefas de outros nós.

O CCOF é um projecto que partilha os mesmos objectivos de implementação de uma infra-estrutura de *grid peer-to-peer* genérica da solução apresentada, apesar da diferença em alguns aspectos do desenho. Contudo, este projecto procurou evidenciar aspectos como a descoberta de recursos e a detecção de resultados incorrectos.

2.3.5 Mapeamento de sistemas Desktop Grid

O mapeamento de sistemas e projectos *Desktop Grid* são ilustrados na seguinte tabela:

Sistema	Organização	Escala	Fornecedor	Tecnologias
Condor	Centralizado	LAN	Institucional	*Grid*
SETI@home	Centralizado	Internet	Voluntário	*Grid* + P2P

BOINC	Centralizado	Internet	Voluntário	*Grid* + P2P
XtremWeb	Centralizado	Internet	Voluntário	P2P
P3 [38]	Parcial (nós *manager*)	Internet	Voluntário	P2P
CCOF	Distribuído	Internet	Voluntário	*Grid* + P2P

Tabela 3: Sistemas Desktop Grids

2.4 ANÁLISE E COMPARAÇÃO

Recentemente duas abordagens de computação distribuída têm sido desenvolvidas no sentido de colmatar os problemas de organização das sociedades de computação em grande escala: *peer-to-peer* e computação em *Grid*. Ambas as tecnologias parecem ter o mesmo objectivo final: a utilização e coordenação de um largo conjunto de recursos distribuídos; mas baseando-se em distintas comunidades e dando prioridade a diferentes requisitos. No entanto, estes dois sistemas parecem convergir, no que diz respeito às suas preocupações, na medida em que, as *Grids* se tornam mais escaláveis e os sistemas P2P se interessam por requisitos mais sofisticados [39].

Já as *Desktop Grids* têm vindo obter grande notoriedade no meio da computação em *Grid*, tendo-se tornado na forte atracção para a execução de aplicações de grande processamento, à medida que as CPUs, unidades de armazenamento e as capacidades de rede têm vindo a melhorar e a ficar mais baratas.

Comparação entre P2P e Grids: Estes dois sistemas seguiram dois caminhos evolutivos diferentes. Deste modo, as Grids possuem aplicações e serviços sofisticados que permitem interligar um pequeno número de sites com o objectivo de colaborar em aplicações científicas complexas. Mas, à medida que a escalabilidade destes sistemas tem vindo a aumentar, os programadores de aplicações em Grid enfrentam problemas relacionados com configuração e gestão automatizada desses sistemas. Por sua vez, as comunidades P2P têm-se desenvolvido essencialmente à volta de serviços pouco sofisticados, mas populares, como a partilha de ficheiros, e procuram agora a expansão para aplicações mais sofisticadas, e a continuação da inovação nas áreas de gestão de sistemas automatizados de grandes escalas. Como resultado destes diferentes percursos evolucionários, os dois sistemas diferem principalmente em cinco aspectos: comunidades alvo, recursos, aplicações, escalabilidade, e serviços e infra-estrutura, que serão abordados seguidamente.

Comunidades alvo: Apesar das tecnologias em Grid terem sido inicialmente desenvolvidas para resolver necessidades no ramo científico, o interesse comercial está a crescer. Os participantes neste tipo de sistemas formam parte de comunidades preparadas para dedicarem os seus esforços na criação e funcionamento da infra-estrutura necessária, dentro da qual existem níveis de confiança, responsabilidade e oportunidade, e a aplicação de penalizações como consequência de comportamentos inapropriados.

Em contraste, as tecnologias P2P têm-se destacado na popularização de partilha de ficheiros e de aplicações de computação paralela em culturas de massa [40], que chegam a atingir a participação de centenas de milhares de nós. Mas as comunidades constituídas por estas aplicações englobam

utilizadores diversificados e anónimos com pouco incentivo à cooperação. Daí que seja possível encontrar aplicações de partilha de ficheiros onde existem poucos fornecedores de conteúdos e muitos consumidores [41], o que justifica que os operadores do SETI@home [40] dediquem um esforço significativo na detecção de resultados incorrectos produzidos deliberadamente. Por isso, estes sistemas devem fornecer mecanismos de incentivos e de autoridade.

Recursos: No geral, os sistemas em Grid integram recursos que são mais potentes, mais diversificados e com melhor interligação do que os recursos típicos em P2P. A existência de uma administração explícita em Grids, permite disponibilizar recursos capazes de garantirem a qualidade dos serviços e a fácil manutenção e actualização a nível de software. No entanto, pode levar a um aumento dos custos de integração dos recursos numa Grid.

Os sistemas P2P lidam normalmente com populações muito dinâmicas, isto é, constantes entradas e saídas de nós, pelo que apresentam uma grande variação dos recursos disponibilizados. Este tipo de sistemas deve providenciar mecanismos de descoberta de recursos que se ajustem ao dinamismo dos participantes. Já nas Grids, os recursos são agregados dentro de cada domínio administrativo, através de tecnologias como o Condor [14][42], que permite criar um conjunto local de recursos que vão ser integrados em Grids maiores como outros recursos computacionais.

Aplicações: Os sistemas P2P costumam apresentar soluções especializadas em problemas de partilha de recursos, uma vez que grande parte das suas funcionalidades agrega a partilha de ficheiros ou de ciclos de processamento. A diversificação destes sistemas vem dos diferentes objectivos de desenho, como a escalabilidade [43][3][44], anonimato ou disponibilidade.

Escalabilidade: A escalabilidade pode ser medida em termos de duas dimensões: o número de entidades participantes e a quantidade de actividades. As comunidades científicas participantes nas Grids englobam normalmente um número modesto de participantes, quer sejam instituições (dezenas), computadores (milhares) ou utilizadores em simultâneo (centenas). Já a quantidade de actividades pode ser enorme. Uma consequência das características destas comunidades reflecte-se em antigas implementações em Grid que não davam prioridade à escalabilidade e à auto-gestão. Desta forma, enquanto o desenho de protocolos de Grid (como os instanciados no Globus Toolkit [5]) não preconiza a escalabilidade, implementações actuais aplicam componentes centralizados, como repositórios centrais de dados, componentes de gestão de recursos centralizados (Condor Matchmaker [32]) e directórios de informação centralizados.

No caso dos sistemas P2P, existem muitas comunidades que são constituídas por milhões de nós em simultâneo. A quantidade de actividades também é significativa mas não chega a ultrapassar as quantidades verificadas em Grids de pequenas dimensões. Esta grande escalabilidade resulta da evolução dos sistemas com estruturas centralizadas até sistemas totalmente descentralizados e da capacidade de auto-organização de um grande número de nós.

Serviços e Infra-estrutura: Nas comunidades de Grids é possível encontrar aspectos técnicos e organizacionais associados com a criação e operação de serviços de autenticação [42], autorização, procura [10], acesso a recursos, entre outros. No entanto, são encontrados menos

aspectos relacionados com a gestão dos participantes na ausência de confiança, responsabilidade e reputação entre os nós. Outro aspecto dos sistemas de Grids prende-se à uniformização de protocolos e interfaces que permitam a interoperabilidade entre diferentes sistemas de Grid. Um exemplo é a arquitectura de serviços do Open Grid (OGS) [45] que integra tecnologias de Grid e de Web Services.

Os sistemas P2P focam-se na integração de recursos simples (computadores pessoais) através de protocolos criados com o objectivo de fornecer essa funcionalidade específica de integração. Assim, por exemplo, o Gnutella [6] define os seus próprios protocolos de procura e manutenção da rede. Tais protocolos definem a infra-estrutura.

Com o tempo, emergiu a necessidade de novos serviços como: o anonimato e resistência à censura, mecanismos de incentivo à partilha justa e de reputação e verificação de resultados. Estes aspectos não são necessários na computação em Grid devido às assumpções de confiança entre os participantes.

Convergência: Tanto as *Grids* como o P2P partilham da mesma visão: um computador virtual a nível global onde o acesso a recursos e a serviços pode ser obtido em qualquer lugar e a qualquer hora quando necessários. Para tal, é essencial combinar as forças de cada um: atacar o problema de falhas utilizando sistemas escaláveis e protocolos de auto-configuração; fornecer uma infra-estrutura com um suporte organizado e distribuído, capaz de alcançar robustez, confiança e desempenho, e fornecer serviços altamente disponíveis com múltiplas finalidades (p.e. o mesmo serviço de monitorização e descoberta ser utilizado por várias funcionalidades de alto nível, como escalonamento, replicação e detecção de falhas).

Comparação entre Grid e Desktop Grids: A *Desktop Grid* tem como objectivo a recolha de recursos ociosos de *desktops* de utilizadores comuns na internet [35][46][47]. Uma *Desktop Grid* é como uma *Grid*, mas com algumas diferenças em termo da tipologia e características dos recursos e dos tipos de partilha (ver tabela 4).

Aspectos	Desktop Grid (DG)		Grid
	Internet-based	*LAN-based*	
Recursos	Fornecedor anónimo	Instituição, Universidade, etc.	Super-computador, *cluster*, base de dados, etc.
Ligação	-Baixo débito -Ligação activa -Ter em conta firewall e NAT	-Débito intermédio -Ligações mais constantes	Alto débito
Heterogeneidade	Alta – necessita de agrupamento de recursos	Intermédia	Pouco heterogénea
Dedicação	-Não dedicado -Muito volátil – necessita mecanismos de incentivo	-Não dedicado -Pouco volátil - necessita mecanismos de	Dedicado - possibilidade de alocação

		incentivo	
Confiança	Necessita certificação de resultados	Pouca confiança nos fornecedores	Alta confiança nos fornecedores
Falha	Inseguro (faltoso)	Inseguro	Mais seguro que DG
Gestão	-Totalmente distribuída -Difícil de gerir	Mais controlável que *Internet-based*	Administrador do domínio
Aplicação	-Independente -Computação intensiva -Grande carga	-Independente -Computação intensiva -Grande carga	-Dependente ou independente -Computação intensiva

Tabela 4: Comparação entre *Grid* e *Desktop Grid*

A *Desktop Grid* tem vindo a alcançar maior interesse e atracção devido ao sucesso de vários exemplos populares como o SETI@*home* [48] e a *Distributed.net* [49]. Têm-se também desenvolvido alguns estudos em sistemas de *Desktop Grids* que fornecem uma plataforma subjacente, como o BOINC [35], *XtremWeb* [46], *Entropia* [47] ou o *Condor* [50].

2.5 CONCLUSÃO

Neste capítulo foi apresentado o estado da arte das tecnologias *Peer-to-Peer*, *Grids* Institucionais e sistemas de *Cycle Sharing*/*Desktop Grids*, fundamental para enquadramento teórico do projecto a desenvolver. Assim, foram apresentados e analisados (secção 5) diversos sistemas, tendo sido reforçada a ideia inicial da necessidade de convergência entre sistemas de *Grids* e P2P, uma vez que foi possível concluir que ambos: se preocupam com a organização da partilha de recursos; abordam o problema com a criação de topologias *overlay* sobre uma estrutura subjacente, e cada um possui problemas e soluções que se complementam, p.e., o problema comum da escalabilidade presente nos sistemas em *Grids* é secundário ou inexistente nos sistemas P2P.

No que concerne à solução do projecto, foi apresentada uma descrição geral de um modelo de programação baseado no conceito de *gridlets*, capaz de estabelecer uma ligação entre infra-estruturas de *Grids* Institucionais, aplicações populares de partilha de ciclos e aplicações P2P descentralizadas de partilha de ficheiros, levando a tecnologia das *Grids* até aos utilizadores comuns. Ao contrário de abordagens anteriores, esta solução possibilita, de forma transparente, a exploração do paralelismo da execução de aplicações populares para melhorar o desempenho, sem a necessidade de modificações às mesmas, utilizando os ciclos de processamento de máquinas ociosas numa rede.

3. Arquitectura

Neste capítulo são discutidos os objectivos do desenho e os requisitos do sistema apresentado. Inicialmente, é dada uma visão geral sobre o projecto GINGER que originou o desenvolvimento deste sistema, sendo em seguida representada e explicada a arquitectura adoptada para o sistema, bem como os seus componentes e funcionalidades.

3.1 Projecto GINGER

Como já foi anteriormente mencionado, existem vários projectos que possibilitam a utilização e partilha de recursos semelhante às infra-estruturas de *Grid* Institucionais, de uma forma *peer-to-peer*. No entanto, nenhum destes projectos implementa uma infra-estrutura *Grid peer-to-peer* genérica, isto é, um sistema capaz de permitir, em grande escala, a utilizadores comuns, a execução das suas aplicações aproveitando os recursos livres de outras máquinas.

O projecto GINGER, ou GiGi, tem como objectivo a síntese de três tecnologias: infra-estruturas de *Grid* Institucionais, arquitecturas *peer-to-peer* descentralizadas e partilha distribuída de ciclos excedentários. O projecto desenvolve uma plataforma genérica através da criação de um substrato para uma infra-estrutura *Grid peer-to-peer*, possibilitando a utilização por qualquer pessoa às tecnologias *Grid*.

O modelo de programação deste projecto baseia-se no conceito de uma unidade básica de trabalho divisível em pequenas tarefas, consciente da semântica dos seus dados – uma *Gridlet*. O principal objectivo é a utilização deste conceito em aplicações genéricas, como por exemplo, um codificador de vídeo, sem que sejam necessárias modificações das mesmas. A aplicação não é obrigada a estar consciente da plataforma subjacente. Os dados da aplicação a serem processados são divididos em pequenas tarefas e enviadas para a plataforma, que criará, a partir desses dados, as *Gridlets*. As *Gridlets* transportam os dados da aplicação a serem processados e um custo estimado da computação sobre esses dados.

O funcionamento global do projecto inicia-se com a recepção dos dados a serem processados, onde vão ser divididos em pequenas tarefas e encapsulados em *Gridlets.* Em seguida, são descobertos recursos em vários nós disponíveis na rede, para onde serão encaminhadas as *Gridlets*. Quando os dados estiverem processados, estes são recolhidos e agregados, enviando o resultado final para a aplicação. Durante todo este processo, a aplicação não tem percepção das fases envolventes no processo, a não ser do fornecimento dos dados e da recolha do resultado da computação sobre os mesmos.

3.1.1 Arquitectura do GINGER

A plataforma do *GiGi* corre em cada nó pertencente à *grid-overlay* e segue uma arquitectura estruturada em camadas, de forma a favorecer a portabilidade e extensibilidade. Esta arquitectura está ilustrada na Figura 13, onde se verifica as interacções ao nível dos nós e da rede. A plataforma presente no nó serve de *middleware* entre a aplicação e o *overlay*. As

Gridlets, retratadas na figura como pequenos quadrados, são submetidas pelos nós e transmitidas pelo *overlay* P2P, onde serão recebidas e processadas por nós ociosos. Após a computação dos dados, os resultados são devolvidos aos nós que submeteram os pedidos sob a forma de *Gridlet-results*.

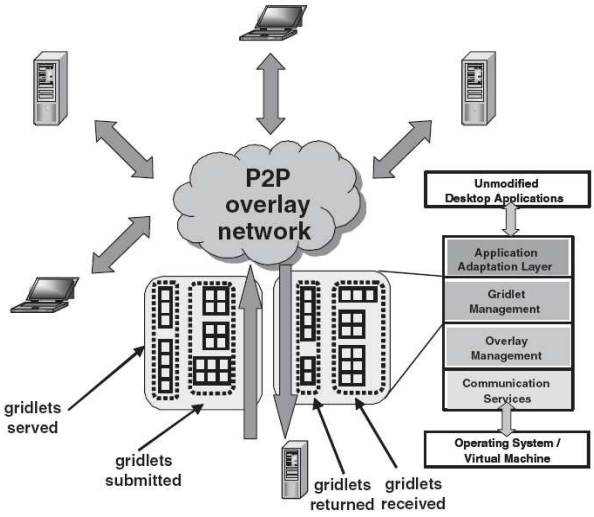

Figura 13: Arquitectura do *GiGi* [51]

As camadas da plataforma, responsáveis pela gestão e criação de *Gridlets* e gestão dos serviços do *overlay*, são a seguir referidas:

- Camada de *Application Adaptation* – é responsável pela interacção com as aplicações *desktop* originais, por exemplo, executá-las, fornecer os dados das *Gridlets* e recolher os resultados.
- Camada de *Gridlet Management* – efectua as operações necessárias para a partição dos ficheiros em *Gridlets* e é responsável pela agregação dos *Gridlet-results*. Todos os processos ou alterações ao nível das *Gridlets* são efectuados nesta camada;
- Camada de *Overlay Management* – é responsável pela manutenção da rede *overlay*, como a descoberta de nós, o encaminhamento de mensagens, endereçamento, gestão dos recursos do nó local e a manutenção dos recursos anunciados por nós vizinhos;
- Camada de *Communication Services* – efectua as comunicações e transferências na rede;

No *GiGi*, os nós estão organizados num *overlay* P2P DHT, como por exemplo, o *Pastry* [4]. Desta forma, a descoberta de recursos será facilitada com o encaminhamento pelas ligações do *overlay*, que podem não corresponder às ligações físicas existentes, e são herdadas importantes propriedades como o ajustamento automático ao dinamismo existente nas redes P2P ou a compreensão de proximidade de localização, de acordo com uma métrica escalar de proximidade.

