Hermes: um arcabouço para a programação de aplicações P2P

Emílio de Camargo Francesquini

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA
OBTENÇÃO DO TÍTULO DE MESTRE
EM
CIÊNCIAS

Área de concentração: Ciência da Computação
Orientador: Prof. Dr. Francisco Carlos da Rocha Reverbel

São Paulo, maio de 2007.

Hermes: um arcabouço para a programação de aplicações P2P

Este exemplar corresponde à redação
final da dissertação devidamente
corrigida e defendida por
Emílio de Camargo Francesquini
e aprovada pela Comissão Julgadora.

São Paulo, 09 de maio de 2007.

Banca Examinadora:
Prof. Dr. Francisco Carlos da Rocha Reverbel (Orientador) - IME/USP
Prof. Dr. Fabio Kon - IME/USP
Profa. Dra. Graça Bressan - POLI/USP

Aos meus pais

Resumo

Hermes é um arcabouço para a programação de aplicações P2P. Com ele, pode-se criar diversos tipos de aplicações distribuídas, sem se preocupar com a camada de comunicação. O Hermes não é uma implementação de uma rede de sobreposição P2P, e sim uma camada acima das implementações já existentes. O desenvolvedor da aplicação fica isolado da implementação da rede de sobreposição utilizada. Esse isolamento é feito de forma tal que não há limitações quanto à arquitetura de rede utilizada pela implementação, seja ela centralizada, descentralizada, distribuída estruturada ou distribuída não-estruturada. Entre os serviços oferecidos pelo Hermes estão: troca de mensagens, busca, comunicação em grupo e armazenamento distribuído.

Geralmente, no início do desenvolvimento de uma aplicação distribuída, tem-se poucas informações sobre o seu tamanho final ou perfil de utilização. O Hermes possibilita ao desenvolvedor da aplicação adiar, até o momento da efetiva implantação do sistema, a decisão sobre qual arquitetura de rede ou qual implementação de rede de sobreposição são as mais apropriadas para suas necessidades. Possibilita também, quando o perfil de utilização muda com o tempo, a troca da implementação utilizada por uma outra que se adeque mais ao novo perfil sem alterações no código da aplicação.

Abstract

Hermes is a framework for P2P application programming. Using it, one can create several kinds of distributed applications without worrying about the underlying network. Hermes is not a P2P overlay network implementation, but a shell envolving existing implementations. The application developer is isolated from the implementation of the overlay network in use. This isolation is done in a way that poses no limitations on the network architecture used, which may be centralized, decentralized, structured or unstructured. Amongst the services offered by Hermes are: message exchange, search, group communication, and distributed storage.

In the early stages of the development of a distributed application, information as to its final size or utilization profile is often unknown. Hermes gives the application developer the possibility of delaying, until the actual moment of system deployment, the decision as to which network architecture or which overlay network implementation is the most appropriate. It also gives the developer the choice, when utilization profile changes over time, of replacing the network implementation with one more suitable to the application needs, without changes on the application code.

Sumário

Agradecimentos

É uma satisfação poder escrever estes agradecimentos àqueles que de alguma forma contribuíram para tornar possível a realização deste trabalho. Todos os produtos deste resultam da minha atuação em conjunto com essas pessoas.

Primeiramente gostaria de agradecer aos meus pais, Sueli e Fernando. Certamente não teria conseguido chegar até aqui se não fosse pela constante ajuda e apoio incondicional deles. Não acredito que haja uma maneira de agradecer por todas as coisas que lhes são devidas. Eu só posso esperar que, um dia, possa ser tão bom para alguém assim como eles foram para mim.

Também gostaria de agradecer aos meus irmãos, Fernanda e Felipe, e aos demais membros da minha família. O crédito que por eles me foi dado, juntamente com a sua torcida e incentivo, foram essenciais.

Agradeço a todos os meus amigos. Eles tiveram uma participação ímpar durante a elaboração deste trabalho. É suficiente dizer que, sem o apoio deles, eu não teria seguido adiante, e perseguido este objetivo. É impossível citar o nome de todos aqueles que me ajudaram neste período. Ainda assim, gostaria de deixar um agradecimento especial à Carla, à Flávia, à Sandra, ao Danilo, ao Gobetti, ao Júnior e ao Wagner. Estes me mostraram o que realmente significa a amizade.

Agradecimentos especiais, também, ao Alessandro M. de Camargo, Alexandre E. P. de A. Gabriel e ao Marcelo P. Barbosa pela leitura e sugestões feitas nas versões preliminares deste texto.

Agradeço ao meu orientador Prof. Dr. Francisco Carlos da Rocha Reverbel pela sua confiança. Esta foi depositada em mim desde o momento da escolha do assunto da dissertação. Isso fez com que o trabalho seguisse de uma forma fluida e tranqüila além de permitir que eu encontrasse o meu próprio caminho.

1 Introdução

O termo *peer-to-peer*[1] (P2P) descreve um conjunto de conceitos e mecanismos para computação distribuída e comunicação direta entre os nós participantes da rede. Ele não representa uma tecnologia, arquitetura ou sistema em particular. Em uma rede P2P pura, nenhum nó é considerado diferente dos demais, ou seja, não existem servidores centrais para cuidar das requisições dos clientes [41, 48, 52]. De forma geral, uma rede P2P é uma rede que possui três características: auto-organização (adaptação automática à entrada e saída, voluntária ou por falha, de nós), comunicação simétrica e controle distribuído [61].

Apesar desses conceitos já existirem por muito tempo (a Internet já os utilizava desde a sua criação como, por exemplo, nos protocolos SMTP, para envio de e-mail, e NNTP, para *Usenet News*), foi o surgimento de algumas aplicações e sistemas P2P que acabou fazendo com que eles se difundissem e se tornassem largamente conhecidos. Por este motivo, o termo P2P é freqüentemente associado às aplicações e não aos conceitos técnicos que ele realmente representa.