3.2 Objectivos do Desenho e Requisitos

No contexto do projecto GINGER, este trabalho emerge da necessidade da criação de uma plataforma, capaz de unir infra-estruturas *Grid* com aplicações P2P e de partilha de recursos, explorando diferentes topologias da rede P2P que maximizem determinadas métricas de desempenho dos sistemas (por exemplo, consumo de largura de banda ou tempo para a execução de uma tarefa) para a escolha de vizinhos na rede. A importância da topologia da rede deriva do facto de os mecanismos de descoberta de recursos, no *overlay* P2P, seguirem as ligações formadas pela topologia da rede e não as ligações físicas. Desta forma, o sistema criado terá como principal objectivo, o correcto encaminhamento dos pedidos para os melhores nós, tendo em conta o custo associado aos pedidos e os vários critérios que definem a melhor escolha, como a largura de banda da ligação ou os recursos disponíveis no nó.

O sistema é uma plataforma de *middleware* sobre uma rede *overlay* P2P estruturada que baseia o seu funcionamento à volta de uma *Gridlet.* Uma *Gridlet* é um fragmento de dados, capaz de descrever todos os aspectos de uma tarefa de trabalho, bem como as transformações necessárias aos dados das tarefas. Quando um nó submete um trabalho para execução, este é repartido em pequenas tarefas e são geradas as *Gridlets*, que vão ser encaminhadas pelo *overlay* P2P para outros nós onde vão ser processadas. Mais tarde, os resultados, *Gridlet-results,* podem ser enviados para os nós iniciais ou ficam disponíveis no *overlay*.

3.2.1 Overlay

A solução pretendida requer a construção de um *overlay peer-to-peer* robusto. O *Pastry* [4] é uma rede *overlay* e de encaminhamento *peer-to-peer* genérica, escalável e eficiente. A auto-organização dos nós e o *overlay* estruturado sobre a Internet, totalmente descentralizado e tolerante a faltas do *Pastry*, representa uma boa estrutura *overlay* para a solução proposta.

3.2.2 Submissão e tratamento de pedidos

Cada nó poderá submeter pedidos sob a forma de *Gridlets*. Essas *Gridlets* serão compostas com os dados necessários para a computação da tarefa e com o custo necessário para a mesma. Uma vez que os objectivos deste sistema não abordam as interacções com a aplicação *desktop*, nem a divisão das tarefas ou processamento das mesmas, o conteúdo dos dados que as *Gridlets* transportam não são importantes. Desta forma, o custo da computação de uma *Gridlet* é pré-definido na sua criação e o processamento de uma *Gridlet* deverá apenas incutir a redução nos recursos locais e consumir o tempo de processamento, que é calculado, por opção, de acordo com o custo que lhe foi atribuído e o tamanho dos dados que transporta.

3.2.3 Descoberta de Recursos

Para acedermos aos recursos partilhados por outras máquinas ligadas ao *overlay*, é necessária a implementação de um mecanismo de descoberta de recursos, capaz de encontrar de forma eficaz esses recursos. A solução abordada consiste na descoberta e observação das capacidades de um número restrito de nós na rede, por exemplo, o conjunto de vizinhança disponibilizado pelo *Pastry*. Cada nó anunciará os seus recursos, através do envio de mensagens do tipo *update*, apenas aos nós que compõem o seu conjunto de vizinhança.

Quando um nó submeter pedidos, verifica as informações que dispõe sobre os seus vizinhos e encaminha a *Gridlet* para o nó que achar mais conveniente. O processo de selecção do melhor nó para encaminhar os pedidos é um processo delicado e essencial para uma eficaz e eficiente resolução das tarefas, quer por parte do nó que as submete, quer pelos restantes nós da rede.

3.2.4 Recolha dos resultados

Existem várias soluções para a forma de recolher os resultados das *Gridlets*, as *Gridlet-results*, processadas pela rede. Foram consideradas e estudadas três soluções:

1) Uma solução simples consiste na inserção do ID do remetente na *Gridlet* correspondente à tarefa a processar, permitindo o encaminhamento directo dos resultados para o remetente. Apesar da vantagem de permitir a entrega imediata do resultado após a sua conclusão, esta solução resultaria na perda de privacidade dos pedidos.

2) Outra solução capaz de garantir alguma privacidade, consiste em enviar a *Gridlet-result* pelo caminho de volta entre os nós que a *Gridlet* original percorreu. Este método pode trazer vantagens em termos de *caching*. No entanto, é pouco eficaz para casos onde o número de *hops* não seja baixo, resultando daí várias desvantagens como o aumento do tempo de envio em relação ao envio directo, o aumento do *overhead* de mensagens na rede, a sobrecarga dos nós desse trajecto devido aos estados que estes teriam de manter para as diferentes *Gridlets* e no caso de saídas ou falhas de nós, o caminho de volta ficaria inviabilizado.

3) Por último, o envio do resultado processado para uma cache de ficheiros em P2P (ex: PAST) é a solução mais flexível e garante a privacidade. Para efeitos de *caching*, os resultados estarão já guardados na cache. No entanto, apresenta algumas desvantagens, na medida em que, perde tempo com o envio dos resultados para a cache e, adicionalmente, implica a previsão de um tempo ideal, por parte do nó remetente, para activar um mecanismo de recolha dos resultados da cache.

A solução 3) adequa-se melhor aos objectivos do projecto, dando-se assim primazia à flexibilidade e privacidade, em detrimento do tempo de recolha das *Gridlet-results*.

3.3 Arquitectura do Sistema

Analogamente à arquitectura do *GiGi,* a arquitectura da aplicação do *GiGi* proposta é também estruturada em camadas. O ambiente de execução deste sistema é controlado por um componente adicional, o *GiGiSimulator*, responsável pela criação e monitorização do *overlay*. O sistema proposto engloba a aplicação do *GiGi*, uma rede *overlay* e um simulador. A interacção entre cada componente determina o funcionamento do sistema.

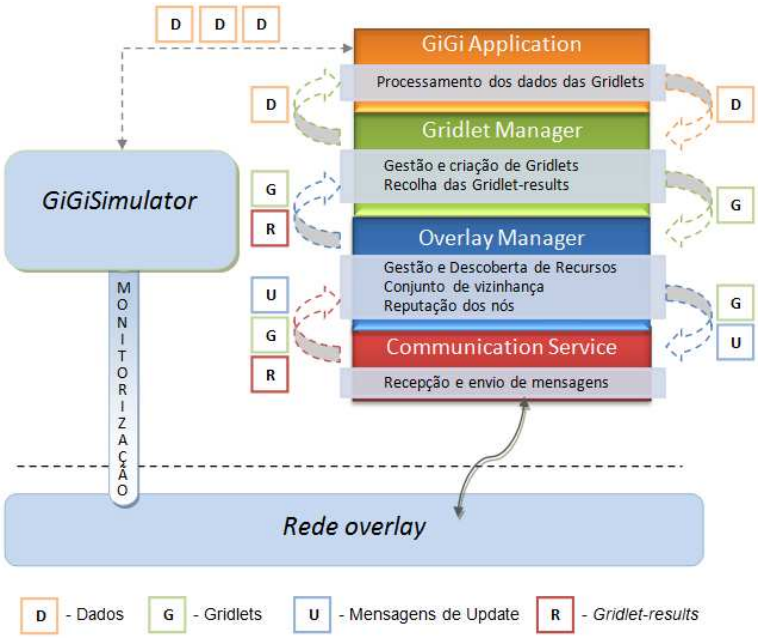

Figura 14: Arquitectura do sistema

A Figura 14 apresenta a arquitectura do sistema, com os vários componentes que o constituem, e respectivas interacções, descrevendo as principais funcionalidades de cada um. O *GiGiSimulator* é responsável pela simulação do sistema, criando uma rede *overlay*, com um número de nós parametrizável, e uma aplicação do *GiGi,* que engloba as quatro camadas representadas, para cada nó da rede. Cada camada da aplicação interage com a camada imediatamente superior e inferior.

Uma vez que os objectivos deste trabalho se prendem apenas aos aspectos da topologia da rede, a agregação de *Gridlet-results* e a interacção com aplicações *desktop* não foram consideradas.

3.4 Especificação da Arquitectura

Nesta secção será analisada detalhadamente a arquitectura do sistema, desde o seu funcionamento geral até às diversas camadas que a constituem e respectivas interacções. Serão também descritos o componente responsável pela criação do *overlay*, o *GiGiSimulator*, o próprio *overlay*, as mensagens transmitidas pela rede e o mecanismo de gestão e de descoberta dos recursos.

3.4.1 GiGiSimulator

Este componente simula o funcionamento de toda a rede e o sistema, servindo de suporte à rede e possibilitando a interacção com a aplicação desenvolvida do *GiGi*, através de uma interface de comandos. O seu papel estende-se desde a geração de eventos sobre a rede e sobre a aplicação que corre nos nós, até à monitorização das mensagens transmitidas entre os nós e recolha de dados estatísticos referentes às actividades na rede.

Assim, a formação da rede é iniciada no simulador com a criação do primeiro nó, sendo que os nós seguintes se inserem no mesmo *overlay*. Após a formação da rede, são lançadas as aplicações do *GiGi* que serão associadas a cada nó do *overlay*. Através de uma interface de linha de comandos no *GiGiSimulator*, é possível gerar eventos na rede ou nas aplicações, como por exemplo a entrada ou saída de nós na rede ou a submissão de pedidos pelas aplicações do *GiGi*. Todo o tráfego transmitido pela rede pode ser monitorizado no simulador.

3.4.2 GiGi Application

A camada *GiGi Application* é baseada na camada de *Application Adaptation* da arquitectura do projecto GINGER. No entanto, as funcionalidades inerentes à camada do GINGER não são herdadas neste sistema, pelo que a camada *GiGi Application* tem um papel meramente representativo na estrutura da arquitectura: transmite os dados recebidos do simulador à camada do *Gridlet Manager* e recebe os dados das *Gridlet-results* provenientes do *Gridlet Manager*, notificando o simulador da conclusão do pedido.

Para além disso, é nesta camada que a computação sobre os dados das *Gridlets* é simulada, através do consumo do tempo necessário, de acordo com o custo associado à *Gridlet* e o tamanho dos seus dados. Nenhuma computação é efectivamente aplicada aos dados.

3.4.3 Gridlet Manager

O *Gridlet Manager* trata de todas as operações a serem efectuadas com *Gridlets*. É nesta camada que são criadas as *Gridlets* a partir dos dados recebidos da camada anterior. Por sua vez, estas, após a sua elaboração, são enviadas para a camada inferior, o *Overlay Manager*.

Todas as mensagens do tipo *Gridlet* recebidas da rede são automaticamente encaminhadas para esta camada onde serão analisadas. De acordo com a disponibilidade actual do nó, uma *Gridlet* poderá vir a ser processada na camada do *GiGi Application* ou enviada de volta para o *Overlay Manager* para ser reencaminhada para outro nó na rede.

Esta camada é também responsável pela recolha dos resultados das *Gridlets* submetidas. Como não existe uma entidade que simula a aplicação *desktop*, os resultados recolhidos são apenas enviados à camada do *GiGi Application*, que notifica o simulador da conclusão do pedido.

3.4.4 Overlay Manager

O *Overlay Manager* é responsável pelas operações de encaminhamento e endereçamento na rede *overlay*. A gestão e descoberta de recursos na rede são efectuadas nesta camada. Os recursos do nó local são controlados por este componente, que realiza as operações de redução ou aumento dos recursos. Quando ocorrem alterações nos recursos, estas são anunciadas aos seus nós vizinhos através de mensagens de *update*.

O conjunto de vizinhança é criado e gerido a este nível. Os nós do conjunto são seleccionados com base na proximidade física que separa os dois nós. Para além disso, é mantida nesta

camada toda a informação relativa aos recursos disponíveis em cada nó da vizinhança, que será avaliada segundo um conjunto de critérios destinados à selecção de um nó de destino.

Quando uma *Gridlet* chega da camada de *Gridlet Manager*, o *Overlay Manager* escolhe um nó com recursos disponíveis suficientes para tratar esta *Gridlet*. Os resultados do encaminhamento de um pedido por um dado nó são assimilados numa tabela de reputação dos nós. O *Overlay Manager* recorre a esta tabela quando, no seu conjunto de vizinhança, não existem nós com recursos disponíveis que satisfaçam o custo da *Gridlet*. Consultando a tabela, é escolhido o nó que, com base em resultados anteriores, teve menos casos de insucesso e que menos atraso teve no processamento dos pedidos.

3.4.5 Communication Service

A camada do *Communication Service* trata de todas as comunicações entre a aplicação do *GiGi* e a rede *overlay*. O envio e recepção efectiva de mensagens são realizados nesta camada. Todas as mensagens recebidas da camada do *Overlay Manager* são enviadas para a rede. A rede contacta a camada de *Communication Service* quando existe uma mensagem destinada ao nó associado à aplicação. Essa mensagem é recebida e encaminhada para o *Overlay Manager*.

Para além disso, são notificadas todas as alterações, a entrada e saída de nós, que ocorram no conjunto de vizinhança desse nó. A resposta a essas alterações é realizada pela camada do *Overlay Manager*.

3.4.6 Overlay

A rede *overlay* DHT utilizada é o *Pastry*. A rede interliga os nós de acordo com a ordem dos identificadores que lhe foram atribuídos. O seu espaço de endereçamento tem a forma circular no seu espaço de identificadores, em que um nó estabelece duas ligações, uma com o nó precedente, com o identificador imediatamente menor que o seu, e o nó posterior, com o identificador imediatamente maior que o seu. A atribuição dos identificadores é feita aleatoriamente, pelo que nós adjacentes no espaço de identificadores têm fortes probabilidades de estarem geograficamente dispersos.

Este tipo de estrutura fornece um conjunto de propriedades chave para a sustentabilidade da rede: descentralização, auto-organização entre os nós sem necessidade de nenhum tipo de coordenação central ou de super-nós; escalabilidade, a rede deverá funcionar correctamente para grandes números de nós; tolerância a faltas, a rede será fiável mesmo com a constante entrada, saída e falhas de nós.

3.4.7 Tipos de Mensagens

Neste sistema existem dois tipos principais de mensagens que são enviados pela rede, as *Gridlets* e as mensagens de *update*. As *Gridlets* constituem a unidade básica de trabalho dos pedidos gerados pelo simulador. Posteriormente, estas mensagens são submetidas na rede, para serem processadas por nós com disponibilidade suficiente, em função do custo que está

associado a cada *Gridlet*. Por sua vez, as mensagens de *update*, cujo objectivo passa pela divulgação dos recursos de um nó e a disponibilidade do mesmo, são transmitidas quando ocorrem alterações na disponibilidade de um nó. A mensagem contém os recursos disponíveis do nó que a enviou e, opcionalmente, a duração dessa disponibilidade (como descrito em [52]). Cada nó envia apenas informações sobre os seus próprios recursos e apenas para a sua lista de nós vizinhos.

Existem também outros dois tipos de mensagens propagados a partir da aplicação do *GiGi*, as *Gridlet-results* e as mensagens de *ContentResult*. As *Gridlet-results* são mensagens do tipo *Gridlet*, variando apenas a finalidade dos seus conteúdos. Os dados são os resultados da computação sobre os dados da *Gridlet* original e o custo associado refere-se ao custo efectivo do processamento da tarefa. As mensagens de *ContentResult* são mensagens que encapsulam as *Gridlet-results* para que estas possam ser enviadas através do sistema de cache utilizado, o PAST.

3.4.8 Mecanismos de Gestão e Descoberta de Recursos

É na camada do *Overlay Manager* que se gere e controla os recursos do nó e a informação sobre os recursos disponíveis noutros nós. Para se obter um encaminhamento eficiente é necessário ter uma ideia logo à partida do estado da disponibilidade na rede, isto é, quem são os nós com disponibilidade e que tipo e quantidade dos recursos estão disponíveis. Esta informação é propagada pela rede através das mensagens de *update*. Cada nó anuncia a sua disponibilidade para realizar tarefas de outros nós, apenas quando estes se juntam ao *overlay* ou quando ocorrem alterações na disponibilidade dos seus recursos. De forma a limitar propagações deste tipo de informação por toda a rede, gerando um enorme fluxo de mensagens e o sobre carregamento dos nós, é definido um conjunto de nós da vizinhança. As mensagens de *update* são, como tal, apenas enviadas para nós pertencentes a este conjunto. Da mesma forma, a primeira transmissão de uma *Gridlet* será para um dos nós da vizinhança.

A selecção do nó para encaminhar um pedido é efectuada com base na análise a métricas pré-definidas, que representam a disponibilidade de um nó. Para além das métricas, é também tido em conta o grau de proximidade dos nós. Sempre que existe um pedido para ser enviado, é escolhido o nó do conjunto de vizinhos capaz de suportar os custos do pedido. Se não houver nós disponíveis, ou que não dispõe de recursos suficientes para a computação do pedido, estes serão utilizados para reencaminharem os pedidos para fora do conjunto de vizinhança para outros nós que eventualmente estejam disponíveis.

Para além da informação mantida nos nós sobre a disponibilidade dos seus nós vizinhos, são também geradas medidas de desempenho sobre um dado nó, estabelecendo um valor de reputação para esse nó. Este mecanismo funciona como um histórico do encaminhamento dos pedidos através de um dado nó. O seu objectivo é ser utilizado quando não existem nós com disponibilidade para processar um pedido, fundamentando-se em resultados anteriores para escolher o nó que melhor reencaminhará o pedido para outros nós com disponibilidade.

4. Implementação

Este capítulo descreve a implementação (realização) da solução apresentada no Capítulo 3. Inicialmente é apresentada uma visão geral do *overlay* adoptado e em seguida é descrita a implementação da gestão e descoberta de recursos. Posteriormente são explicados os componentes principais do *overlay* utilizado. A implementação da estrutura da arquitectura do sistema proposto é detalhadamente explicada nos subcapítulos seguintes.

4.1 *Overlay* Adoptado

A implementação deste sistema e de toda a sua arquitectura é feita sobre o *overlay* do *Freepastry* [53]. O *Freepastry* é uma implementação *open-source* em Java do protocolo P2P do *Pastry* [4]. Esta ferramenta destina-se sobretudo à avaliação do *Pastry*, à investigação e ao desenvolvimento de topologias P2P e ao desenvolvimento de aplicações.

Os elementos principais para o funcionamento de uma rede do *Freepastry* são o componente *environment*, que define as configurações a serem utilizadas na rede, e o simulador, conduzido por eventos, onde são lançados nós que irão formar a rede. Todas as operações no *Freepastry* são executadas por uma *thread*, o *Selector Manager*, que trata de todos os *inputs* e *outputs* e eventos da rede. A utilização de uma só *thread* na simulação da rede garante a consistência dos resultados e uma sincronização mais simples do que com várias *threads*. No entanto, é importante que tarefas feitas pela *thread* do *Selector* não sejam morosas, uma vez que se trata da *thread* que gere todo o I/O da rede. Regularmente, esta *thread* troca mensagens de controlo entre os nós e verifica periodicamente possíveis falhas de nós. A execução de tarefas demoradas nesta *thread* poderia bloquear a manutenção dos nós da rede e levar a falsas detecções de falhas em nós.

Esta é umas das razões que levam o sistema proposto a ser executado num ambiente *multi-threading*. Para além disso, com várias *threads* é possível efectuar simulações mais realistas explorando o paralelismo e concorrência das execuções nos nós. Com *multi-threading* será também possível acelerar a simulação com processadores *multi-core*.

4.2 Gestão e descoberta dos recursos

A camada do *Overlay Manager* do *GiGi* é a principal responsável pela gestão e descoberta de recursos disponíveis no *overlay*. O conceito de *Gridlets*, utilizado neste sistema, não se pode simplificar à simples injecção deste tipo de mensagens na rede e esperar que estas se propaguem pelos nós até que algum se mostre disponível para processar o trabalho. Desta feita, é necessário conhecer a disponibilidade da rede e explorar a sua topologia antes de se enviarem os pedidos.

Cada nó anuncia os seus recursos, através do envio de mensagens de *update*, apenas aos nós que compõem o seu conjunto de vizinhança (4.2.1). Quando um nó tem tarefas pendentes para submissão, este verifica as informações que dispõe sobre os seus vizinhos e encaminha a *Gridlet* para o nó que achar mais conveniente (4.2.2). Se nenhum dos nós vizinhos possuir disponibilidade para a execução da tarefa, o pedido é enviado para o nó da vizinhança que melhor probabilidade

terá (baseada em envios anteriores) de reencaminhar esse pedido para outros nós com disponibilidade (4.2.3).

4.2.1 Conjunto de Vizinhança

Logo desde início, são inicializados os elementos no *Overlay Manager* cruciais para a descoberta de recursos. Na fase final da criação da rede, os nós anunciam a sua presença e os seus recursos através das mensagens de *update* e cada um constrói o seu conjunto de vizinhança. Este conjunto de nós da vizinhança é disponibilizado pelo *Pastry* e é construído com base numa métrica de proximidade entre os nós, englobando os *n* (valor variável conforme a configuração e dimensão da rede) nós mais próximos geograficamente, isto é, com os menores valores de RTT.

No entanto, podem ocorrer situações em que um dado nó tem no seu conjunto de vizinhança nós que não o vêem como vizinho. Por exemplo, para conjuntos em que *n* igual a 20, o nó A pode ver o nó B como um dos seus 20 mais próximos, e o nó B pode ter no seu conjunto 20 nós que lhe são mais próximos do que o A. Estas situações tornam-se muito frequentes para redes de grandes dimensões. Para garantir um mínimo de simetria na relação entre os vizinhos, foi definido que, utilizando o exemplo anterior, quando o nó A anunciar a sua disponibilidade ao nó B, este irá aceitá-lo como seu vizinho e adiciona-o ao seu conjunto e anuncia-lhe a sua disponibilidade. Para limitar o crescimento descontrolado dos conjuntos, foi definido como critério de escolha de um nó como vizinho apenas se este possuir um identificador numericamente menor, isto é, o B só aceitará A como seu vizinho se este possuir um identificador menor que o do B. Caso tal não acontece, o nó B devolve a mensagem de *update* ao A. O nó A, ao receber a mensagem que ele próprio enviou, apercebe-se da rejeição e excluiu o nó B do seu conjunto.

4.2.2 Selecção do melhor nó com disponibilidade

A definição de melhor nó com disponibilidade é aquele que apresenta maior disponibilidade de acordo com uma medida ponderada entre as métricas definidas (grau de proximidade, CPU, memória e largura de banda).

As métricas utilizadas, relativas aos recursos disponibilizados, contribuem, regra geral, com pesos semelhantes no cálculo ponderado da disponibilidade de um nó. Estas métricas representam os recursos disponíveis nesse nó, pelo que será preferencial a escolha de um nó capaz de satisfazer os pedidos e mesmo assim continuar com recursos disponíveis. O factor da proximidade poderá também ter grande importância sobre a escolha, na medida em que nós relativamente próximos do nó local sejam capazes de tratar os pedidos, evitando a propagação ou longas transmissões desses pedidos pela rede, restringindo assim a alocação de recursos aos nós mais próximos.

4.2.3 Selecção do melhor nó sem disponibilidade

A selecção de um nó que não está disponível para processar uma tarefa, deverá basear-se na sua capacidade de reencaminhar o pedido para outros nós capazes de a processar. O valor desta capacidade é alcançado com base no histórico de pedidos encaminhados por um dado nó que definem a reputação desse nó.

Esse histórico mantém valores relevantes do resultado de encaminhar através desse nó. Tais valores indicam o número de resultados falhados, isto é, o número de vezes em que uma *Gridlet* foi dada como falhada, sendo necessário a sua retransmissão; o número de vezes que o pedido regressou ao nó de origem; e o número de tentativas na recolha dos resultados. Estas medidas, sendo medidas de insucessos ou ineficiências, permitem definir um nível de rejeição sobre esse nó, através de um cálculo ponderado sobre as mesmas:

1) Falhas x 0.7 + Regresso à Origem x 0.25 + Tentativas x 0.05

Quanto maior for o valor resultante, maior será o nível de rejeição para este nó. Por essa razão, é dado maior peso ao número de falhas. Seria intuitivo logo à partida, excluir nós onde se tivessem verificadas muitas falhas, no entanto, tal não é aconselhável visto que se tratam de medidas baseadas em acções passadas e o próprio desempenho não depende do nó em questão, mas sim dos seus vizinhos, onde a entrada de um novo nó vizinho ou um aumento da disponibilidade na vizinhança desse nó permitem a recuperação da reputação sobre o mesmo. Em relação às restantes medidas, ambas garantem que o trabalho será correctamente processado e devolvido, tendo apenas implicações de restrição de tempo ou de excessivo número de retransmissões, no caso do Regresso à Origem, dando-se menor importância a estas.

Para além destas medidas, existe ainda uma medida que refere o valor do cálculo da reputação 1) sobre o melhor vizinho de um nó, isto é, na tabela de reputação, existe para cada nó, uma entrada com o valor da reputação do seu melhor nó. Esta medida é obtida através de um campo com esse valor enviado nas mensagens de *update* desse nó sendo utilizada para casos em que o cálculo anterior não conseguiu escolher um nó. Isto acontece quando o nó não possui informações de histórico sobre os nós.

4.3 Estrutura do *Overlay* Adoptada

O *Freepastry* é constituído por todo um tipo de componentes, capazes de criar e gerir uma *overlay* do *Pastry* com diversos nós. Para além disso, o *Freepastry* disponibiliza várias aplicações úteis que correm no seu *overlay*, como é o caso do PAST [54]. O PAST é um sistema de armazenamento distribuído de ficheiros, que corre sobre a rede *overlay* P2P do *Pastry*. Em seguida são enunciados e explicados estes constituintes do *Freepastry* que foram utilizados na solução.

4.3.1 Aplicações Utilizadas

4.3.1.1 PAST

O PAST é um sistema distribuído de armazenamento de ficheiros, construído sobre o *Pastry*. No contexto do *Freepastry*, o PAST vem inserido como uma aplicação de armazenamento P2P de ficheiros, que serve de cache para o sistema. As suas operações de *insert* e de *lookup* permitem a inserção e obtenção de ficheiros na rede, respectivamente.

Um ficheiro é guardado no sistema através de uma chave obtida na computação da *hash* do nome do ficheiro. Em seguida, o *Freepastry* encaminha os conteúdos do ficheiro para o nó que no espaço de *IDs* tem o identificador mais próximo da chave obtida anteriormente. A replicação é conseguida através do envio de cópias dos dados, por parte do nó que os recebeu, aos seus *k* nós mais próximos no espaço de *IDs* (provavelmente nós que pertencem ao seu *leafset*). Os dados são recolhidos através da chave referente aos dados, adquirida a partir da computação da *hash* do nome do ficheiro, e o encaminhamento de um pedido para o espaço de *IDs* apropriado do *overlay* (de acordo com o valor da chave).

A aplicação do PAST disponibiliza ainda diferentes tipos de armazenamento: *MemoryStorage*, uma implementação de armazenamento orientado a objectos não persistente, em memória (a chave utilizada nas operações *insert* e *lookup* é obtida através de uma *String* única para cada objecto, ao invés do nome do ficheiro); *PersistentStorage*, uma implementação de armazenamento persistente de ficheiros em disco.

4.3.2 *Common* API

A *common* API permite a utilização de várias aplicações sobre diferentes tipos de *overlays* sem que sejam necessárias alterações às suas implementações. Esta API comum define as interfaces dos vários elementos essenciais utilizados nas típicas *overlays* P2P. Desta forma, qualquer aplicação, ao utilizar esta API, é capaz de comunicar e interagir com outras aplicações através das interfaces comuns. Algumas das interfaces utilizadas da *common* API do *Freepastry* especificam a interface de uma aplicação (*Application*), de um identificador (*Id)*, de uma mensagem (*Message*) e de um nó (*Node*).

4.3.3 Principais Componentes

4.3.3.1 Simulator

O simulador do *FreePastry* é um simulador de eventos discreto, que permite criar um *overlay* do *Pastry* com diversos nós a correrem numa só máquina e monitorizar todo o seu tráfego. Este simulador oferece também diversas funcionalidades úteis para testar o desempenho do sistema criado. A sua simulação do *Pastry* é efectuada ao nível do *overlay* em vez de ao nível dos pacotes, como é realizado, por exemplo, no *ns-2 tool*. Desta forma, não se consegue obter uma contagem dos *hops* físicos, mas sim apenas dos *hops* no *overlay*. No entanto, este factor não impõe qualquer problema à simulação do nosso sistema, nomeadamente no encaminhamento de *Gridlets*, visto que não existem influências ao nível dos protocolos utilizados na rede.

O funcionamento do simulador baseia-se numa lista de prioridades constituída por tarefas. Essas tarefas estão ordenadas pelo tempo em que deverão ser executadas. Para simular o envio de uma mensagem pela rede, a tarefa é agendada para ser entregue após o tempo de atraso entre o nó remetente e o nó de destino. Em cada iteração do simulador, o primeiro elemento da lista de prioridades é verificado. O relógio virtual é incrementado caso seja necessário avançar o tempo para a execução da primeira tarefa na lista.

De uma forma geral, o simulador proporciona vantagens em vários aspectos, permitindo a utilização de aplicações do *Freepastry,* a simulação da rede de forma rápida e eficiente e a possibilidade de escolha de várias topologias de rede:

- Executa aplicações do *Freepastry* sem serem necessárias modificações ao código de origem;
- Executa o código mais rapidamente do que em tempo real. A utilização de um relógio virtual permite ao simulador progredir mais rápido no tempo e consequentemente executar mais tarefas sem que haja restrições de tempo real. Portanto, a velocidade do ambiente simulado irá depender apenas da capacidade da CPU e da memória da máquina onde é executado. No caso de simulações de redes muito grandes ou de intensa carga computacional, a eficiência não será afectada pela falta de recursos, uma vez que o relógio virtual avança no tempo apenas quando não há tarefas imediatas para serem executadas. Em casos extremos, poderá resultar em execuções mais demoradas do que seria em redes reais, mas o resultado final corresponde ao tempo simulado e não ao real;
- Aceita uma variedade de topologias para a latência da rede. O *Freepastry* disponibiliza 3 topologias: *Euclidean* (Planar), *Spherical* e *GenericNetwork*. As topologias *Euclidean* e *Spherical* utilizam equações geométricas para determinar as latências entre os nós. Na criação dos nós, são atribuídas coordenadas aleatórias que iram definir a sua localização no espaço geométrico. A latência entre dois nós é obtida através da distância entre os dois pontos que localizam os nós no espaço [55]. A topologia *GenericNetwork* obtém as latências a partir de uma matriz num ficheiro.

Contudo, o simulador não possibilita a simulação de perdas de mensagens, de largura de banda ou de variações na latência.

4.3.3.2 Continuations

À semelhança de um *callback*, ou de um *Listener* em Java, o *Freepastry* disponibiliza uma interface, a *Continuation,* que é usada no tratamento de tarefas com grandes latências na rede ou de outros tipos de I/O susceptíveis de originarem chamadas bloqueantes ou de consumos significativos de tempo. No caso onde um pedido à rede demore algum tempo a ser executado, como um *lookup* do PAST, por exemplo, o programa ficará bloqueado à espera da sua finalização. Para se evitarem estas esperas, existem as *continuations* que

executam o pedido em *background*, permitindo continuar a execução do programa ou a realização de outros pedidos. Quando o resultado de um pedido de *lookup* do PAST, onde se utilizou um objecto *Continuation*, está disponível, é chamada uma função nesse objecto tendo como parâmetro o valor de retorno. Posteriormente, poderá aceder-se a esse objecto e obter o resultado, ou aquando da recepção do resultado invocar um método que trate o resultado obtido.

4.3.3.3 Environmet

O *environment* é um componente chave para o funcionamento do *Freepastry*. Este componente permite criar uma simulação do ambiente do *pastry* dentro de uma máquina virtual Java. Tal simulação deste *environment* é obtida pelos elementos que o compõem:

- *Logging* – processo de registo de eventos relevantes da rede para diagnóstico de eventuais problemas no sistema (*debugging*).
- *Parameters* – parâmetros que definem as preferências do ambiente virtual a criar, como por exemplo, o tamanho dos *leafsets* do *Pastry*, tamanho máximo das mensagens, frequência das mensagens de manutenção, etc. Estes parâmetros vão ser utilizados para configurar o ambiente de execução do *overlay* do *Freepastry*. Os parâmetros podem ser guardados de forma persistente num ficheiro que é lido no inicio da execução do programa, ou podem ser alterados em tempo de execução, através de uma interface gráfica, por exemplo.
- *SelectorManager* – é uma *thread* única que trata de todos os pedidos I/O e dos eventos da rede. Qualquer actividade accionada na rede é detecta pelo selector que identifica e entrega a actividade aos elementos interessados (por exemplo, a entrega de uma mensagem à aplicação do nó a que a mensagem se destina).
- *TimeSource* – relógio virtual do *Freepastry*. Este elemento é utilizado pelo simulador como relógio virtual. Esta virtualização tem a vantagem de permitir aos nós do *Freepastry* serem executados com um relógio independente do relógio real do computador. Desta forma, o tempo pode avançar muito mais rápido, bem como o tempo de execução dos processos.
- *RandomSource* – interface aleatória virtual para o *Freepastry*. É utilizado para gerar um fluxo de números aleatórios.

4.3.3.4 Application

A aplicação é o programa que acede e se executa sobre o overlay do *Freepastry*. A interface *Application* possibilita a criação de uma aplicação capaz de interagir com a rede, isto é, comunicar com os nós da rede e enviar mensagens para o *overlay.* É também através desta *interface* que a rede comunica com a aplicação e a notifica sobre a chegada de mensagens ou de alterações nos nós vizinhos.

4.3.3.5 Endpoint

Esta interface representa um *endpoint* utilizado pelas aplicações. É graças a esta interface que as aplicações, implementadas de acordo com a *common API*, funcionam com o *Pastry*. O *endpoint* representa, para a aplicação, o ponto de acesso e comunicação com o *overlay*. O encaminhamento efectivo de mensagens para a rede é realizado pelo *endpoint*. No caso de recepção de mensagens serializadas com o mecanismo *RawSerialization* (ver ponto 4.3.3.6), o *endpoint* pode ser configurado com um método de desserialização definido para os diferentes tipos de mensagens e utiliza-lo automaticamente aquando da recepção dessas mensagens.

4.3.3.6 Serialization

No *Freepastry*, uma mensagem é um objecto serializável. A transmissão de mensagens pela rede no *Freepastry* é feita através da transmissão de um fluxo de *bytes*. É na camada de transporte que este objecto é serializado em *bytes* e enviado para a rede. Aquando da recepção num nó, a mensagem é desserializada de volta para um objecto e entregue à aplicação. Por omissão, o *Freepastry* utiliza a serialização do Java, existindo outro mecanismo de serialização mais eficiente disponível no *Freepastry*, a *RawSerialization*. Apesar da serialização do Java ser eficaz e requerer esforço reduzido na criação de novas mensagens, este novo mecanismo foi desenvolvido para combater a grande exigência de CPU, memória e largura de banda por parte da serialização do Java.

4.4 Aplicação do *GiGi*

A aplicação do *GiGi* é composta por quatro camadas que interagem entre si. Em cada nó da rede existe uma instância desta aplicação a ser executada, permitindo que os nós submetam ou recebam tarefas. Em seguida é explicada a implementação das operações de cada camada.