Existem vários sistemas P2P implementados para os mais diversos fins. Alguns exemplos de utilização são:

Compartilhamento de arquivos Este é o caso de uso mais difundido tanto que o termo *"programa P2P"* fora do meio acadêmico se transformou praticamente em um sinônimo para programas de compartilhamento de arquivos. Exemplos deste tipo de programa são: o Gnutella[2], Napster[3], KaZaA[4], Freenet [19] e E-Donkey[5]. O uso mais comum para este tipo de programa é o compartilhamento de arquivos de música, vídeo e software. Um dos potenciais usos de sistemas P2P que tem sido origem de discussões e polêmicas é a obtenção de produtos (músicas, filmes, programas, jogos, entre outros) de forma gratuita e ilegal. Há também relatos de utilização de sistemas como o Freenet, que pro-

[1] A palavra inglesa *peer* se refere a alguém que está a mesma altura de alguma outra pessoa no que diz respeito a algum círculo social baseado no grau de instrução ou posição financeira. Palavras semelhantes em português são companheiro, colega ou par. Neste texto utilizaremos tanto o termo *peer*, já consagrado na literatura, quanto o termo *nó* para referenciar cada um dos participantes de um sistema P2P.

[2] `http://www9.limewire.com/developer/gnutella\_protocol\_0.4.pdf`

[3] `http://www.napster.com`

[4] `http://www.kazaa.com`

[5] Página do E-donkey `http://www.edonkey2000.com` e da implementação aberta alternativa, o eMule `http://www.emule-project.net/`

mete anonimato e, portanto, liberdade de expressão [19], para obtenção e distribuição de informações censuradas ou ilícitas. Diversas disputas judiciais para a resolução desses tipos de problemas ocorreram. Dessas, o caso mais conhecido é o do Napster, que em 2001 foi obrigado a interromper o seu funcionamento quando a justiça assim o determinou[6].

Compartilhamento de recursos/Computação distribuída Existem diversos sistemas que se utilizam deste tipo de rede para o processamento ou armazenamento de informações. A idéia é aproveitar melhor os recursos que normalmente seriam desperdiçados como, por exemplo, o tempo ocioso de processamento ou espaço de armazenamento. Grande parte destes sistemas utiliza uma arquitetura centralizada ou descentralizada. Exemplos disto são SETI@Home[7] (centralizado) e Integrade [26,62] (descentralizado).

Outros, entretanto, utilizam-se de uma arquitetura totalmente distribuída como o Freenet[8] e o OceanStore [38].

Comunicação instantânea/Colaboração de grupos Com mais de 200 milhões de usuários em todo o mundo [51], os sistemas de comunicação instantânea (*Instant Messaging - IM*) como ICQ[9], MSN Messenger[10], AOL Instant Messenger[11] e Skype[12] são, com uma grande vantagem, os maiores sistemas P2P em uso. Sistemas deste tipo são exemplos claros da fuga do paradigma cliente/servidor: o fluxo de mensagens é feito entre os clientes diretamente, sem interferência de um terceiro participante.

Mecanismos de busca distribuídos Em uma rede como a Internet, onde grande parte da informação disponível está concentrada em servidores Web públicos, a procura por informações pode ser feita

[6] *Top 10 Internet Law Developments in 2001* - `http://www.skadden.com/content/publications/785library.pdf`

[7] `http://setiathome.ssl.berkeley.edu/`

[8] Pode parecer estranho, à primeira vista, que o Freenet esteja classificado tanto como um sistema de compartilhamento de arquivos quanto como um sistema de compartilhamento de recursos. De fato a arquitetura do Freenet foi criada de uma maneira diferente dos demais sistemas de compartilhamento de arquivos. No Freenet, qualquer arquivo publicado é armazenado de forma não centralizada na rede, espalhado pelos nós participantes e não no nó que o publicou.

[9] `http://www.icq.com`

[10] `http://messenger.msn.com`

[11] `http://www.aim.com`

[12] `http://www.skype.com/`

facilmente através de *spiders* [66] que varrem as páginas para a construção de um índice do seu conteúdo. O Google[13] e o Altavista[14] são exemplos de serviços que se utilizam deste tipo de mecanismo. Entretanto, conforme a geração e o armazenamento das informações vão migrando dos servidores para cada um dos nós da rede, este tipo de sistema de vasculhamento acaba sendo cada vez menos eficiente [7, 40]. Nestes casos, a busca distribuída, que dinamicamente procura informações através dos computadores participantes ao invés de criar um índice, tem grandes vantagens. O conteúdo pode ser imediatamente localizado após a sua publicação, o que não acontece em sistemas como o Google, e também não existe a necessidade de um parque de máquinas ou de uma conexão de grande capacidade com a Internet para manter o sistema de busca funcional.

Outras aplicações Arquiteturas P2P podem ser empregadas em qualquer tipo de sistema que necessite de comunicação entre os nós da rede. Poder-se-ia utilizar um sistema com este tipo de arquitetura para comunicação entre participantes de um jogo *on line* (desta forma eliminando a necessidade de servidores como os que existem atualmente), para a implementação de um sistema de agentes móveis [3], para a implementação de um banco de dados distribuído [24] ou de um sistema de armazenamento distribuído [38].

1.1 Problema

Para entrar em funcionamento, uma aplicação P2P precisa resolver alguns problemas básicos. Os dois problemas mais importantes são a busca, seja por um dado ou por outro nó da rede, e o roteamento de mensagens entre os nós. Muitas soluções foram propostas e cada uma delas, dependendo do caso, tem vantagens e desvantagens. Entretanto, não é muito fácil distinguir, *a priori*, qual a solução mais apropriada para a construção de um sistema P2P. Antes da efetiva implantação do sistema, fica muito difícil obter dados precisos sobre o perfil de utilização e sobre as proporções da rede. A obtenção de uma estimativa do desempenho real que uma certa solução pode alcançar também não é simples. Muitos modelos de simulação foram propostos mas, em qualquer simulação, é sempre necessário fazer diversas suposições, algumas delas

[13] `http://www.google.com/`
[14] `http://www.altavista.com/`

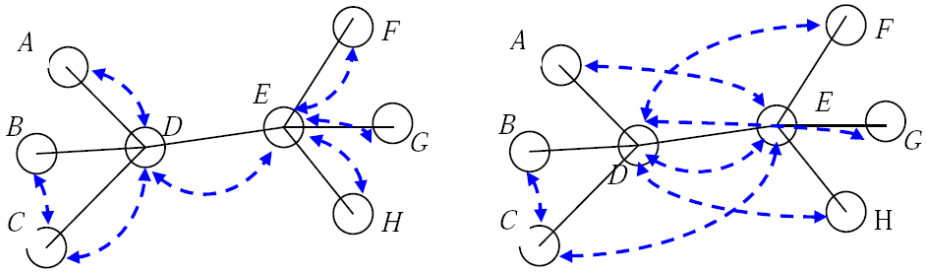

Figura 1: Em linhas contínuas está representada a conectividade física da rede e, em linhas tracejadas, a topologia virtual da rede Gnutella. Na figura à esquerda o mapeamento eficiente: uma mensagem enviada pelo nó A passa através da conexão D-E apenas uma vez para alcançar todos os nós da rede. Na figura à direita um mapeamento ineficiente: o envio da mesma mensagem pelo nó A passa pela conexão D-E por seis vezes.

discutíveis, e aproximações para a simplificação do modelo. Os simuladores disponíveis estão cada vez mais completos e robustos. Entretanto, não há uma maneira segura de avaliar se o comportamento de um sistema feito para executar em milhões de nós vai ser o mesmo no simulador e no ambiente real [28].