4.4.1 *GiGi Application*

Finalidade – Esta camada é apenas uma representação da interface do sistema com as aplicações reais. Neste caso, o simulador é que activa o funcionamento desta camada.

Funcionamento

Comunica bidireccionalmente com o simulador e com a 2ª camada invocando os serviços seleccionados pelo simulador e fornecendo dados para criação das *Gridlets*.

Processa as *Gridlets* recebidas da 2ª camada para processamento e devolve uma *Gridlet-result*. O processamento não é efectivamente realizado, sendo apenas agendada uma função que devolve um resultado fictício á 2ª camada para o tempo actual mais o tempo de processamento da *Gridlet*.

Utilidade – Meramente representativa para dar noção do sistema como um todo.

4.4.2 *Gridlet Manager*

Finalidade – Responsável pelo tratamento, análise, envio, recepção e reencaminhamento de *Gridlets*.

Funcionamento

Submissão de *Gridlets*

A criação de *Gridlets* é feita a partir dos dados fornecidos pela 1ª camada. Esses dados indicam um custo pré-definido que deverá ser associado à *Gridlet* e contém uma pequena frase associada a um número aleatório, de forma a garantir unicidade dos dados, para ser inserido no campo de dados da *Gridlet*.

Recolha dos Resultados

O mecanismo de recolha de uma *Gridlet-result* utiliza a aplicação do PAST. A chave que identifica o objecto é dada pela função de *hash* do campo de dados da *Gridlet* original. É realizado um pedido de *loookup* ao(s) nó(s) que irá(ão) receber a *Gridlet-result*, resultante de uma inserção no PAST pelo nó que processou a *Gridlet*. Contudo, é necessário aguardar um certo período de tempo para que a *Gridlet* original seja transmitida e processada pela rede e inserida no PAST. Esse tempo corresponde à soma dos tempos de transmissão da *Gridlet* pelos nós da rede, do tempo de processamento dos dados da *Gridlet* e do tempo de inserção do resultado no PAST. No entanto, este valor não pode ser obtido com exactidão já que é impossível determinar o número de *hops* que a *Gridlet* irá percorrer, bem como os tempos de transmissão para além do 1º *hop*, do processamento da *Gridlet* que depende da capacidade do nó que a recebe e o tempo de inserção do resultado no PAST, sendo necessário fazer uma estimativa desse valor. Numa primeira tentativa é estimado um tempo para o melhor caso, através da seguinte fórmula:

2) *Tempo para recolha = 1º hop + Tempo de computação + Tempo médio de inserção*

O tempo de espera por um resultado logo após um *hop* consiste na soma do tempo desse *hop* (transmissão do nó original para o nó vizinho escolhido), com o tempo estimado de processamento dos dados da *Gridlet* (sabemos o custo desse processamento) e com o tempo médio de inserção no PAST (este valor pode ser obtido com base em inserções efectuadas pelo próprio nó). Mais uma vez, este cálculo é uma mera aproximação incapaz de obter valores exactos, tendo em conta a quantidade de factores que influenciam os tempos das actividades. Existe também um mecanismo de tentativas, em que são feitas novas tentativas (valor pré-definido) de recolha do resultado no caso de o *lookup* à cache retornar valor *null*. Para tal, definiu-se uma nova fórmula para os tempos entre as tentativas:

3) *Tempo para nova tentativa = Tempo médio de um hop x Número médio de hops*

O tempo de espera para uma nova tentativa consiste no produto do tempo médio de transmissão entre dois nós e o número médio de *hops* necessários para que uma *Gridlet* alcance um nó com disponibilidade. Estes valores médios são ajustáveis à medida que a rede evolui no tempo, podendo desta forma melhor a estimativa. Quando atingido o valor pré-

definido do número de tentativas, a hipótese de recolha desse resultado é descartada e a *Gridlet* correspondente é dada como falhada, sendo novamente inserida na rede para novo processamento.

Tratamento de *Gridlets*

Na recepção de *Gridlets*, é verificado o custo da *Gridlet* e comparado com os recursos disponíveis actuais do nó. Caso hajam recursos disponíveis e o nó cumpra os requisitos para o processamento, os recursos são actualizados (reduzidos), a *Gridlet* é processada na 1ª camada e o resultado daí derivado é enviado para o sistema de cache, o PAST. A chave do objecto resultante é obtida a partir da função de *hash* do campo de dados da *Gridlet* original. Na impossibilidade da computação da *Gridlet*, em que o nó não tenha recursos disponíveis ou não suporte o custo de processamento da *Gridlet*, esta é reencaminhada para outro nó.

Utilidade – Ponto de decisão da aceitação para processamento das *Gridlets*, de acordo com a disponibilidade do nó, e inserção dos resultados obtidos do seu processamento. Todas as operações efectuadas ao nível das *Gridlets* são feitas nesta camada.

4.4.3 *Overlay Manager*

Finalidade – Responsável pela manutenção da rede *overlay*, pela troca de mensagens na rede e pela manutenção da informação sobre os recursos da máquina local e dos nós vizinhos.

Funcionamento

Recursos locais

Os recursos de cada nó são valores fictícios que podem ser gerados aleatoriamente ou obtidos a partir de um ficheiro. O *Overlay Manager* é responsável pelas alterações dos recursos.

Tabela de recursos e reputação

São mantidas duas tabelas de informação relativas ao conjunto dos nós vizinhos mais próximos geograficamente, a tabela de recursos e a tabela de reputação. Na tabela de recursos descrevem-se as disponibilidades de cada nó em termos das métricas utilizadas (CPU, memória e largura de banda) e o grau de proximidade (latência) que separa o nó local de nó. Na tabela de reputação, cada nó vizinho possui uma entrada composta por quatro valores que definem a reputação desse nó: o número de resultados falhados, isto é, o número de vezes que ao encaminhar por este nó a *Gridlet* foi dada como foi falhada, sendo necessária nova transmissão da *Gridlet*; o número de tentativas utilizadas para obter os resultados; o número de vezes que a *Gridlet* submetida retornou à origem; o valor da medida ponderada da reputação do melhor vizinho desse nó. Estes valores são actualizados com a progressão da rede no tempo, à medida que o nó local envia pedidos para esses nós, e no final analisa a eficiência da realização do trabalho obtido, consequência de ter escolhido aquele nó como primeira transmissão. Após um certo número de actualizações, 100 por exemplo, a tabela de reputação é reinicializada, isto é, os dados desta tabela são repostos a zero, excepto o valor do melhor vizinho desse nó, para diminuir o erro gerado pelas entradas muito antigas.

Selecção de um destino

O envio das *Gridlets* é feito para o nó vizinho capaz de satisfazer a computação pedida (custo da *Gridlet*) tendo em conta vários critérios na selecção, como o cumprimento de recursos mínimos disponíveis para o processamento dos dados da *Gridlet* e a escolha do melhor nó. Este processo de selecção do nó é efectuado, no mínimo, em duas fases, podendo chegar até quatro fases: 1) o nó deve ter a disponibilidade mínima capaz de processar a *Gridlet*; 2) de entre os escolhidos na fase 1, é seleccionado aquele que manterá maior disponibilidade após receber o pedido e cujo grau de proximidade seja o menor (processo explicado em 4.2.2); 3) caso as duas fases anteriores não tenham achado nenhum nó adequado, é escolhido o nó com o melhor nível de reputação (processo explicado em 4.2.3); 4) se em nenhuma das fases anteriores foi possível achar um nó, pode-se deduzir que a rede local não possui disponibilidade ou está sobrecarregada, sendo então escolhido o nó vizinho mais distante geograficamente, com o objectivo de propagar o pedido o mais longe possível onde haja disponibilidade.

Verificação do estado dos nós

Durante a iteração dos nós do conjunto de vizinhança no processo de selecção de um destino, é verificado o estado de cada nó (activo ou inactivo). Caso o estado seja inactivo, as entradas desse nó nas tabelas de recursos e reputação são removidas, caso contrário a selecção procede normalmente. Esta verificação é necessária devido à existência de nós no conjunto de vizinhança que, originalmente, não viam o nó local como sendo seu vizinho e o aceitaram posteriormente (ver 4.2.1). Quando um dado nó fica inactivo, o *overlay* notifica apenas o conjunto de vizinhança facultado pelo *overlay* desse nó, de que o mesmo já não faz parte da rede. Assim, esta verificação serve para detectar este tipo de casos em que um nó já não faz parte da rede mas o nó local não recebeu a notificação do *overlay*.

Pré-redução

Quando um nó é escolhido na fase 1 e 2 significa que este tem disponibilidade para processar o pedido. Desta forma, pressupõe-se que será esse nó a processar a *Gridlet* e podemos actualizar de imediato a informação que temos sobre o nó, incutindo o custo dessa *Gridlet* sobre a informação dos recursos disponíveis desse nó. O facto de actualizarmos imediatamente este valor é crucial para evitar sobrecarregar o nó com maior disponibilidade que se conhece. No entanto, pode acontecer que a *Gridlet* não seja processada por esse mesmo nó, porque outra *Gridlet* chegou primeiro, por exemplo. Assim sendo, a pré-redução dos recursos é temporária, isto é, só dura o tempo de RTT entre o nó local e o nó escolhido mais cinco milissegundos, para efeitos de atraso no nó. Caso o nó processe efectivamente a *Gridlet*, o nó local receberá, na melhor das hipóteses, uma mensagem de *update* no tempo de RTT. Caso contrário, a redução dos custos é reposta em RTT + 5 milissegundos.

Medidas restritivas

São também implementadas quatro medidas restritivas aplicadas na selecção dos nós de destino com o objectivo de evitar a repetibilidade. A primeira medida passa pela negação da

escolha do nó que enviou a *Gridlet*. A segunda consiste numa tabela com os identificadores das *Gridlets* enviadas e a respectiva lista dos nós que foram escolhidos na fase 1 e 2, ou seja, nós com presumível disponibilidade. Se a *Gridlet* regressar a um nó que a recebeu anteriormente e a reencaminhou para um nó que pensava estar disponível, significa que essa informação estava incorrecta ou desactualizada. Assim, durante a fase 1 e 2 não são escolhidos novamente os mesmos nós que antes reencaminharam essa *Gridlet*. Estas restrições são levantadas para todas as entradas de um nó quando for recebida uma mensagem de *update* válida proveniente desse nó. Existem ainda outras duas listas para a fase 3 e 4. Uma delas irá conter todos os nós que foram escolhidos, quer nas fases 1 e 2 quer na fase 3. Mas esta lista só restringe a selecção na fase 3, para evitar que nós repetidos sejam novamente escolhidos por terem melhor reputação. Finalmente, uma última lista contendo os nós escolhidos na fase 4. Apesar de ser escolhido o nó mais distante, é necessário evitar que o mesmo seja escolhido repetidamente, adicionando-o a uma lista que o restringe. Estas últimas listas são esvaziadas quando todos os nós vizinhos fizerem parte dela.

Alterações na vizinhança

Quando ocorrem alterações ao conjunto da vizinhança, o overlay do *Freepastry* notifica a aplicação, através da 4ª camada que direcciona a chamada para a 3ª camada, sobre essas alterações, indicando o vizinho que entrou ou deixou o conjunto. No caso da entrada de novos nós, são adicionadas novas entradas para esse novo nó à tabela de recursos e à tabela de reputação do nó local. Em seguida, o nó local responde com o envio de uma mensagem de *update* anunciando os seus recursos. Caso se verifique a saída de um nó, são apenas removidas as entradas das tabelas referentes a esse nó.

Anunciar disponibilidade

Aquando da ocorrência de redução dos recursos de um nó devido ao processamento de uma *Gridlet*, a disponibilidade do mesmo é reduzida. São então actualizados os seus recursos e enviadas mensagens de *update*, contendo os valores das alterações e a duração das mesmas [52], para os seus nós vizinhos. Para o caso de ocorrerem alterações permanentes (definidas pelo utilizador ou simplesmente negação de recursos) as mensagens de *update* definem apenas os novos valores dos recursos. Desta forma, a informação relativa a nós vizinhos estará a mais actualizada possível, conferindo maior eficiência nas decisões de encaminhamento tomadas. De notar que os nós que recebem as mensagens de *update* não necessitam de responder à mensagem. A opção de anunciar os recursos apenas quando se sofrem alterações ao nível dos recursos ou ao nível da rede (ponto anterior) é mais eficiente do que anunciar periodicamente. No entanto, esta opção deverá ser melhor abordada quando o sistema for executado num ambiente real, já que os recursos de uma máquina estão em constante alteração. Para este caso, o ideal seria definir intervalos nas variações dos recursos e, ultrapassados esses limites, propagar as mensagens de *update*.

Utilidade – Manutenção de informação sobre a rede de forma a avaliar esses dados e tomar decisões de encaminhamento, o que influencia directamente a eficiência e eficácia do sistema.

A o envio e recepção de mensagens *update* com os nós vizinhos permite anunciar e adquirir, respectivamente, informação actualizada do estado dos nós e da própria rede.

4.4.4 *Communication Service*

Finalidade – Interage e estabelece a verdadeira comunicação com a rede, desde o envio, recepção e notificações de entrada e saída de nós do conjunto da vizinhança.

Funcionamento

Esta camada possui a referência para a instância do nó local, isto é, o seu *endpoint*. Desta forma é capaz de interagir com a rede do *Freepastry*, podendo enviar e receber mensagens. Todas as mensagens recebidas e enviadas estão serializadas (*RawMessage*).

Na recepção de mensagens, é verificado o tipo da mensagem (*Gridlet* ou *update*) de forma a invocar a função apropriada para o tratamento da mensagem, na 3ª camada.

Quando ocorrem alterações no conjunto da vizinhança, o *overlay* notifica as aplicações dos nós afectados através desta camada. A informação recebida é enviada para a camada acima, o *Overlay Manager*, onde será tratada em conformidade.

Utilidade – Ponto de saída e de entrada nas comunicações entre os nós da rede, resultando na interface do sistema com o *overlay* p2p utilizado, o *Freepastry*.

4.5 Funcionalidade do *GiGiSimulator*

É neste componente que todo o sistema é iniciado e se procede ao controlo da simulação. Todos os mecanismos essenciais para o funcionamento da rede do *Freepastry* são aqui iniciados e configurados, bem como a gestão do simulador, a monitorização e a recolha de dados estatísticos das mensagens transmitidas pela rede.

4.5.1 Criação da Rede

Numa primeira instância, são carregadas as configurações do *Pastry* e é iniciado o ambiente para o simulador. A criação do simulador começa por definir a topologia da rede a utilizar. Para este sistema é utilizada uma rede do tipo *Euclidean* (Planar), uma vez que emula uma rede de nós que estão aleatoriamente dispersos num plano de duas coordenadas. A proximidade entre os nós é baseada numa distância euclidiana (norma do vector), que corresponde à distância vectorial entre os dois pontos que definem os nós.

Posteriormente são criados e lançados os nós para a rede, sendo executados e controlados pela thread do *Selector* do *Freepastry*. O primeiro nó irá criar a sua própria rede onde os outros nós se ligaram sucessivamente. À medida que os nós são criados e lançados na rede, é-lhes associada uma nova instância da aplicação do *GiGi*. O simulador interage com os nós através da 1ª camada, a *Application Adaptation.* Desta forma, teremos cada nó com a sua própria aplicação do *GiGi*, que só é executada quando o simulador invoca chamadas à mesma ou

quando o *Selector* notifica o nó, correspondente dessa aplicação, com recepção de mensagens ou alterações ao seu conjunto de nós vizinhos.

Na criação de cada nó, é também lançada a aplicação de cache a utilizar, o PAST. A instância desta aplicação é controlada pela aplicação do *GiGi*, de forma a permitir o armazenamento e a recolha dos resultados na cache. O sistema de cache é definido com um algoritmo de cache LRU (*Least Recently Used*), isto é, quando a cache estiver cheia, são descartados os itens menos utilizados. O *MemoryStorage* é o tipo de armazenamento utilizado, uma implementação de armazenamento não persistente em memória. Esta escolha, em detrimento do armazenamento persistente, deve-se ao facto de a utilização de pequenos objectos guardados em memória consegue não obstar a simulação de uma grande rede, dependendo apenas na quantidade de memória RAM disponível.

Após toda a iniciação e configuração dos nós da rede segue-se um reconhecimento da rede local, isto é, dos nós vizinhos, de cada nó. O reconhecimento tem como objectivo a criação do conjunto de nós vizinhos e a manutenção dessa informação nas tabelas da camada de *Overlay Manager*. Durante este processo, todos os nós anunciam a suas disponibilidades e capacidades, em termos de recursos, através do envio de mensagens de *update* pelos seus nós vizinhos. Assim, o sistema arranca de um ponto em que as informações sobre a rede, presentes nos nós, estão já completas.

4.5.2 Interface do Simulador

A geração de eventos a partir do simulador sobre as aplicações do *GiGi* nos nós é apresentada numa interface de comandos onde se disponibilizam as seguintes opções: “submissão de *Gridlets”*, “ligar ou remover nós”, “obter informações”, “testar rede” e “simulação automatizada”.

Submissão de *Gridlets*

De uma forma linear, a submissão de tarefas para a rede através de *Gridlets* é iniciada na 1ª camada do *GiGi* que fornece os dados para a criação de *Gridlets* na 2ª camada, que por sua vez envia para a 3ª camada onde as *Gridlets* são encapsuladas no formato *Message* e é seleccionado o nó de destino. Posteriormente são enviadas para a 4ª camada que serializa e envia a mensagem para o nó de destino no *overlay*. A submissão de *Gridlets* é iniciada pelo simulador que invoca uma ou mais aplicações do *GiGi* em novas *threads*. Uma vez que este sistema contempla apenas o funcionamento ao nível da rede, é necessário simular de alguma forma a existência de um fornecedor de dados para a 1ª camada. O *GiGiSimulator* providencia esses dados e os custos de processamento necessários à criação das *Gridlets*. Como o processamento de uma *Gridlet* é simulado, consumindo apenas o tempo de processamento, o campo de dados que esta transporta é irrelevante, tendo apenas que ter um valor distinto de outras *Gridlets*, visto que é a partir da função de *hash* deste campo que se obtém a chave a utilizar pelo PAST. No entanto, é necessário um custo adequado para as *Gridlets*. Para atingir este objectivo, foram criados perfis, cada um caracterizado por um custo pré-definido: alto custo, médio custo, baixo custo (todos os recursos possuem iguais valores de

elevadas/médias/baixas exigências), alto custo de CPU, alto custo de Largura de Banda e alto custo de Memória (grande custo de CPU/Memória/Largura de Banda e baixo custo idêntico para os restantes). Para além da escolha de perfis, é possível definir o número de nós a enviarem *Gridlets* bem como o tempo de intervalo entre cada envio.

Ligar / Remover nós

Permite adicionar novos nós à rede ou remover nós existentes. A ligação de um novo nó será estabelecida através de um nó pertencente à rede, escolhido aleatoriamente. De acordo com os valores das coordenadas no espaço Euclidiano que o novo nó recebe durante a sua criação, este irá alterar alguns conjuntos de vizinhança de nós já existentes, originando a sua entrada nesses conjuntos ou a substituição com outro nó desse conjunto (a substituição dá-se com o nó mais distante, quando o conjunto da vizinhança de um nó está já completo e o novo nó está mais próximo do que o nó mais distante). A remoção de um nó consiste simplesmente em remover a instância da aplicação do GiGi mantida no simulador e na destruição e remoção do nó da rede. O *overlay* irá, automaticamente, detectar a saída do nó. As alterações são notificadas à camada de *Communication Service* dos nós respectivos pela *thread* do *Selector*. Esta camada envia a informação recebida para a camada acima, responsável pela manutenção do *overlay* e das tabelas relativas aos nós vizinhos, que irá proceder às alterações necessárias.

Obter informações

Recolhe informações relativas a todos os nós da rede. Permite visualizar os recursos actuais de cada nó; as tabelas de métricas de cada nó, que mantêm os valores dos recursos dos seus vizinhos obtidos através das mensagens de *update* enviadas pelos vizinhos; as tabelas de reputação, referente às análises obtidas no passado sobre os nós vizinhos, resultado da escolha desse nó como primeiro nó de encaminhamento de uma *Gridlet*; estatísticas do historial dos nós, isto é, medições efectuadas à execução desse nó, como o número de *Gridlets* enviadas, processadas, redireccionadas, resultados falhados, número de tentativas para recolha dos resultados e nó mais usado como primeira opção.

Testar rede

São enviadas *Gridlets* tipo (*Gridlets* todas iguais com um custo unitário predefinido) por diferentes nós com o objectivo de medir a eficácia e o tempo de resposta da rede. Os valores obtidos neste teste podem ser utilizados para ajustar certas variáveis de medição como o tempo médio de inserção ou *lookup* do PAST ou o tempo médio de um *hop*.

Simulação automatizada

Corre uma bateria de testes pré-definida (*scripts*). Os testes consistem apenas na utilização das funcionalidades anteriores, alterando os parâmetros seleccionados para cada caso. Esses testes estão *hard-coded*.

4.5.3 Monitorização

O simulador do *Freepastry,* oferece ainda uma forma de monitorizar as mensagens transmitidas pelo *overlay*, através da interface de um *SimulatorListener*. A implementação desta interface no simulador permite receber os dados de qualquer mensagem, bem como a sua origem e destino, sempre que uma mensagem é enviada para a rede. Esta funcionalidade permite-nos obter estatísticas sobre o fluxo da rede como o número de mensagens transmitidas, o tipo das mensagens, os tempos de envio das mensagens ou o número de *hops* que uma mensagem percorreu.

4.6 Tipos de Mensagens

Neste sistema, com a excepção das mensagens de controlo enviadas pelo *Freepastry*, existem dois tipos principais de mensagens que são enviados pela rede, as *Gridlets* e as mensagens de *update*. As *Gridlets* são constituídas por um custo do processamento de uma tarefa e pelos dados necessários ao processamento da tarefa. Nós com disponibilidade suficiente para suportarem os custos inerentes a uma *Gridlet* processam a tarefa. As mensagens de *update* são mensagens cujo objectivo passa pela divulgação dos recursos de um nó e a disponibilidade do mesmo. Existem ainda outros dois tipos de mensagens utilizados, as *Gridlet-results* e os *Content Result*. As *Gridlet-results* são mensagens particulares do tipo *Gridlet*. As mensagens do tipo *Content Result* são o único tipo de mensagens que a aplicação do PAST aceita.

Serialização

Como já foi referido anteriormente em 4.3.3.6, no *Freepastry*, uma mensagem é um objecto serializável. Cada um destes tipos deve então ser serializável. Quando uma destas mensagens é enviada para a rede, o *Freepastry* chama o método *serialialize()* destas mensagens com um *OutputBuffer*. Cada variável constituinte da mensagem é serializada para o *OutputBuffer*. Para desserializar as mensagens, é necessário instalar um desserializador no *endpoint* da aplicação que será automaticamente utilizado na recepção das mensagens. Quando o desserializador é invocado, é-lhe passado um *InputBuffer*, que possui as funções correspondentes ao *OutputBuffer*, possibilitando a desserialização das variáveis das mensagens. Graças a este mecanismo é possível obter uma serialização mais eficiente, em termos dos recursos necessários para o processo, das mensagens.

Em seguida, são analisados e descritos os tipos de mensagens utilizadas pela aplicação do *GiGi*.

4.6.1 Gridlets

As mensagens *Gridlet* transportam os dados alvos de processamento e o custo associado a essa computação. O conteúdo das mensagens é constituído pelas variáveis existentes no objecto que definem as *Gridlets*. As variáveis são o CPU, memória e largura de banda, constituindo o custo de processamento da *Gridlet*. Estas variáveis tomam valores normalizados (em vez de termos 1MFLOP ou 10MB temos unidades de equivalente valor para cada recurso), de forma a simplificar o cálculo do custo de uma *Gridlet.* Para além disso, existe o campo de dados com a informação e dados necessários à computação. No entanto, a informação contida

neste campo consiste apenas num valor único, de forma a distinguir as diferentes *Gridlets*, já que a computação neste sistema é simulada e não são efectuadas operações no campo dos dados.

4.6.2 Mensagens de *Update*

Estas mensagens transportam informações acerca dos recursos de um nó. Têm a finalidade de propagar pelos nós vizinhos a disponibilidade dos recursos acerca de um nó. Cada mensagem é constituída pelo identificador do nó que originou a mensagem, pelos valores actuais dos recursos desse nó e um número de sequência. Para além disso, existe uma opção que permite a um nó anunciar a alocação dos seus recursos durante um determinado intervalo de tempo, utilizando para isso um campo na mensagem com o valor da duração dos novos recursos. No valor dos recursos vai especificado o custo a reflectir sobre os recursos anteriores e uma *flag* sinalizando se o custo é relativo a consumo ou acréscimo.

Esta opção é sempre utilizada quando um nó decide processar uma dada *Gridlet*, incutindo o custo da computação nos seus recursos e nas mensagens de *update*. Desta forma, os seus vizinhos são notificados de que durante o processamento da *Gridlet*, aquele nó terá os seus recursos reduzidos. Quando os nós recebem este tipo de mensagem, diminuem os recursos alusivos a esse nó nas suas tabelas de métricas e agendam uma soma de igual valor nesses recursos para o tempo definido na mensagem de *update*. Este mecanismo [52] permite uma redução de 50% de mensagens de *update* enviadas, para casos de alteração temporária dos recursos, em comparação a enviar mensagens sempre que ocorram alterações.

Cada nó envia este tipo de mensagens com valores relativos apenas aos seus recursos. A profundidade de propagação destas mensagens nunca ultrapassa o conjunto de vizinhos do nó remetente. Cada nó é responsável pela divulgação da sua disponibilidade. Desta forma, um nó que queira limitar a disponibilidade dos seus recursos por qualquer razão, pode fazê-lo alterando os valores destas mensagens.

4.6.3 Gridlet-result

As *Gridlets* podem tomar outro tipo particular de mensagem, utilizado para transportar os resultados obtidos da computação dos dados de uma *Gridlet* original, as *Gridlet-results*. A única particularidade deste tipo é o valor dos campos da *Gridlet*, onde o campo dos dados transporta os resultados derivados da computação sobre os dados e os valores relativos ao custo indicam o custo efectivo do processamento da *Gridlet* original.

4.6.4 Content Result

Este tipo de mensagens serve apenas de contentor para as mensagens do tipo *Gridlet*, nomeadamente, as *Gridlet-results*. Esta necessidade deve-se à aplicação do PAST, que requer a utilização de um tipo de dados pré-definido, através da interface *ContentHashPastContent*, para a transmissão de mensagens no seu sistema de armazenamento. Qualquer mensagem inserida na cache do PAST ou obtida da mesma tem de estar neste formato específico. O

encapsulamento de uma *Gridlet* numa mensagem de *ContentResult* e a aquisição da *Gridlet* a partir da mensagem *ContentResult* é efectuado ao nível da 2ª camada da aplicação do *GiGi*, na *Gridlet Manager*.

4.7 Considerações da Implementação

Um aspecto a ter em consideração no funcionamento deste sistema é o facto de se utilizar um simulador com relógio virtual e todos os nós serem emulados numa só máquina. Este facto leva a algumas limitações, mas em contra partida, proporciona um ambiente satisfatório e o mais próximo do real para a execução de testes ao sistema.

O desempenho da simulação está directamente relacionado com a velocidade com que o relógio do simulador é actualizado. É possível limitar a velocidade do simulador, por exemplo, para apenas dez vezes mais rápido que o tempo real. O simulador, antes de avançar o tempo para executar uma tarefa da lista de espera, verifica se existe alguma limitação de tempo e espera o tempo real necessário para ir de acordo com essa limitação. Ou simplesmente pode-se ajustá-lo à velocidade máxima, isto é, avançar o relógio sempre que haja uma tarefa a executar. Contudo, definir a velocidade máxima para o simulador não é prudente, devido à execução das aplicações sobre várias *threads*. Por exemplo, no caso em que, após a entrega de uma *Gridlet* a um nó, não haja tarefas imediatas na lista de espera da *thread* do *Selector* para serem executadas, o simulador avançara de imediato o relógio. O nó que recebeu a *Gridlet* não dispõe de recursos para a processar e reencaminha a mensagem para outro nó. No entanto, o tempo do seu pedido de envio da mensagem e o tempo de recepção da mesma irá apresentar uma grande flutuação entre eles. Este fenómeno deve-se ao facto de o simulador não estar preparado para ambientes *multi-threading* e pode gerar flutuações na medição dos tempos de envio e recepção de mensagens, caso o relógio seja incrementado muito depressa. Assim, o factor da velocidade é escolhido com base em testes prévios e, dependendo do tamanho da rede, poderá ser alterado em tempo de execução.

Outro aspecto importante ao desempenho deve-se à utilização de várias *threads* para correrem os diferentes processos em cada nó. Um nó pode ter uma *thread* a enviar mensagens e outra a receber. De referir no entanto que a utilização de *threads* afecta ligeiramente a escalabilidade do sistema no que diz respeito ao número de nós na rede e à quantidade de pedidos que podem ser efectuados. No entanto, a consistência dos resultados não é afectada já que o simulador do *Freepastry* garante que todas as tarefas serão processadas e só avança o relógio quando não tem tarefas imediatas. Desta forma, as únicas limitações para a escalabilidade são a memória da máquina onde corre o simulador e o número de *threads* que se podem criar e executar concorrentemente de forma eficiente.

A maior limitação do simulador, e do próprio *overlay* DHT, é o *overhead* que cada nó do *Pastry* apresenta (alguns *kilobytes*). Este valor de utilização da memória RAM será sempre fixo enquanto a rede estiver a funcionar. Em última instância, os requisitos de memória serão os principais responsáveis pela capacidade de execução de muitos nós numa única máquina.

5. Simulação e Avaliação do Desempenho

Nesta secção são descritos vários testes para a medição e avaliação do desempenho do sistema criado. Tais testes consistem na simulação de fluxos de mensagens pela rede, sobre diferentes comportamentos que a simulação pode tomar através da alteração e combinação de parâmetros da rede e variações nos parâmetros de configuração do sistema. Para a realização deste objectivo, foi utilizado o simulador do *Freepastry* que permite criar e executar a plataforma criado no overlay P2P do *Freepastry*. O mesmo permite simular diversos nós, em condições muito próximas da realidade, oferecendo toda uma gama de ferramentas de monitorização da rede, evitando assim a necessidade de infra-estruturas físicas para a realização dos testes.

Durante as simulações, os nós no sistema estarão divididos em 2 grupos: um grupo constituído por nós ociosos, que disponibiliza os seus ciclos de processamento para execução de tarefas; e outro grupo de nós solicitadores, que irão submeter pedidos de utilização dos ciclos excedentários disponíveis nos nós ociosos.

Cada teste terá ao seu dispor o controlo sobre diversos parâmetros de configuração, sendo elas:

1) O número de nós da rede;
2) A percentagem dos tipos de nós, nós indisponíveis e nós ociosos;
3) A disponibilidade total presente na rede;
4) O número de *Gridlets* a submeter;
5) O custo da computação (em unidades) das *Gridlets* submetidas;
6) O número de nós solicitadores, isto é, que enviam pedido.

Para além disso, poderão também ser configurados valores de variáveis internas ao sistema, que influenciam o encaminhamento dos pedidos, com o objectivo de revelar a melhor opção. No final são analisados e justificados os resultados obtidos em cada teste.

Todos os testes foram realizados numa só máquina com as seguintes características: processador Intel Core 2 Duo T8300 2.40GHz, 3070 MB de memória RAM e Sistema Operativo Windows Vista 32 bits. O sistema foi executado sobre a aplicação do *Freepastry* versão 2.0_01, a correr na plataforma do *NetBeans* IDE 6.0.1 com a versão 1.6.0_06 do Java JDK.

5.1 Aplicação e Procedimento dos Testes

5.1.1 Aproveitamento de recursos

Uma primeira avaliação é feita à principal funcionalidade do sistema, isto é, a descoberta e o aproveitamento de recursos ociosos na rede. O processo consiste no sucessivo envio de pedidos para a rede, até que a quantidade de recursos solicitados seja o dobro do que a rede dispõe. Para além disso, este cenário é testado para diferentes quantidades de nós solicitadores, com o pretexto de estudar a possibilidade de este factor influenciar a descoberta e consumo dos recursos.

O objectivo principal consiste na descoberta e consumo dos recursos solicitados, atingindo e ultrapassando a saturação (limite) da disponibilidade instalada nos nós da rede. Como

objectivos secundários, espera-se uma boa eficiência por parte do sistema, em termos do número médio de *hops* na rede necessários para que uma *Gridlet* seja atendida, da variação da quantidade de mensagens transmitidas para os diferentes níveis de saturação atingidos, a evolução do número de tentativas necessárias à recolha dos resultados e do tempo necessário para a concretização das simulações.

Procedimento

Parâmetro	Valor	Parâmetro	Valor
Número de nós	1000 nós	Tipos de nós	500 indisponíveis 500 ociosos
Disponibilidade total	[4000, 4000, 4000] [CPU, memória, larg. banda]	Custo das *Gridlets*	Perfil: Alto Custo [8,8,8]

Variáveis: o número de pedidos a enviar, que varia entre 300 e 1000 *Gridlets*; e a quantidade de nós que enviam pedidos, desde todas as *Gridlets* a serem enviadas a partir de um único nó até o envio disseminado a partir de todos os nós.

A aplicação deste teste combina os valores das variáveis flexíveis, acima referidas. Para cada número de *Gridlets* a submeter, são repetidos os testes para as diferentes quantidades de nós que geram os pedidos: 1, 100, 200, 300, 400 e 500 nós a enviar as *Gridlets*. Pelo número de *Gridlets* a submeter é possível determinar a quantidade dos recursos requisitados:

300 G x [8, 8, 8] = [2400, 2400, 2400]; 400 G x [8, 8, 8] = [3200, 3200, 3200];

500 G x [8, 8, 8] = [4000, 4000, 4000]; 600 G x [8, 8, 8] = [4800, 4800, 4800];

700 G x [8, 8, 8] = [5600, 5600, 5600]; 800 G x [8, 8, 8] = [6400, 6400, 6400];

900 G x [8, 8, 8] = [6400, 6400, 6400]; 1000 G x [8, 8, 8] = [8000, 8000, 8000];

De notar que cada simulação será executada sobre uma nova rede criada, evitando assim influências sobre os resultados por parte dos mecanismos de reputação dos nós.