A escolha de uma solução incorreta pode comprometer o bom funcionamento da aplicação e, dependendo do grau de acoplamento da aplicação com a solução escolhida, uma mudança em um estágio avançado do projeto pode ser extremamente custosa. Muitos sistemas, em busca de desempenho e simplicidade de implementação, acabam não colocando uma divisão clara entre a aplicação e a camada de comunicação P2P. Várias redes sofrem deste problema. Um exemplo muito conhecido é a rede Gnutella. A solução adotada durante o seu desenvolvimento, apesar de funcionar perfeitamente enquanto a rede era pequena, é comprovadamente não escalável [5]. Estima-se que, em junho de 2001, um dos períodos de auge de utilização desta rede, apenas para manter a rede conectada e difundir as pesquisas feitas pelos usuários, fossem trafegados 330TB por mês. Isso se deve predominantemente ao fato de a topologia da rede Gnutella não mapear a infra-estrutura da rede física e à utilização de técnicas de inundação para a busca de informações [60]. A Figura 1 (extraída de [60]) mostra dois exemplos de possíveis mapeamentos e suas implicações para a infra-estrutura da rede.

1.2 Objetivos

Para evitar problemas como os enfrentados pela rede Gnutella, propomos não escolher a solução *a priori*, mas sim testar diversas soluções *a posteriori* e quiçá alterar a solução adotada com o programa já em produção. Para isto, os diversos sistemas P2P, ao invés de acessarem a camada de comunicação diretamente, fariam-no através de uma fachada [25] que oferece uma API geral que interage com a camada de comunicação através de uma *Service Provider Interface* (SPI)[15]. A existência de diversas implementações dessa SPI, cada uma delas específica para determinada camada de comunicação P2P, permite que a seleção da camada de comunicação seja feita através de um arquivo de configuração. Este tipo de solução não é inédito [20] e exemplos de utilização podem ser vistos nas interfaces de portabilidade do ORB para Java [54], em JNDI[16] e em ODBC[17].

Propostas semelhantes de unificação da interface de programação já foram feitas [18, 22]. Elas definem APIs de baixo nível, entretanto não especificam como os mapeamentos para as linguagens de programação devem ocorrer. O projeto JXTA [27] estabelece uma proposta de unificação de API com um nível mais alto e define como os mapeamentos para as linguagens de programação devem ser feitos, porém define o protocolo de comunicação. O projeto cP2Pc [39] é o que mais se aproxima da presente proposta. Ele define uma API que é suficientemente genérica para poder ser implementada através de diversas redes sem ter que, para isso, definir suas arquiteturas ou protocolos. Com mapeamentos disponíveis para Gnutella e GDN [4], a aplicação P2P que o utiliza tem um bom nível de isolamento da rede de sobreposição utilizada. O projeto cP2Pc, entretanto, tem como foco o mapeamento das diferentes redes de compartilhamento de arquivos, e não o mapeamento de redes de uso geral.

A presente proposta, denominada Hermes, define uma API de uso geral com um nível mais alto que as propostas já existentes, facilitando a implementação de uma aplicação P2P. A existência de uma API deste tipo permite a troca da implementação da infra-estrutura P2P utilizada sem alterações no código da aplicação. Uma aplicação que use uma API de baixo nível, ou que utilize uma rede com os mapeamentos e

[15] Uma implementação da SPI é, em essência, uma estratégia (*strategy*) conforme descrita em [25].
[16] *Java Naming and Directory Interface* (JNDI), `http://java.sun.com/products/jndi/`
[17] *Open Database Connectivity* (ODBC)

protocolos já estabelecidos, não tem esse tipo de liberdade.

1.3 Organização da dissertação

O restante deste documento é organizado da seguinte forma: no Capítulo 2, são introduzidos os conceitos básicos de uma infra-estrutura P2P; no Capítulo 3, uma API que se propõe a resolver os problemas expostos é apresentada; no Capítulo 4, os mapeamentos da API do Hermes para cada uma das implementações das redes de sobreposição são pormenorizados; no Capítulo 5, os trabalhos relacionados são comparados com a nossa proposta e, finalmente, no Capítulo 6, discutimos os trabalhos futuros e apresentamos nossas conclusões.

2 Fundamentos de P2P

As redes P2P são redes de computadores que não necessitam de servidores centrais para funcionar. Todos os participantes da rede são tratados como iguais (*peers*) e cada um destes assume, a cada instante, um papel diferente. Quando estão acessando informações, são clientes; quando estão servindo informações, são servidores; e quando estão repassando informações, são roteadores. O grande desafio é modelar uma rede P2P robusta e escalável, composta de computadores baratos, individualmente não confiáveis e arbitrariamente distribuídos pela rede [37].

Considere os seguintes requisitos para a construção de uma rede:

Disponibilidade da rede A rede deve estar disponível mesmo que ocorra a falha de alguns de seus nós. Isso é essencial para sistemas críticos.

Interoperabilidade entre diversas máquinas Implica que a rede é independente das várias arquiteturas de suas máquinas componentes. Todos os seus nós "falam a língua" da rede, e não o contrário.

Segurança da informação A rede deve oferecer proteção contra a interceptação das transmissões.

Estes três requisitos foram as linhas mestras para a definição do protocolo TCP/IP. Em uma rede TCP/IP todos os nós atuam da mesma forma, com os mesmos direitos e deveres (como *peers*), não existindo nenhum ponto central de controle para monitorar as comunicações. Essa igualdade, no entanto, chega apenas até a camada de transporte. Hoje, grande parte dos protocolos na camada de aplicação da Internet são baseados na arquitetura cliente/servidor (*e.g.*, *Web* e *e-mail*). Nesse sentido, o grande aumento da utilização de aplicações P2P é uma volta aos princípios que nortearam a criação do TCP/IP, onde a igualdade entre os nós participantes da rede é levada ao extremo.

Em 1964, Paul Baran escreveu uma série de trabalhos sobre comunicação distribuída. Em [9] Baran avalia as possíveis estruturas que uma rede de comunicação pode assumir. Ele sugere que as redes podem ser classificadas em duas categorias:

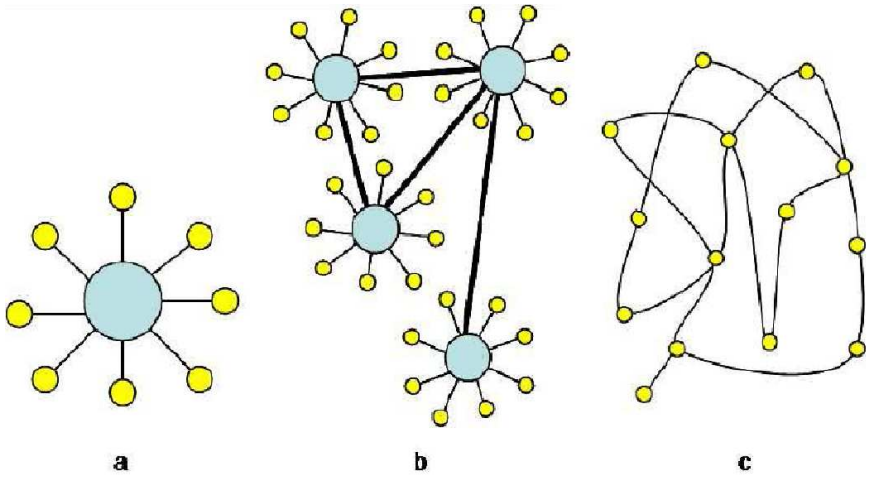

Figura 2: Categoria das redes. (a) uma rede centralizada, (b) uma rede descentralizada e (c) uma rede distribuída.

Centralizadas Uma rede centralizada (Figura 2a) é uma rede na qual os clientes se conectam a um servidor. É uma estrutura parecida com uma estrela, onde o centro é o servidor e as pontas são os clientes. Em uma rede deste tipo, caso o servidor falhe, toda a rede falha. Existe um único ponto central de falha;

Distribuídas Em uma rede distribuída (Figura 2c) não existem servidores pré-definidos. Qualquer nó pode ser um servidor ou um cliente. Não existem pontos centrais de falha.

Na prática, o mais comum é a utilização de uma rede que mistura características de redes centralizadas e distribuídas. Uma rede que utiliza essa estrutura híbrida é denominada descentralizada (Figura 2b). Uma rede descentralizada é um aglomerado de estrelas com os seus centros conectados. Nesse caso existem diversos pontos centrais de falha e a falha de um deles compromete apenas parte da rede [9].

Dizemos que uma rede é P2P quando ela é puramente distribuída. Redes P2P trazem diversas vantagens quando comparadas com redes centralizadas ou descentralizadas. Dois exemplos são a menor suscetibilidade a falhas (não há pontos de falha centrais) e distribuição da carga (o serviço está distribuído pelos nós e não concentrado em alguns poucos servidores). Maiores comparações serão feitas nas seções que seguem.

2.1 Motivação

Algumas das vantagens do uso de redes P2P são apresentadas abaixo:

- As barreiras para a criação de uma rede deste tipo são mínimas já que, ao contrário do que ocorre com sistemas centralizados/descentralizados, esse tipo de rede não necessita de nenhum tipo de administração manual/externa ou de instalações específicas;
- Sistemas P2P oferecem uma maneira de utilizar, de forma conjunta, o enorme potencial de armazenamento e processamento espalhado pelos computadores conectados à Internet;
- A natureza distribuída deste tipo de rede lhe dá o potencial de ser resistente a falhas ou ataques intencionais, o que faz com que tais redes sejam o ambiente ideal para o armazenamento de informações a longo prazo ou computações demoradas.

Uma discussão mais extensa sobre as vantagens de sistemas P2P sobre outros tipos de sistemas pode ser vista em [55].

Redes P2P trazem à tona uma grande quantidade de problemas interessantes para a pesquisa no campo de sistemas distribuídos. Os dois mais importantes são os problemas de busca e roteamento: este compreende o envio de mensagens entre os nós da rede de forma eficiente; aquele pode ser visto como o método utilizado para a localização de um recurso dentro de uma grande rede de forma escalável, sem o auxílio de servidores centrais ou de hierarquia. Diversas soluções foram propostas e na seção seguinte algumas delas serão apresentadas. Freqüentemente as soluções adotadas para a busca e para o roteamento se inter-relacionam de tal forma que o funcionamento de uma depende totalmente do funcionamento da outra.

2.2 Arquiteturas P2P

Abaixo são apresentadas as arquiteturas mais comumente utilizadas na construção de sistemas P2P, juntamente com as suas principais características.

2.2.1 Arquiteturas baseadas em serviços centralizados e descentralizados

Este tipo de arquitetura não cria uma rede totalmente distribuída e, portanto, este tipo de sistema não é um sistema P2P puro. Quando o usuário se conecta à rede, ele, na verdade, está se conectando a um ou mais servidores. Para implementar um serviço de busca, por exemplo, esses servidores funcionam como um índice para busca dos recursos disponíveis. Quando um usuário se conecta, ele envia ao servidor uma lista com todos os seus recursos compartilhados. O servidor atualiza seus índices e assim disponibiliza os recursos deste novo nó para a pesquisa pelos demais usuários. Quando um dos usuários deseja fazer uma busca por um determinado item, ele contacta um dos servidores e efetua a busca pela informação desejada. A resposta do servidor é, geralmente, o endereço IP de uma das máquinas que tem o recurso desejado (essa máquina também já passou por todo o processo de conexão descrito no parágrafo anterior). Desse instante em diante, as duas máquinas passam a se comunicar diretamente uma com a outra para efetuar o uso dos recursos compartilhados. Esta arquitetura geralmente faz uso da camada de roteamento da infra-estrutura de rede para tratar do tráfego das mensagens. A Figura 3 ilustra o processo descrito acima.

O serviço de localização centralizado traz vantagens e desvantagens. Uma das vantagens é a garantia de que, se o recurso estiver disponível em algum dos nós da rede, então ele será encontrado. Esta garantia não está presente em todos os sistemas P2P, como veremos adiante. Além disto, buscas com palavras-chave, coringas (*wildcards*), expressões regulares ou múltiplos critérios são facilmente implementadas. A desvantagem, entretanto, é a alta suscetibilidade a falhas, pois existem pontos centrais de falha. Se esses pontos falharem, a rede toda falha.