5.1.2 Ganhos comparativos dos mecanismos de selecção

O ganho proporcionado pela descoberta dos recursos e selecção do melhor nó para encaminhar os pedidos, pode ser calculado analisando a diferença do funcionamento normal do sistema ou configurando ou desactivando os mecanismos de selecção. Para se obter as diferenças da eficiência na transmissão de pedidos pela rede até que estes sejam tratados, é necessário configurar individualmente os três componentes que influenciam a selecção dos nós a encaminhar: a) a informação sobre a disponibilidade dos nós vizinhos, b) a pré-redução sobre o conhecimento dos recursos de um nó com disponibilidade que acabou de ser seleccionado e c) a capacidade de os nós aprenderem sobre os seus vizinhos.

a) A informação sobre a disponibilidade de um nó é obtida a partir de uma medida ponderada sobre os conhecimentos dos recursos desse nó e a sua proximidade com o nó local. Daí, deverão ser testados as diferentes execuções com diferentes pesos para o cálculo desta

medida. Para medições eficazes, os testes deverão ocorrer em ambientes da rede com recursos suficientes, exactos (ponto de saturação) e em excesso.

b) Quando se escolhe um nó que se julga estar disponível para realizar uma dada tarefa, ao conhecimento sobre os recursos desse nó é-lhe automaticamente reduzido o custo da tarefa a submeter, com o objectivo de alcançar um estado o mais actualizado possível, através da antecipação do que irá acontecer. A diferença da eficiência (número de hops, número de tentativas de recolha dos resultados, etc.) entre a execução com e sem pré-redução indicará a importância da necessidade de actualização imediata.

c) Graças ao sistema de reputação mantido em cada nó, estes podem adquirir informações sobre o rendimento da sua vizinhança no passado e aprender os melhores caminhos à medida que se enviam mais pedidos para a rede. O ganho da utilização deste mecanismo pode ser obtido, a partir da diferença entre a execução do sistema com e sem mecanismo de reputação activo, tendo em conta que cada simulação deverá ocorrer sequencialmente e deverá ser sempre o mesmo nó a enviar os pedidos, para permitir que o sistema evolua e aprenda sobre as vizinhanças onde os pedidos foram submetidos.

O grande objectivo deste teste é avaliar o rendimento e desempenho extra obtidos a partir dos métodos utilizados para a gestão e descoberta de recursos neste sistema.

Procedimento

Parâmetro	Valor	Parâmetro	Valor
Número de nós	1000 nós	Tipos de nós	500 indisponíveis 500 ociosos
Disponibilidade total	[4000, 4000, 4000] [CPU, memória, larg. banda]	Custo das *Gridlets*	Perfil: Alto Custo [8,8,8]
Nós solicitadores	100 nós	-	-

Variáveis:

a) O número de *Gridlets* a submeter, que varia desde 300, 500 (1º ponto de saturação), 700, 900, 1000 (2º ponto de saturação[1]) e 1200 *Gridlets*; e a variação do peso das métricas (CPU, memória e largura de banda) e da proximidade no cálculo de selecção de um nó;

b) O número de *Gridlets* a submeter, que varia desde 300 até 1000 *Gridlets*; execução com e sem pré-redução de recursos;

c) O número de *Gridlets* a submeter será fixo, 700 *Gridlets*, sempre enviadas pelo mesmo nó, sendo os dados recolhidos ao longo do tempo em que se injectam 700 pedidos de forma sequencial; e execução com e sem o mecanismo de reputação.

[1] Recursos requeridos perfazem o dobro da disponibilidade na rede

Os três mecanismos alvos deste teste foram testados individualmente, de acordo com as especificações necessárias para cada um. No entanto, estes mecanismos estão dependentes de uma forma cumulativa entre si, por exemplo, a pré-redução em b) só é efectuada depois de um nó ter sido escolhido em a) e o sistema de reputação só funciona com base em resultados anteriores de escolhas em a), não sendo clarificador, ou muito relevante, isolar um mecanismo e apenas esse influenciar o encaminhamento.

5.1.3 Escalabilidade

Este teste permite verificar o comportamento da plataforma em casos de alterações do tamanho, em número de nós, e da disponibilidade da rede. Para isso, é iniciada uma rede de pequenas dimensões e disponibilidade adequada ao seu tamanho, que vai aumentando gradualmente com a entrada de novos nós que trazem mais recursos, até um dado pico. A partir desse pico, são removidos nós que vão diminuindo o tamanho e a disponibilidade da rede, até ao tamanho e quantidade de recursos inicial. Para testar adequadamente o comportamento da plataforma, deverá ter-se em conta, durante a simulação e na análise dos resultados, os factores inerentes ao sistema de aprendizagem dos nós durante as transacções e outras contra-medidas implementadas.

O objectivo passa por estudar e analisar de que forma a entrada de novos nós na rede, trazendo mais recursos, e a saída de nós, influenciam os resultados do sistema, tendo em conta os seus mecanismos de gestão e descoberta dos novos recursos ou de recursos que já não existem no sistema. Desta forma, este teste irá mostrar a maneira como o sistema lida com a escalabilidade e dinamismo da rede, através das variações do número de nós e da disponibilidade na rede.

Procedimento

Variáveis fixas: a percentagem dos tipos de nós, 50% de nós solicitadores e 50% de nós ociosos; a forma de distribuição dos recursos será homogénea; o custo das *Gridlets* submetidas, perfil de baixo custo (custo de uma unidade para cada métrica); o número de nós que enviam pedidos, 100 nós.

Parâmetro	Valor	Parâmetro	Valor
Tipos de nós	50% indisponíveis 50% ociosos	Custo das *Gridlets*	Perfil: Alto Custo [8,8,8]
Nós solicitadores	100 nós	-	-

Variáveis: o número de nós na rede que varia em dois sentidos, crescente, começando com 200 nós, 400, 600, 800, 1000 e 2000 nós, e no sentido decrescente desde 2000 para 1000, 800, 600, 400 e 200; a disponibilidade da rede será aumentada conforme a entrada de novos nós na rede, desde [200, 200, 200], e visto que ao entrarem 200 nós, apenas 100 serão ociosos, cada um desses 100 trará 2 unidades para cada métrica de recursos; e o número de *Gridlets* a enviar que será sempre igual a 120% da disponibilidade na rede, por exemplo, com 200 nós e disponibilidade de [200, 200, 200] são submetidos 240 pedidos.

5.2 Resultados dos Testes

5.2.1 Aproveitamento de recursos

Para a disponibilidade da rede definida, esperam-se resultados satisfatórios, pelo menos até ao envio de 500 *Gridlets* (onde os pedidos alcançam os valores de recursos disponíveis na rede). A partir desse ponto prevêem-se maior número de retransmissões e saltos pelos nós da rede até que haja recursos novamente livres. Em relação ao número de nós que submetem *Gridlets,* esperam-se melhores resultados quando forem enviadas apenas uma *Gridlet* por nó, ficando assim todos os pedidos dispersos pela rede.

Neste cenário foram executadas várias simulações, variando os valores do número de pedidos submetidos e da quantidade de nós que geravam esses pedidos. Em seguida são apresentados os resultados para as medições definidas no procedimento: o número médio de *hops* na rede para o atendimento das *Gridlets*, a variação da quantidade de mensagens transmitidas, o número de tentativas para a recolha dos resultados e o tempo necessário para a concretização das simulações.

Em todas as simulações executadas o tratamento das *Gridlets* e a recolha de todos os resultados foi efectuado com sucesso. Mas para cada simulação verificaram-se diferentes valores nas medições definidas. Na Figura 15 é possível verificar qualidade da descoberta de recursos obtida, através do número médio de *hops* de cada pedido, consoante o número de *Gridlets* que foram submetidas. As linhas de diferentes cores indicam a simulação com o respectivo número de nós solicitadores.

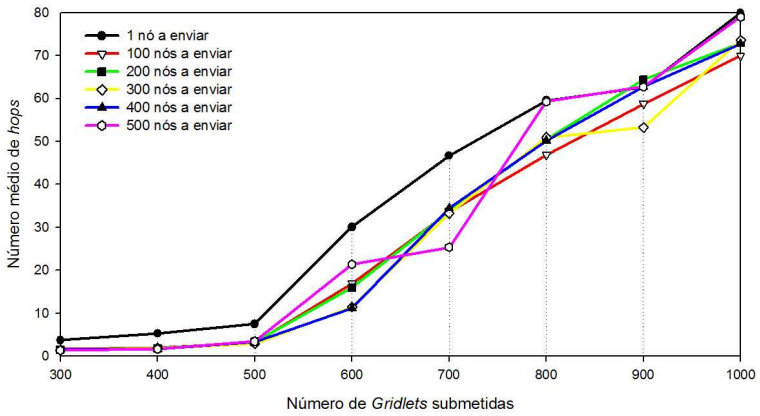

Figura 15: Número médio de *hops* na rede.

Com o envio de 300 e 400 pedidos, verifica-se um número médio de *hops* baixo e semelhante nas simulações com vários nós solicitadores. Este valor médio varia entre 1.5 e 2 *hops*, isto é, o número de retransmissões necessárias para que uma *Gridlet* alcance um nó com disponibilidade, numa rede onde existem 500 nós ociosos. Com 500 pedidos submetidos, estes valores sobem ligeiramente, uma vez que a quantidade de recursos solicitados constitui 100% da disponibilidade da rede. Esta subida deve-se às últimas *Gridlets* que têm maior dificuldade

em encontrar os últimos nós disponíveis na rede, necessitando de mais retransmissões pela rede.

À medida que o número de *Gridlets* submetidas aumenta, desde 600 até 1000 *Gridlets*, o valor de recursos solicitados é superior ao disponível na rede, pelo que esses recursos em excesso vão ser retransmitidos constantemente pela rede até o final do processamento das primeiras que irão libertar recursos. Por esta razão, verifica-se um crescimento acentuado no número médio de *hops* para todas as simulações, à medida que o número de *Gridlets* em excesso aumenta. No entanto, esse crescimento é mais acentuado e variável para algumas simulações, nomeadamente com 500 nós solicitadores em que os resultados são mais instáveis, sendo mais baixos ou mais altos conforme os casos em que as *Gridlets* em excesso, que saltam de nó em nó enquanto não há recursos, estejam perto ou não de nós que ficam disponíveis quando estes acabam o processamento das *Gridlets* anteriores.

Na simulação com apenas um nó a enviar todos os pedidos verifica-se um crescimento e valores médios dos *hops* ligeiramente superiores, até o ponto de saturação, e significativamente superiores, depois do ponto de saturação, em relação às simulações com vários solicitadores. Estes valores devem-se ao facto de as *Gridlets* serem todas transmitidas a partir de um único ponto da rede, sendo esse ponto o epicentro de consumo de recursos. Enquanto todos os nós mais próximos desse epicentro estiverem ocupados com as primeiras *Gridlets*, estes vão ter de reencaminhar as seguintes para nós disponíveis mais distantes, justificando assim os valores superiores no número médio de *hops*.

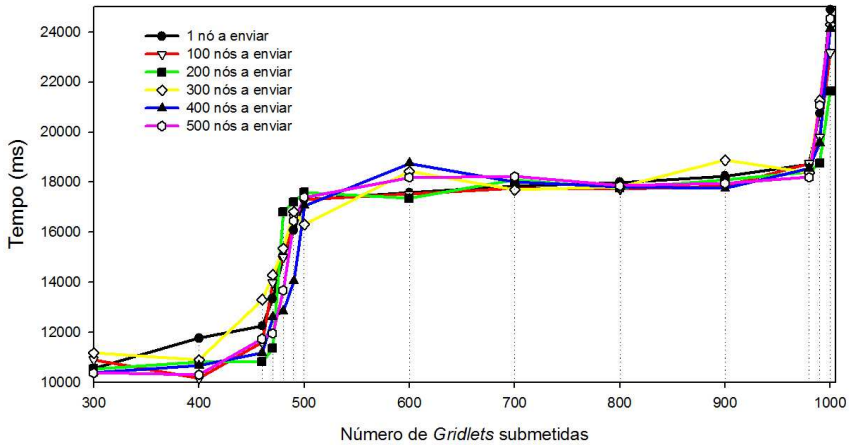

Figura 16: Tempo total dispendido

Através dos tempos de processamento e recolha dos resultados dispendidos em cada simulação, ilustrados no gráfico da Figura 16, é possível identificar os pontos de saturação do sistema, 1º ponto com 500 *Gridlets* submetidas e o 2º com 1000 *Gridlets* submetidas. De forma a obter com maior exactidão estes pontos, foram realizados testes intermédios para o envio de 460, 470, 480 e 490 *Gridlets* (a tracejado no gráfico) para identificação do 1º ponto de saturação, e 980 e 990 *Gridlets* para o 2º ponto. Desta forma pode-se verificar o início da

saturação do sistema, em termos de recursos disponíveis, com o respectivo aumento de tempo das simulações, devido ao início da escassez de recursos por parte da rede.

Número de *Gridlets* que nós serviram	**Número de *Gridlets* submetidas**					
	460	**480**	**500**	**700**	**900**	**1000**
1	460	476	488	300	100	2
2	0	2	6	200	400	496
3	0	0	0	0	0	2

Tabela 5: Quantidade de *Gridlets* servidas por nós que processaram até 3 *Gridlets*.

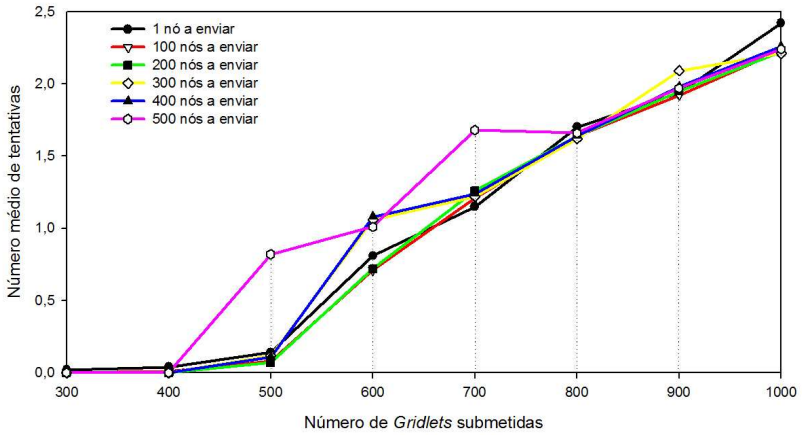

Figura 17: Número médio de tentativas para a recolha dos resultados

Outra propriedade do sistema que é possível auferir através da evolução do gráfico da Figura 16 é a capacidade de finalizar as simulações em tempos semelhantes para quantidades variáveis de *Gridlets* entre dois pontos de saturação. Tal é possível graças ao facto de a rede poder servir 500 *Gridlets* em paralelo, uma vez que existem 500 nós ociosos. No entanto, por vezes, nem sempre 500 *Gridlets* descobrem os 500 nós disponíveis, pelo que essas *Gridlets* são servidas por nós que acabaram de computar uma *Gridlet*, como mostra a Tabela 5. Para o caso de 460 *Gridlets* submetidas, todas as *Gridlets* foram processadas por nós diferentes. Já com 480 *Gridlets*, duas *Gridlets* foram processadas por nós que antes já tinham processado uma. O 2º ponto de saturação verifica-se com 1000 *Gridlets*, onde duas *Gridlets* foram servidas por nós que já tinham previamente servido duas e dois nós serviram apenas uma *Gridlet*.

Os gráficos da Figura 17 e da Figura 18 mostram a relação directa existente entre o número médio de tentativas necessárias para a recolha dos resultados e o tempo médio de processamento de uma *Gridlet*. Quanto maior for o tempo de conclusão de uma *Gridlet*, mais tardia será a inserção do resultado na *cache* do sistema e consequente recolha.

Analogamente aos resultados da Figura 15, o número de mensagens enviadas pela rede na Figura 19 sofreu os mesmos impulsos que o número médio de *hops*. Os factores que influenciam esta tendência devem-se aos maiores contributos para a quantidade de mensagens transmitidas pela rede dada pelas *Gridlets* e pelas mensagens de *update*. Antes do ponto de saturação da rede, o maior contributo é dada pelas mensagens de *update*, devido ao

facto de que cada vez que uma *Gridlet* é processada e consume recursos, as mensagens de *update* são propagadas para todos os vizinhos do nó que está a processar. Após o ponto de saturação, quando o número de *Gridlets* é maior do que a capacidade da rede, o tráfego de *Gridlets* vai ganhando terreno devido às retransmissões pelos nós quando todos os recursos estão alocados. Numa rede destas dimensões, as mensagens do sistema de armazenamento de ficheiros, de *insert* e de *lookup*, têm pouca influência no peso total.

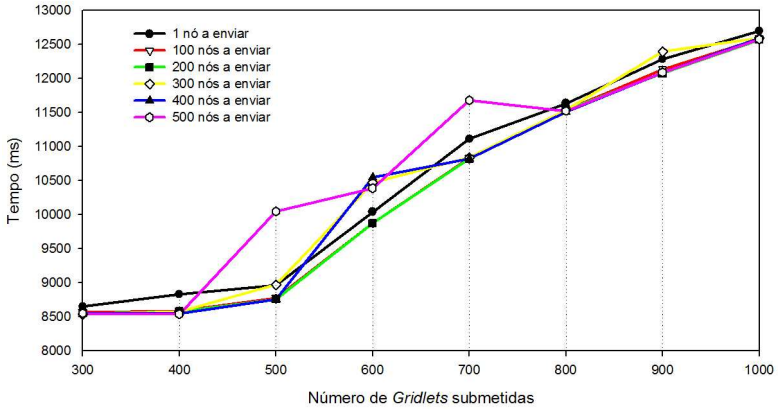

Figura 18: Tempo médio de conclusão de uma *Gridlet*

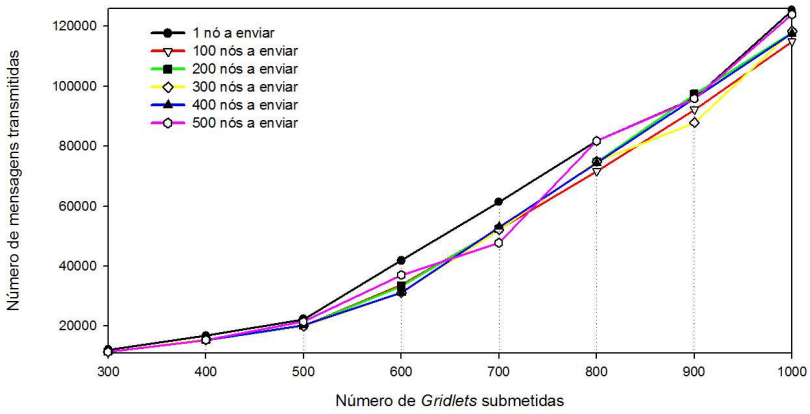

Figura 19: Número de mensagens transmitidas. São contabilizadas as seguintes mensagens: *Gridlets*, *updates*, mensagens de *insert* e de *lookup* do PAST.

5.2.2 Ganhos com os diferentes mecanismos de selecção

De uma forma geral, espera-se que a utilização dos mecanismos de gestão e descoberta de recursos deste sistema traga resultados melhorados. Mas o verdadeiro objectivo passa por descobrir o quão melhor estes mecanismos são, qual a importância da utilização dos mesmos e quais as melhores configurações, caso existam, a escolher.

Para o teste a), sobre a medição da disponibilidade, foram definidas três ponderações sobre o cálculo da disponibilidade conhecida de um nó. O primeiro cálculo avalia apenas a disponibilidade em termos de recursos, distribuindo os pesos do cálculo de igual modo pelas

métricas: 33% para o CPU, 33% para a memória e 33% para a largura de banda. O segundo cálculo avalia apenas a proximidade do nó. E por fim, um último cálculo que pondera as duas medidas, favorecendo os recursos: 40% para a proximidade e 60% para os recursos, repartidos de igual modo, em 20%, para cada métrica.

Na Figura 20 é possível observar a qualidade da descoberta de recursos obtidos pelos três cálculos. Analisando os resultados, é possível constatar que o cálculo a partir apenas dos recursos, para envios de pedidos inferiores ou igual a 500, obteve os piores resultados, mas a partir dos 500 pedidos melhorou, em relação aos outros cálculos. De notar que até o ponto de saturação (500) não existe falta de recursos, pelo que a partir desse ponto, a falta de recursos é uma constante. Assim sendo, podemos inferir que o cálculo baseado nos recursos é favorável para situações de solicitação em excesso de recursos. Já o cálculo baseado na proximidade dos nós, é favorável enquanto existirem muitos recursos disponíveis na rede e muito desfavorável em situações de inexistência imediata de recursos, já que é a partir de 700 *Gridlets* submetidas que obtém o maior crescimento para piores resultados.

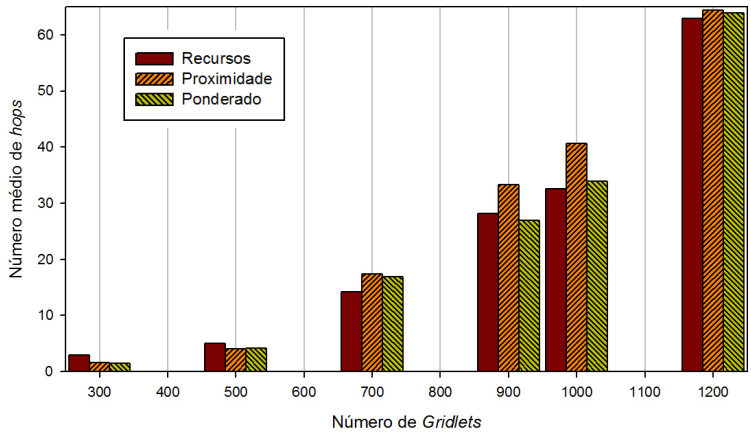

Figura 20: Número médio de *hops* obtido com a)

Em situações de enorme procura de recursos, os nós que escolherem nós com a maior disponibilidade vão ter maiores probabilidades de serem bem sucedidos, já que enviar um pedido para um nó com capacidade para tratar apenas uma *Gridlet*, poderá ocorrer que outro nó também lhe tenha enviado um pedido e consequentemente, quem chegar primeiro irá ser o escolhido, retransmitindo o outro. Escolhendo o nó com maior disponibilidade irá permitir ter mais hipóteses de tratamento de um pedido, bem como uma maior probabilidade de esse nó, se estiver completamente ocupado, acabar uma tarefa e poder tratar logo o pedido. Neste tipo de situações, o grau de proximidade entre os nós não influencia em nada a capacidade de um nó tratar ou não um pedido, e em alguns casos, poderá até dificultar a descoberta dos recursos, caso os únicos disponíveis se encontrem em nós distantes.

Numa situação de abundante disponibilidade, o cálculo baseado em recursos é indiferente. Aí, já o cálculo baseado na proximidade ganha eficiência, uma vez que escolhe os nós disponíveis mais próximos.

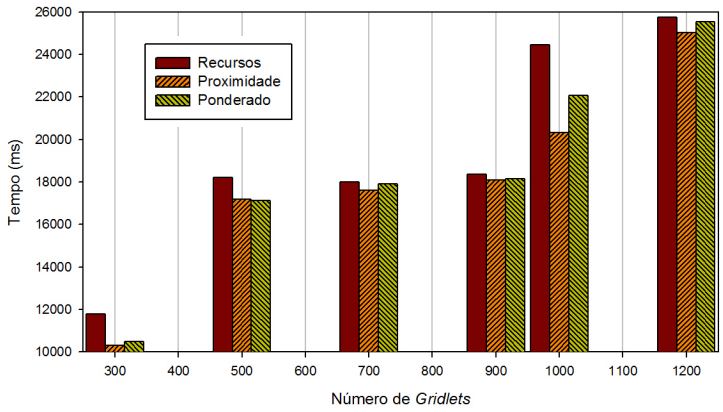

Figura 21: Tempo total dispendido com a)

Em termos do tempo necessário à conclusão das 700 *Gridlets*, é possível verificar na Figura 21 que o cálculo por recursos consome muito tempo em todas as situações. Mesmo nas situações de falta de recursos, em que este cálculo favorece a eficiência (ver Figura 20), tem o custo de um maior consumo de tempo. Este custo relativo ao tempo advém do erro de escolha devido ao favorecimento de nós com mais recursos mas também mais distantes geograficamente (maior latência). Já o cálculo por proximidade, como se esperava, tem os melhores tempos em todas as situações, uma vez que dá primazia à selecção dos nós mais próximos. Com o cálculo ponderado obtemos valores intermédios em relação aos dois cálculos anteriores. Com maior utilização dos recursos a decisão ponderada também produz os menores tempos.

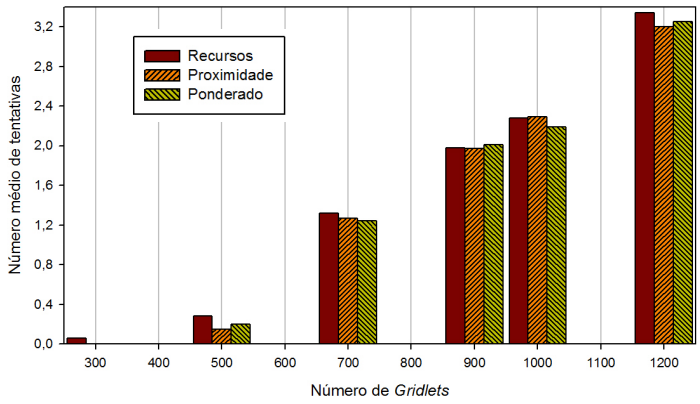

Figura 22: Número médio de tentativas para a recolha dos resultados com a)

Por consequência, ao tempo necessário para a descoberta dos nós com maior disponibilidade, resulta a produção tardia dos resultados dos pedidos e a inserção dos mesmos na cache, pelo que aumenta o número de tentativas necessárias para a recolha dos resultados, como se pode verificar na Figura 22, para o cálculo baseado nos recursos. Apesar de o cálculo baseado apenas na proximidade apresentar um baixo número de tentativas na recolha, para todas as situações, graças à sua rapidez em escolher nós para tratarem os pedidos, o cálculo

ponderado compete com esses valores, superando-o por vezes com menor número de tentativas.

Desta feita, temos o cálculo ponderado entre os recursos e a proximidade (a favorecer os recursos), a apresentar os melhores resultados na maioria dos casos, já que aproveita o melhor das duas medidas, para as situações com e sem disponibilidade. Podemos assim concluir que, o cálculo ideal seria ajustar dinamicamente, de acordo com a disponibilidade presente na rede, os valores dos pesos sobre os recursos e a proximidade. Quando a rede abunda em recursos, deverá ser dado um maior ou total peso à proximidade, usufruindo da sua rápida conclusão dos pedidos em detrimento do tempo perdido na descoberta de nós com maior disponibilidade pela selecção baseada em recursos. Para situações de escassez ou falta de recursos na rede, deverá atribuir-se maior peso aos recursos e um peso significativo à proximidade, aproveitando a selecção sobre os nós com mais recursos e tirando partido do factor tempo na selecção pela proximidade. O principal problema prende-se na forma de obter o estado da disponibilidade na rede. Na melhor das hipóteses, cada nó conhecerá apenas com a disponibilidade da sua vizinhança.

Para o teste b), sobre a utilização de pré-redução, foram efectuados dois tipos de simulações, a primeira executando o sistema normalmente, recorrendo aos métodos especificados para a gestão e descoberta de recursos, e um segundo em que se executa o sistema com os mesmos métodos, exceptuando-se o método de pré-redução dos valores conhecidos de um nó vizinhos, quando esse nó é escolhido para o encaminhamento como tendo disponibilidade para realizar a tarefa submetida.

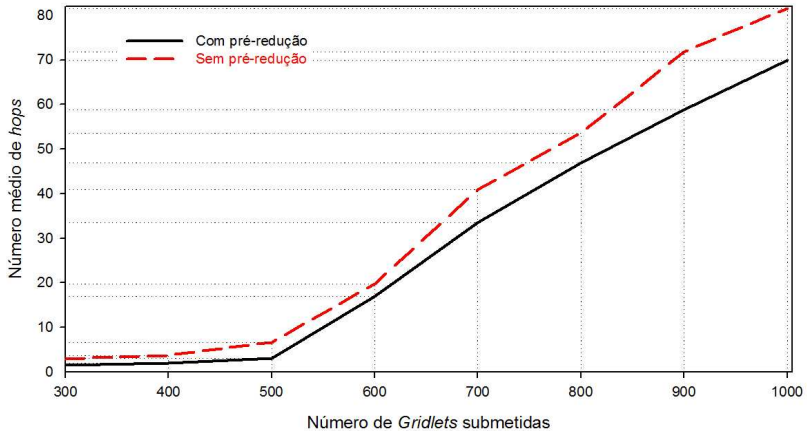

Figura 23: Número médio de *hops* com b)

A Figura 23 mostra o ganho da utilização de pré-redução. À medida que a demanda de recursos na rede aumenta, o proveito, em termos de eficiência, da utilização deste mecanismo é mais visível, pois o número de retransmissões pelos nós com a utilização de pré-redução é constantemente inferior e à medida que o número de pedidos submetidos aumenta, a diferença é cada vez maior. Analogamente, observando o gráfico da Figura 24 é possível verificar que o uso de pré-redução origina um fluxo inferior de mensagens pela rede, diminuindo as

transmissões. Em grande parte, estes valores baixos devem-se à diferença das quantidades de *Grildlets* que são retransmitidas entre os nós verificados na Figura 23.

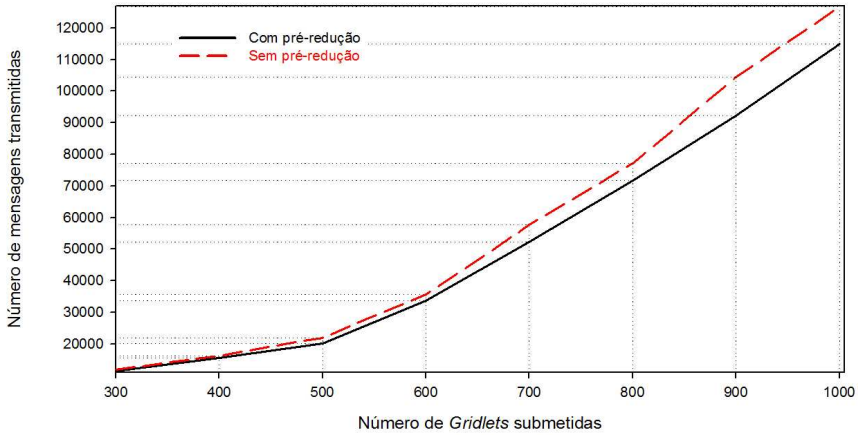

Figura 24: Número de mensagens transmitidas pela rede com b). São contabilizadas as seguintes mensagens: *Gridlets*, *updates*, mensagens de *insert* e de *lookup* do PAST.

Relativamente ao tempo de conclusão dos pedidos e o número de tentativas necessárias para a recolha dos resultados, não se verificaram diferenças significativas entre a utilização e a não utilização de pré-redução.

Finalmente, no teste c), sobre o sistema de reputação, a simulação é executada sempre na mesma rede e os pedidos são todos originados a partir do mesmo nó. A taxa de pedidos é sempre a mesma, 700 *Gridlets*, de forma a testar o comportamento do sistema quando há falta de recursos na rede. Só para estas situações é que o mecanismo alvo neste teste, a reputação dos nós, pode operar de forma relevante e influenciar os resultados.

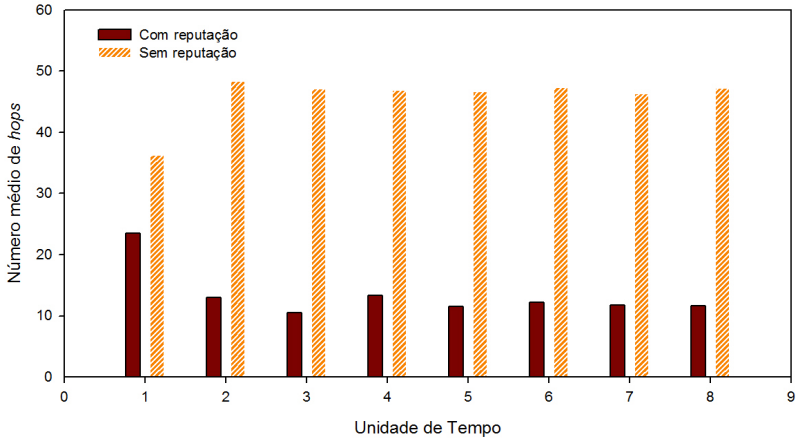

Figura 25: Número médio de *hops* com c)

De acordo com a Figura 25, é possível observar a melhor qualidade de encaminhamento obtida utilizando o mecanismo de reputação. A menor diferença ocorre durante a primeira iteração, uma vez que com o sistema de reputação, o nó de origem ainda não adquiriu informações de reputação sobre os seus vizinhos. Na segunda iteração é já visível uma grande redução no

número de retransmissões efectuadas, mantendo-se a esse nível a partir desse ponto. Assim, podemos afirmar que este mecanismo converge muito rapidamente para o seu melhor desempenho. De notar, que este mecanismo prova melhorar o encaminhamento, mas ao contrário do que foi verificado no mecanismo de pré-redução, descrito em b), a diferença do seu uso é substancial.

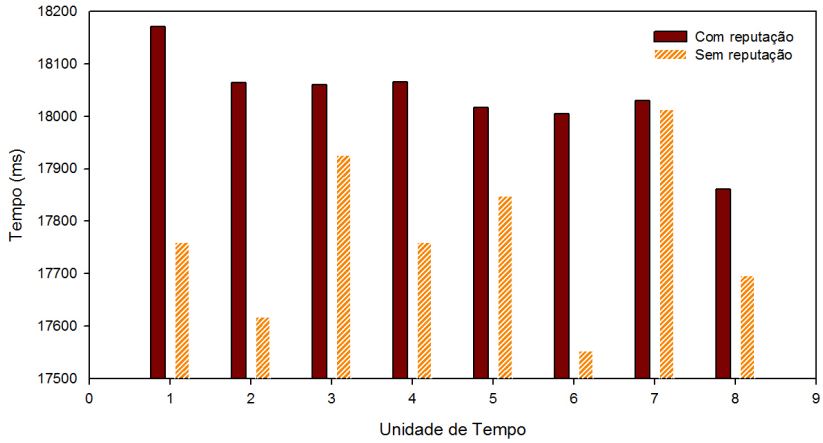

Figura 26: Tempo dispendido em cada simulação com c)

Contudo, a grande eficiência de encaminhamento com reputação perde para o encaminhamento sem reputação no que diz respeito ao tempo necessário para a conclusão dos pedidos (ver Figura 26). Mas é importante referir que a diferença desses tempos é muito baixa, por exemplo, para a primeira iteração, a diferença é apenas de 400 milissegundos.

5.2.3 Escalabilidade

Este teste pode ser considerado como um dos mais importantes testes à funcionalidade do sistema, já que permite testar a escalabilidade do mesmo e é capaz de testar os mecanismos de gestão e descoberta de recursos para os piores cenários, no que diz respeito à entrada de novos nós e novos recursos a serem mapeados, bem como a redefinição dos conjuntos das vizinhanças e também a saída de nós que leva à interpretação de informações erradas, relativas a acontecimentos anteriores.

A Figura 27 mostra a progressão do tamanho da rede, crescente até 2000 nós e decrescente a partir desse ponto até os 200 nós iniciais. Até ao pico da rede, 1000 nós, verifica-se um aumento, na ordem de 2 e 3 *hops*, do número de retransmissões dos pedidos entre os nós, o que pode ser declarado como o crescimento normal devido ao facto de a rede aumentar o seu número de nós. Mas com a redução do tamanho da rede, é possível verificar-se um certo abrandamento na diminuição do número de retransmissões, em particular quando a rede é cada vez mais pequena.