Abaixo citamos dois exemplos de sistemas que usam este tipo de arquitetura e discutimos algumas das suas características.

Napster A rede do Napster[18] é muito parecida, senão idêntica, ao esquema apresentado acima. Justamente por isso foi tão fácil fechar as suas portas quando a justiça assim o determinou. Um dos componentes críticos que compunham a sua rede (o serviço de busca) era centralizado. A parte P2P de seu protocolo não funcionava sem a parte centralizada, o que fez da rede um alvo fácil.

[18]Open Nap Home Page: `http://opennap.sourceforge.net/`

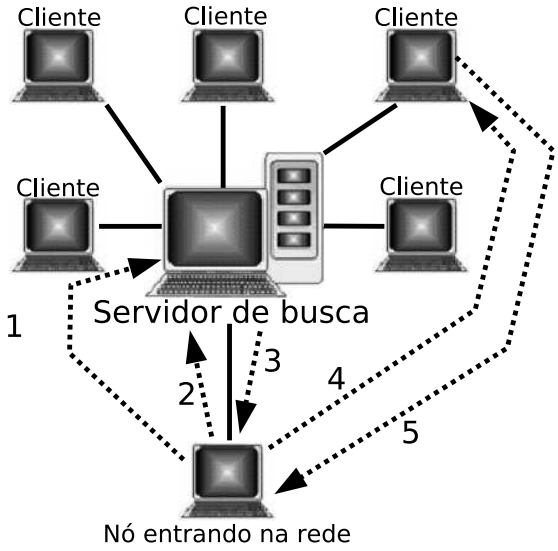

Figura 3: Funcionamento de uma rede centralizada. 1. O novo cliente se conecta ao servidor e envia uma lista dos seus recursos compartilhados. 2. O cliente já registrado no servidor solicita a pesquisa por recursos. 3. O servidor devolve uma lista com o endereço de todos os nós que contêm o recurso procurado. 4. O cliente contacta os nós que contêm o recurso desejado. 5. Os nós contactados transferem o recurso para o nó que o requisitou.

Edonkey A robustez da rede do Edonkey é, em muitos aspectos, superior à rede do Napster. A rede Edonkey é mais resistente a ataques e a comunicação entre os *peers* se dá de uma forma mais inteligente. Essa rede não sofre de muitos dos problemas presentes na rede Napster. Ela foi criada depois do Napster e por isso os seus projetistas tiveram a chance de levar em consideração, durante o seu desenvolvimento, muitas das falhas presentes na rede Napster. Primeiramente o servidor não é centralizado e sim descentralizado. Qualquer usuário pode, tendo uma conexão à Internet, iniciar um servidor na própria máquina. Isto protege, até certo ponto, a rede como um todo: a queda de alguns servidores não compromete a rede toda. Para acabar com a rede ter-se-ia que acabar com todos os servidores - uma tarefa bem difícil, já que eles estão sob controle dos usuários. O esquema de busca é bem parecido com o esquema já apresentado. O cliente se conecta a um dos servidores e lá publica a lista de seus arquivos compartilhados. Quando um nó deseja efetuar uma pesquisa, ele itera sobre a lista de servidores enviando requisições de pesquisa. Cada servidor responde a essa requisição com uma lista dos IPs dos nós de sua responsabilidade que contêm o item pesquisado. O cliente recebe as listas dos servidores e as percorre contactando os nós devolvidos na pesquisa. Os nós contactados então respondem enviando o dado requisitado. Toda vez que um nó fornece um dado a outro, este é adicionado em um cache local daquele. Esse cache guarda os prováveis possuidores daquele dado. Desta forma, quando um nó recebe a requisição por um dado, ele, além do dado propriamente dito, envia uma cópia do seu cache com a lista dos outros nós conhecidos que provavelmente possuem esse dado. Isso faz com que sejam encontrados praticamente todos os nós da rede que têm o dado desejado, mesmo no caso em que a lista de servidores do nó que está efetuando a pesquisa não esteja atualizada. A única restrição é que todos os nós tenham na sua lista de servidores um único servidor em comum. Essa restrição é facilmente cumprida, já que a lista de servidores ativos é publicada regularmente na própria página do Edonkey.

2.2.2 Arquiteturas baseadas em redes de sobreposição não-estruturadas

Arquiteturas de rede baseadas em inundação (*Flooding Style Networks*), ou redes de sobreposição não-estruturadas, são redes 100% P2P. Estas redes são totalmente distribuídas e portanto não possuem um ponto central de falha. A única maneira de desmantelar a rede por completo seria acabar com todos os seus nós, uma tarefa que é, para não dizer impossível, extremamente difícil. O modo pelo qual os nós se ligam à rede não é estruturado. Um nó que deseja entrar na rede simplesmente descobre alguns outros nós já presentes na rede (por difusão ou qualquer outro meio) e se conecta arbitrariamente de modo não estruturado a alguns deles, daí o nome desse tipo de rede. As pesquisas nestas redes são feitas por mecanismos de inundação controlada por TTL (*time-to-live*). Um nó que deseje pesquisar por recursos manda uma mensagem para todos os nós aos quais ele esteja conectado (seus vizinhos). Cada um de seus vizinhos avalia a mensagem e verifica se tem o recurso pesquisado. Caso possua, ele envia uma mensagem de volta ao nó de onde se originou a pesquisa informando o seu endereço. Caso não possua, ele incrementa o número de saltos da mensagem (*hop count*) e a repassa para todos os seus vizinhos. Quando o número de saltos ultrapassar o TTL o repasse de mensagens é interrompido. Logo, o TTL define o horizonte ou o raio de ação da pesquisa. A distância entre dois nós é medida em função do número de saltos que uma mensagem precisa dar para ir de um nó ao outro. Sistemas deste tipo têm mecanismos específicos para lidar com mensagens presas em laços ou mensagens enviadas por usuários mal intencionados apenas para congestionar a rede. Entretanto, sistemas baseados nessas redes não são escaláveis pois necessitam de um grande poder de processamento e largura de banda. Além disso, elas não oferecem garantia alguma quanto à alcançabilidade dos recursos[19] ou ao tempo necessário para encontrá-los [23]. A Figura 4, adaptada de [23], ilustra este esquema mostrando uma busca efetuada pelo nó Nq com TTL = 2.