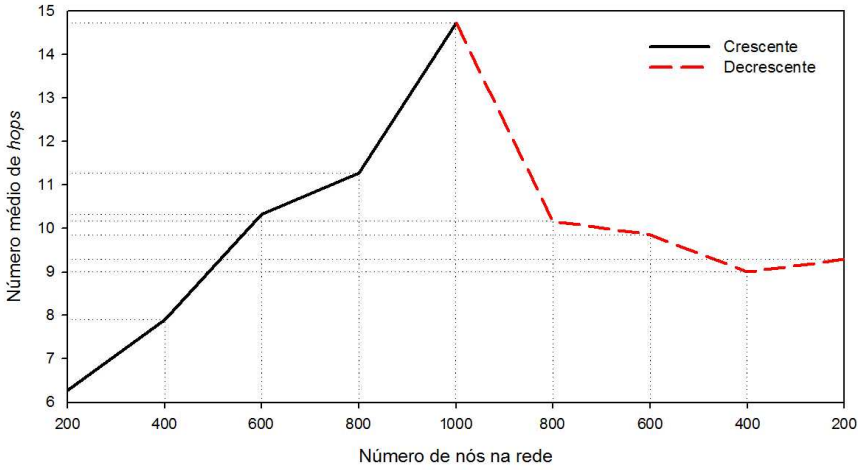

Figura 27: Número médio de *hops*

Este sistema apresenta resultados favoráveis à medida que o tamanho da rede aumenta, já que os seus mecanismos de gestão e descoberta de recursos detectam facilmente alterações na vizinhança e actualizam as suas tabelas de imediato, a partir das mensagens de *update* enviadas pelos novos nós quando estes entram na rede. Para além disso, o sistema de reputação converge rapidamente para o seu melhor desempenho, como vimos no teste anterior. No entanto, quando saem nós da rede, o sistema tem dificuldade em adaptar-se aos restantes nós, uma vez que durante algum tempo são ainda tomadas decisões erradas com base em informações de reputação sobre nós que já deixaram a rede. Este fenómeno é mais notável quando a dimensão da rede é menor (200 e 400 nós), em que os poucos nós restantes têm de cooperar entre si para realizar a tarefa de irradiação da informação de reputação desactualizada. Nestas situações, a utilização da informação de resultados passados é tão ou mais prejudicial do que a ausência de qualquer informação.

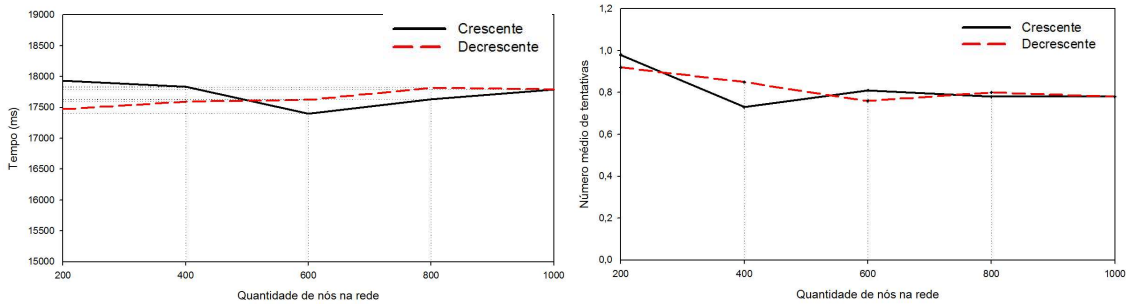

Figura 28: Tempo total dispendido (à esquerda) e número de tentativas necessárias para a recolha dos resultados (à direita).

Em termos do tempo de conclusão dos pedidos, não há nada a assinalar, uma vez que só se notaram diferenças mínimas nos valores, na ordem dos 500 milissegundos de diferença, tal como o número de tentativas para a recolha dos resultados onde a maior diferença é verificada com um valor de 0,25, como se pode constatar nos gráficos da Figura 28.

6. Conclusões

À medida que o desenvolvimento das capacidades dos computadores e de novos dispositivos com recursos de processamento significativos e ligações de rede vão crescendo, é cada vez mais vantajosa a exploração de recursos ociosos das máquinas. Para além disso, tendo em conta a velocidade com que novas aplicações chegam ao mundo para se tornarem acessórios essenciais no dia-a-dia, é uma mais-valia a existência de uma plataforma que explore o paralelismo das execuções sem que sejam necessárias alterações nessas aplicações.

A solução apresentada nesta dissertação refere-se a um modelo de programação baseado no conceito de *gridlets*, capaz de estabelecer uma síntese entre infra-estruturas de *Grids* Institucionais, partilha distribuída de recursos e arquitecturas P2P descentralizadas. Foi desenhada uma arquitectura para uma plataforma de *middleware* capaz de explorar diferentes topologias de uma rede P2P, com base em métricas de desempenho, de forma a poder avaliar vários critérios para a descoberta de recursos na rede. A noção do conceito de *Gridlet* neste projecto permite a distribuição, pelos nós da rede, de partes da execução de uma aplicação para serem processadas remotamente noutras máquinas. Para além disso, a solução contribui com um simulador capaz de gerar variadas condições de rede e solicitações de recursos, monitorizando todos os fluxos das transmissões entre os nós do *overlay*.

A grande realização deste trabalho prende-se ao facto de aplicações populares poderem melhorar, de forma transparente, o seu desempenho, através do paralelismo da execução das suas tarefas utilizando ciclos de processamento excedentários de outras máquinas pertencentes ao mesmo *overlay*. Ao contrário de abordagens anteriores nesta área, a solução proposta permitiu conciliar com sucesso as tecnologias acima mencionadas, permitindo a exploração de recursos ociosos na rede por parte de utilizadores comuns com programas genéricos, sem que estes necessitam de qualquer tipo de modificação nem utilização de API, bibliotecas, ou linguagens de programação específicas.

A execução de testes no sistema elaborado permitiu verificar o sucesso das funcionalidades propostas (descoberta de recursos e eficiente encaminhamento das *Gridlets*) na exploração da topologia da rede com base em métricas de desempenho (*cpu*, memória, largura de banda e proximidade). A utilização de mecanismos de reputação de classificação dos nós permitiu também alcançar resultados mais eficientes no que diz respeito ao encaminhamento das *Gridlets*. Além disso, foram testadas e cumpridas com sucesso execuções de solicitação de pedidos sobre condições extremas na rede, em termos de escalabilidade e disponibilidade de recursos na rede.

6.1 Trabalho Futuro

Uma vez que o futuro aponta para uma maior ligação entre as máquinas e outros tipos de dispositivos, a atenção ao nível da segurança não deverá ser negligenciada para este tipo de plataformas. Desde a protecção dos dados partilhados pela rede até ao controlo dos acessos aos recursos por parte de outras máquinas, será necessário precaver o sistema de uma panóplia de ameaças que existem nas redes nos dias de hoje.

Uma funcionalidade que não foi implementada no sistema e que seria uma mais-valia para o funcionamento do mesmo no mundo real seria um sistema de replicação. A replicação dos pedidos solicitados na rede guardados num sistema de cache facilitariam o acesso aos mesmos, evitariam o processamento repetido das mesmas tarefas e permitiriam a sua validação e descarte de resultados falsificados. Esta funcionalidade não foi implementada devido a problemas no sistema de armazenamento distribuído utilizado para o efeito, o PAST. O problema surge sempre que existem entradas de novos nós, em que o alcance (espaço de identificadores) da rede altera-se e o gestor de réplicas do PAST é activado para recolocar as réplicas guardadas na cache da rede. O problema reside no facto de o gestor de réplicas actuar cedo demais quando os novos nós não estão ainda completamente iniciados (nomeadamente a instância do PAST desses nós).

Nos testes realizados foi possível verificar diferentes desempenhos para diferentes valores nos pesos no cálculo dos critérios de selecção. O estudo do estado da rede e consequente configuração de variáveis em tempo de execução poderia melhorar o desempenho em condições anormais para os valores definidos. Essas variáveis englobam o factor de replicação, os valores médios dos *lookups* e de proximidade incluídos no cálculo dos tempos de espera para as tentativas de recolha dos resultados, bem como os pesos atribuídos ao cálculo das métricas de desempenho são exemplos de alterações que gerariam variações significativas nos resultados finais.

Uma simulação mais completa e realista do sistema pode ser obtida através de testes num *cluster* de máquinas ou em máquinas com processadores *multi-core*. A utilização de um ambiente *multi-threading* assim o permite, tal como o facto de o simulador ser discreto e orientado a eventos. Novos testes deste tipo poderiam dar novas perspectivas sobre as configurações a ter em conta no trabalho, bem como situações particulares que se poderiam verificar, como por exemplo cenários onde a escassez de um determinado recurso em particular pode influenciar o comportamento global do sistema.

Referências

[1] D.P. Anderson et al., “SETI@ home: an experiment in public-resource computing,” *Communications of the ACM*, vol. 45, 2002, pp. 56-61.

[2] “Bittorrent accounts for 35% of Internet tra□c,” *Slashdot article referring to an Internet tra□c study*, Nov. 2004.

[3] I. Stoica et al., “Chord: A scalable peer-to-peer lookup service for Internet applications,” *Proceedings of the 2001 SIGCOMM conference*, vol. 31, 2001, pp. 149-160.

[4] A. Rowstron and P. Druschel, “Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems,” *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, vol. 11, 2001, pp. 329-350.

[5] I. Foster and C. Kesselman, “Globus: a Metacomputing Infrastructure Toolkit,” *International Journal of High Performance Computing Applications*, vol. 11, 1997, p. 115.

[6] Y. Chawathe et al., “Making Gnutella-like P2P Systems Scalable.”

[7] R. Dingledine, M.J. Freedman, and D. Molnar, “The Free Haven Project: Distributed Anonymous Storage Service,” *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, USA, July 25-26, 2000: Proceedings*, 2001.

[8] J. Liang, R. Kumar, and K.W. Ross, “Understanding KaZaA,” *available at*.

[9] S. Androutsellis-Theotokis and D. Spinellis, “A survey of peer-to-peer content distribution technologies,” *ACM Computing Surveys (CSUR)*, vol. 36, 2004, pp. 335-371.

[10] I. Clarke et al., “Freenet: A Distributed Anonymous Information Storage and Retrieval System,” *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, USA, July 25-26, 2000: Proceedings*, 2001.

[11] M. Waldman, A.D. Rubin, and L.F. Cranor, “Publius: A robust, tamper-evident, censorship-resistant,” *Proc. 9th USENIX Security Symposium*, 2000, pp. 59-72.

[12] D.S. Milojicic et al., “Peer-to-Peer Computing,” *HP Laboratories Palo Alto, March*, 2002.

[13] A. Barak and R. Wheeler, “MOSIX: an integrated multiprocessor UNIX,” *Mobility: processes, computers, and agents table of contents*, 1999, pp. 41-53.

[14] M.J. Litzkow, M. Livny, and M.W. Mutka, “Condor-a hunter of idle workstations,” *Distributed Computing Systems, 1988., 8th International Conference on*, 1988, pp. 104-111.

[15] S. Ratnasamy et al., “A Scalable Content-Addressable Network (CAN),” *Proc. of ACM SIGCOMM*, 2001.

[16] D. Andersen et al., *Resilient overlay networks*, ACM Press New York, NY, USA, 2001.

[17] I. Clarke et al., “Protecting free expression online with Freenet,” *Internet Computing, IEEE*, vol. 6, 2002, pp. 40-49.

[18]“Napster”; http://ntrg.cs.tcd.ie/undergrad/4ba2.02-03/p4.html.

[19] A. Bosselaers, R. Govaerts, and J. Vandewalle, “SHA: a design for parallel architectures,” *Advances in Cryptology, Proceedings Eurocrypt’97, LNCS*, vol. 1233, pp. 348-362.

[20] M. Castro et al., “Topology-aware routing in structured peer-to-peer overlay networks,” *Proc. Intl. Workshop on Future Directions in Distrib. Computing (FuDiCo 2003)*, 2003, p. 103–107.

[21]“Dr. Rajkumar Buyya”; http://www.gridcomputing.com/gridfaq.html.

[22] I. Foster, C. Kesselman, and S. Tbecke, “The anatomy of the Grid,” *Grid Computing: Making the Global Infrastructure a Reality*, 2003.

[23] W. Cirne et al., “Running Bag-of-Tasks applications on computational grids: the MyGrid approach,” *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, 2003, pp. 407-416.

[24] D.P. da Silva, W. Cirne, and F.V. Brasileiro, “Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids,” *Euro-Par 2003 Parallel Processing: 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003: Proceedings*, 2003.

[25] E. Santos-Neto et al., “Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids,” *Job Scheduling Strategies for Parallel Processing: 10th International Workshop, JSSPP 2004, New York, NY, USA, June 13, 2004: Revised Selected Papers*, 2005.

[26] N. Andrade et al., “OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing,” *Job Scheduling Strategies for Parallel Processing: 9th International Workshop, Jsspp 2003, Seattle, Wa, Usa, June 24, 2003: Revised Papers*, 2003.

[27] P. Barham et al., “Xen and the art of virtualization,” *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 164-177.

[28] N. Andrade, “Francisco Brasileiro, Walfredo Cirne, Miranda Mowbray, Discouraging free-riding in a Peer-to-Peer grid,” *IEEE International Symposium on High-Performance Distributed Computing*, 2004.

[29] N. Andrade et al., “Peer-to-peer grid computing with the ourgrid community,” *Proceedings of the 23rd Brazilian Symposium on Computer Networks*, 2005.

[30] S. Shostak, “Sharing the universe- Perspectives on extraterrestrial life,” *Berkeley, CA: Berkeley Hills Books, 1998.*, 1998.

[31] M.J. Litzkow, “Remote UNIX: Turning Idle Workstations into Cycle Servers,” *Proceedings of Usenix Summer Conference*, 1987, pp. 381-384.

[32] R. Raman, M. Livny, and M. Solomon, “Matchmaking: Distributed Resource Management for High Throughput Computing,” *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, vol. 146, 1998.

[33] M. Litzkow et al., *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, Technical Report, 1997.

[34] D. Wright, “Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor,” *Conference on Linux Clusters: The HPC Revolution*, 2001.

[35] D.P. Anderson, “BOINC: A System for Public-Resource Computing and Storage,” *5th IEEE/ACM International Workshop on Grid Computing*, 2004, pp. 365-372.

[36] V. Lo et al., “Cluster Computing on the Fly: P2P Scheduling of Idle Cycles in the Internet,” *Peer-To-Peer Systems III: Third International Workshop, IPTPS 2004, La Jolla, CA, USA, February 26-27, 2004: Revised Selected Papers*, 2004.

[37] D. Zhou and V. Lo, “Cluster Computing on the Fly: resource discovery in a cycle sharing peer-to-peer system,” *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, 2004, pp. 66-73.

[38] L. Oliveira, L. Lopes, and F. Silva, “P 3: Parallel Peer to Peer,” *Revised Papers from the NETWORKING 2002 Workshops on Web Engineering and Peer-to-Peer Computing*, p. 274–288.

[39] I. Foster and A. Iamnitchi, “On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing,” *2nd International Workshop on Peer-to-Peer Systems (IPTPS’03)*, 2003, p. 118–128.

[40]“Limewire. www.limewire.com.”

[41]“DZero Experiment. www-d0.fnal.gov.”

[42] I. Forster and C. Kesselman, “The Grid: Blueprint for a New Computing Infrastructure,” *Morgan Kaufiuiann, San Francisco, CA*, vol. 211, 1999.

[43] M. Ripeanu, A. Iamnitchi, and I. Foster, “Performance Predictions for a Numerical Relativity Package in Grid Environments,” *International Journal of High Performance Computing Applications*, vol. 15, 2001, p. 375.

[44] B.Y. Zhao, J. Kubiatowicz, and A.D. Joseph, “Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing,” *Computer*, vol. 74, 2001.

[45] I. Foster and C. Kesselman, “Globus: A Toolkit-Based Grid Architecture,” *The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann*, vol. 259, 1999, p. 278.

[46] F. Cappello et al., “Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid,” *Future Generation Computer Systems*, vol. 21, 2005, pp. 417-437.

[47] A.A. Chien, S. Marlin, and S.T. Elbert, “Resource management in the Entropia system,” *Grid Resource Management: state of the art and future trends. Kluwer Academic Publishers, Norwell*, 2004.

[48]“SETI@home, ”http://setiathome.ssl.berkeley.edu”.”

[49]“Distributed.net, ”http://distributed.net”.”

[50] D. Thain, T. Tannenbaum, and M. Livny, “Condor and the Grid,” *Grid Computing: Making the Global Infrastructure a Reality*, 2003.

[51] L. Veiga, R. Rodrigues, and P. Ferreira, “GiGi: An Ocean of Gridlets on a" Grid-for-the-Masses"," *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, 2007, pp. 783-788.

[52] Filali, I., F. Huet, e C. Vergoni. “A Simple Cache Based Mechanism for Peer to Peer Resource Discovery in Grid Environments.” *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)-Volume 00* (2008): 602-608.

[53] “Pastry - A scalable, decentralized, self-organizing and fault-tolerant substrate for peer-to-peer applications.” http://freepastry.org/.

[54] Rowstron, A., e P. Druschel. “Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility.” *ACM SIGOPS Operating Systems Review* 35, no. 5 (2001): 188-201.

[55] Pietzuch, P., J. Ledlie, M. Mitzenmacher, e M. Seltzer. “Network-Aware Overlays with Network Coordinates.” *Proceedings of the Workshops of the 26th International Conference on Distributed Computing Systems*.