Gnutella O protocolo do Gnutella segue o esquema descrito acima com algumas poucas alterações e, portanto, possui os mesmos pontos fracos. De fato, existem relatos de que no dia seguinte ao fechamento do Napster, a rede Gnutella parou de funcionar devido a carga criada pela grande quantidade de

[19] Diz-se que um recurso é inalcançável quando a distância entre o nó de origem da pesquisa e o nó que contém o recurso é maior que o TTL definido para a pesquisa. Um exemplo é a requisição feita por Nq por um recurso que está em Nar na Figura 4. Numa rede não-estruturada, a busca de um recurso pode resultar em falsos negativos caso o recurso seja inalcançável.

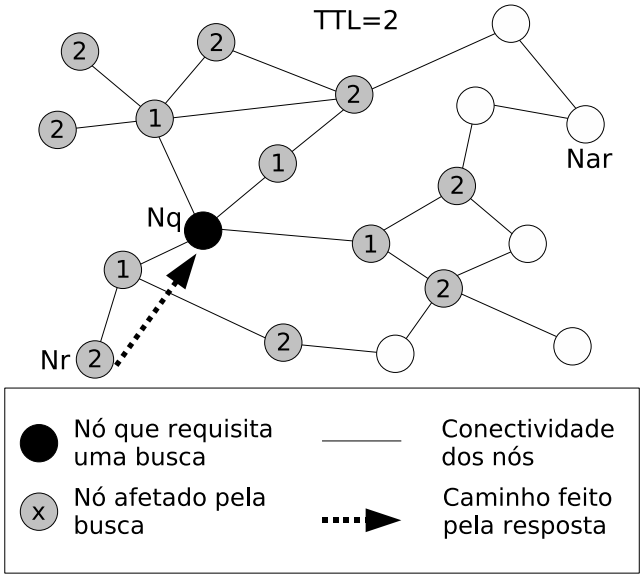

Figura 4: Uma representação do funcionamento de uma rede P2P baseada em inundação controlada por TTL. O nó Nq transmite uma consulta requisitando o valor de uma chave que está localizada em Nr. Dentro de cada um dos nós alcançados pela busca estão os números que representam o número de saltos da mensagem.

usuários migrando do Napster para o Gnutella[20] [5]. O anonimato dos usuários é praticamente inexistente. Qualquer um na rede consegue facilmente listar as pesquisas que estão sendo feitas e identificar os seus pesquisadores. Outro ponto fraco é a proteção de quem está fornecendo os arquivos: os seus endereços IP também podem ser facilmente listados, já que o protocolo utilizado é o HTTP.

A rede Gnutella, no entanto, não tem apenas pontos fracos. Um dos pontos fortes é a liberdade de pesquisa. Cada um dos nós da rede mantém uma lista dos seus arquivos compartilhados. Uma mensagem de pesquisa, dependendo da implementação utilizada, pode ser composta de uma expressão lógica ou de caracteres coringas. Neste sentido as pesquisas por arquivos no Gnutella não deixam nada a desejar às arquiteturas de busca centralizada (obviamente levando em conta o horizonte das mensagens).

Freenet Como um programa de código fonte aberto, o Freenet [19] foi originalmente desenvolvido por

[20]Este tipo de problema poderia ser resolvido através do uso de *superpeers*, como é feito pela plataforma P2P Fasttrack (`http://www.fasttrack.nu`). Entretanto, o uso de *superpeers* traz consigo a perda de resistência a falhas nos *superpeers* nos pontos mais altos da hierarquia, além de não resolver o problema da alcançabilidade dos dados.

Ian Clarke. O Freenet é um sistema bem mais radical que o Napster ou o Gnutella. Assim como no Gnutella, não há um servidor central. Não existe a possibilidade de rastrear a origem de um arquivo, quem o está obtendo ou quem salvou este arquivo em disco. O objetivo é simples: anonimato. Ele promete criar uma Internet sem censura e anônima dentro da Internet. Arquivos, uma vez disponíveis, não podem ser (facilmente) retirados da rede. As metas do Freenet são bem claras: liberdade social e política. A utilização dessas redes é discutida em [11, 19]. Existem relatos da utilização deste sistema na China e no Oriente Médio para espalhar informações censuradas pelo governo. Os arquivos colocados na rede do Freenet não são arquivados nas máquinas das pessoas que os dispõem, mas sim em outras máquinas aleatórias na rede. Quando a requisição por um certo arquivo cresce, ele é automaticamente copiado para os outros nós na vizinhança da sua localização. Esse mecanismo do Freenet distribui por vários nós as informações mais populares na rede. Em contrapartida, como a quantidade de armazenamento é limitada, arquivos muito pouco populares vão pouco a pouco dando lugar aos arquivos mais populares.

Dessa forma os arquivos menos populares tendem a desaparecer completamente da rede após um certo tempo. Todos os arquivos, quando dispostos, são criptografados e assinados digitalmente de modo a garantir que ninguém será capaz de adulterar o seu conteúdo. Por esse mesmo motivo, o dono de um nó não sabe que tipo de dado está armazenado em sua máquina (isso evita que o dono de uma máquina participante seja responsabilizado por possuir certos dados). Para garantir o anonimato, os projetistas do Freenet tiveram de fazer algumas escolhas que, a princípio, podem parecer estranhas.

Um exemplo é a pesquisa por dados. O Freenet usa uma arquitetura baseada em inundação controlada por TTL, assim como o Gnutella. Mas diferentemente do que ocorre no caso do Gnutella, a busca, quando chega a algum nó da rede que possui o dado desejado, resulta no envio do dado pelo caminho por onde a busca veio. Isso evita que o pesquisador seja identificado, já que, em cada nó por onde a pesquisa passa, a origem desta é criptografada antes de ser enviada para o próximo vizinho. As buscas no Freenet são feitas em profundidade e não em largura, o que evita uma grande inundação de mensagens na rede. A busca dos recursos em profundidade no grafo de conexões, em oposição a

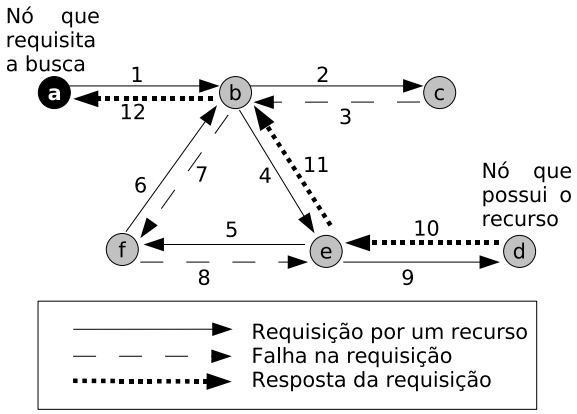

Figura 5: Seqüência típica do funcionamento de uma requisição. O nó a efetua uma busca por um recurso que está armazenado no nó d. A requisição sai de a e varre o grafo em profundidade (note que existe um mecanismo para evitar mensagens presas em laços no passo 6-7) até alcançar d. A resposta é então enviada a a através do mesmo caminho pela qual a requisição veio.

uma busca em largura, poderia fazer com que a busca fosse demorada já que ela não é feita através de todos os vizinhos concorrentemente. A solução encontrada pelos projetistas do Freenet foi manter em cada nó, juntamente com a lista de vizinhos, todos os identificadores dos arquivos que o nó acredita[21] que cada de seus vizinhos possui.

Assim, a primeira busca por um arquivo pode ser demorada, mas as demais devem ser mais rápidas, pois um caminho, que provavelmente é "certo", já está demarcado. Este comportamento está representado na Figura 5 (adaptada de [19]).

O Freenet possibilitou [11]:

- A garantia do anonimato de seus participantes;
- Que nós, ainda que com uma pequena banda, distribuíssem os documentos grandes e populares;
- Que os arquivos fossem distribuídos automaticamente para mais nós e para a parte da rede P2P na qual eles são mais requisitados. Na verdade, o criador de um arquivo pode se preocupar

[21] O nó que guarda a lista não sabe ao certo se o nó vizinho que respondeu era o possuidor do recurso procurado, ou algum outro nó que acabou sendo alcançado pela busca em profundidade através de seu vizinho. Para todos os efeitos, é como se o seu vizinho possuísse o recurso, já que é através dele que será feito o acesso.

menos com espaço para armazenamento e largura de banda. A rede traz a informação para perto de quem a deseja;

- Que arquivos populares se espalham pela rede, enquanto arquivos indesejados acabam desaparecendo naturalmente. Isso é vantajoso, pois previne que arquivos considerados valiosos pela maioria dos usuários sejam retirados da rede.

A fraqueza do Freenet está no seu mecanismo de busca. Assim como o Gnutella, ele não traz garantias quanto à alcançabilidade de um arquivo. Ele também não é apropriado para buscas por nomes de arquivos, pois os nós da rede possuem apenas o identificador do arquivo além do seu conteúdo criptografado. Todas as buscas no Freenet são efetuadas utilizando esse identificador, que é obtido através do cálculo da função SHA-1 [1] sobre o nome do arquivo. Ter apenas o identificador e o arquivo criptografado garante que um utilizador do Freenet não saiba quais dados estão armazenados na sua máquina e, portanto, não seja responsabilizado por possuí-los. A busca dos arquivos por nomes deve ser feita de alguma outra forma, fora da rede Freenet. Isso já ocorre em outras redes como, por exemplo, a rede P2P BitTorrent[22], para a qual existem vários catálogos de arquivos disponíveis[23].

2.2.3 Arquiteturas baseadas em redes de sobreposição estruturadas

Um problema muito comum em aplicações P2P consiste em localizar eficientemente um nó que possua um recurso de interesse. As seções 2.2.1 e 2.2.2 descreveram algumas maneiras de resolver esse problema. A principal desvantagem do método com arquitetura centralizada é a sua grande suscetibilidade a ataques. Já nas redes baseadas em inundação existe a falta de escalabilidade, além da falta de garantias quanto à alcançabilidade dos recursos na rede. As redes de superposição estruturadas (*structured overlay networks*) foram criadas para tentar preencher as lacunas deixadas pelas arquiteturas já apresentadas. Redes como CAN [56], Chord [65], Pastry [63], Kademlia [49] e Viceroy [47] criam uma topologia virtual sobre a topologia física da rede. Numa rede não-estruturada, como a rede Gnutella, a conexão entre os nós é feita

[22] `http://bitconjurer.org/BitTorrent/`

[23] Exemplos de catálogos deste tipo são o Mininova (`http://www.mininova.org/`) e o Torrentreactor (`http://www.torrentreactor.net/`).

de maneira arbitrária. As redes de sobreposição estruturadas diferentemente daquele tipo de rede criam as conexões entre os nós seguindo uma estrutura bem definida, daí o seu nome. O Chord, por exemplo, cria as conexões entre os nós numa topologia de anel. As redes estruturadas são redes que possuem garantias de alcançabilidade dos recursos e nas quais é possível provar os limites superiores para o tempo de busca (geralmente O(log n), onde n é o número de nós da rede). As redes estruturadas oferecem também balanceamento automático de carga e auto-organização.

Essas características merecem uma melhor explicação. A conexão entre os nós da rede é feita tendo como base o identificador do nó. Esse identificador é, geralmente, obtido com alguma função de *hash* segura, como o SHA-1, calculado sobre alguma informação única do nó (*e.g.* endereço IP). As conexões entre os nós são feitas conforme um protocolo pré-estabelecido, diferentemente de redes não-estruturadas. Isso possibilita que a pesquisa por recursos seja modelada através de um processo iterativo determinístico, que garante a alcançabilidade dos dados, ao invés de uma busca baseada em inundação (por profundidade ou largura) no grafo de conexões. A análise deste processo permite, portanto, que seja estabelecido um limite superior para o número de mensagens que precisam ser trocadas em um processo de busca. Em termos abstratos, podemos olhar uma rede de superposição como uma tabela de *hash* distribuída (DHT - *Distributed Hash Table*) que oferece as operações de inserção e consulta. As chaves da tabela são, tipicamente, obtidas através de alguma função de *hash* segura, geralmente a mesma função utilizada na geração do identificador do nó.

Todas as redes estruturadas citadas definem algoritmos de estabilização para tratar a entrada e a saída (seja por falha ou não) de nós da rede. A responsabilidade destes algoritmos é manter a topologia da rede consistente. Essas redes são muito úteis para a criação de aplicações que requerem buscas rápidas, escaláveis e confiáveis por pares únicos [24] de chave-valor.

Chord O protocolo Chord define apenas uma operação: dada uma chave, mapeá-la para um dado nó. O Chord, na verdade, é um arcabouço que aplicações P2P podem utilizar para efetuar as buscas e estruturar a rede.

O esquema de funcionamento deste protocolo é, de uma forma simplificada, explicado a seguir. Mai-

[24] Apesar de ser possível, entretanto improvável, que as funções de *hash* utilizadas forneçam o mesmo resultado para duas entradas diferentes, alguns arcabouços ignoram este fato deixando por conta do seu utilizador o tratamento de colisões na DHT.

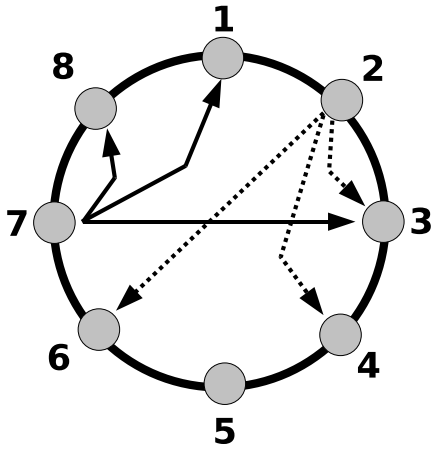

Figura 6: Vizinhança dos nós. As setas com a linha pontilhada apontam para os vizinhos do nó 2. As com a linha contínua apontam para os vizinhos do nó 7. Neste caso, m = 3, portanto o número máximo de nós é $2^3 = 8$. Logo cada nó tem apenas 3 vizinhos na sua tabela de apontadores.

ores detalhes sobre o protocolo utilizado pelo Chord podem ser vistos em [65].

No Chord, cada um dos nós possui um identificador (ID). O identificador de um nó é obtido através da aplicação de uma função de *hash* no seu endereço IP. A rede é estruturada em forma de anel. Os nós são colocados neste anel em ordem crescente de ID[25]. Cada nó i guarda uma tabela de apontadores para outros O($\log n$) nós, onde n é o número de nós da rede. Cada entrada dessa tabela tem, como chave, o ID de um nó e, como valor, o IP desse nó. Os nós apontados por essa tabela são denominados vizinhos de i. Os nós vizinhos do nó i são os nós com os identificadores $(i + 2^{k-1}) \bmod 2^m$, com $1 \leq k \leq m$, onde m é o número máximo de bits que o ID pode assumir (logo, 2^m define o número máximo de nós que a rede comporta). A Figura 6 mostra um exemplo de vizinhança.

Para efetuar uma busca pelo nó X com o identificador ID(X), primeiramente, a tabela de apontadores é investigada para encontrar o nó Y tal que ID(Y) é máximo e que ID(Y) $<=$ ID(X). Se ID(X) $=$ ID(Y), o nó foi encontrado. Caso contrário, uma requisição é enviada para o nó Y. O nó Y repete os mesmos passos feitos acima até que o nó desejado seja encontrado ou seja determinada a sua ausência

[25]Por motivos de simplicidade vamos convencionar que a varredura do anel sempre se dá no sentido horário. Convencionamos também que, quando escrevermos "nó i" estamos na verdade nos referindo ao nó com o identificador i.

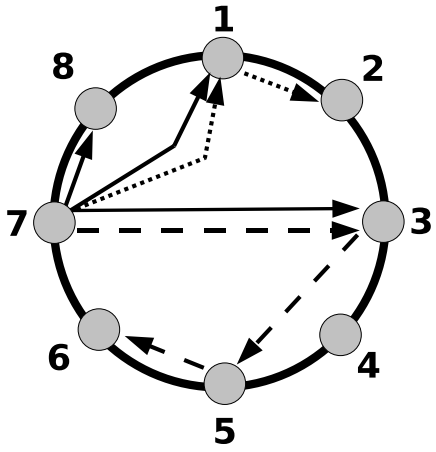

Figura 7: Buscas com sucesso por um nó. As setas com linhas contínuas apontam para os vizinhos do nó 7. As setas com linha pontilhada indicam o caminho percorrido pelo processo para achar o nó com identificador 2 e as setas com linhas tracejadas mostram o caminho para se encontrar o nó com identificador 6.

na rede. É possível provar que em no máximo O($\log n$) passos o nó procurado vai ser encontrado ou será determinada a sua inexistência na rede. As figuras 7 e 8 ilustram este processo.

A localização dos recursos no Chord é bem similar à localização de nós, e por isso pode ser facilmente confundida com esse conceito. Para cada recurso disposto na rede é calculado um ID. O cálculo deste ID é feito utilizando-se uma função de *hash* sobre algum descritor desse recurso. Esta função é a mesma utilizada para calcular cada um dos IDs dos nós. Portanto, o espaço de IDs dos nós é o mesmo espaço dos IDs dos recursos. Os recursos disponibilizados na rede são guardados em uma DHT. Esta tabela é implementada da seguinte maneira: cada nó é responsável por guardar uma lista dos recursos cujos IDs são maiores ou iguais ao seu próprio ID e que são menores que o ID do primeiro nó da sua lista de apontadores (ou seja, ele guarda todas as localizações dos recursos cujos IDs estão entre o seu próprio ID e o ID de seu primeiro vizinho). A Figura 8 pode ser utilizada para ilustrar o processo de busca por um recurso com ID $= 4$ em uma rede com 7 nós com o espaço de IDs de 3 bits ($m = 3$). O nó localizado na busca (3) é o nó da rede cujo ID satisfaz as condições apresentadas logo acima. Portanto é o nó que potencialmente tem armazenado o recurso procurado.

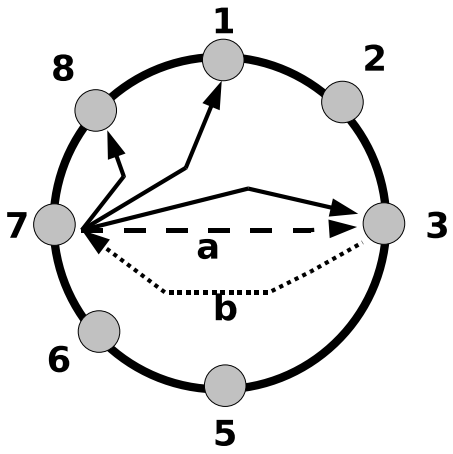

Figura 8: Buscas sem sucesso. As setas com linhas contínuas apontam para os vizinhos do nó 7. Nó 7 tenta localizar o nó 4. O nó 4 não está na lista de apontadores do nó 7 e a pesquisa é repassada para o nó 3 (a). O menor apontador do nó 3 aponta para o nó 5 que é maior que o nó procurado. É, então, devolvido como resposta (b) o maior nó que é menor que o nó procurado, no caso o nó 3.

O Chord garante que os resultados da pesquisa são corretos ainda que apenas uma entrada da sua tabela de apontadores esteja correta. O desempenho do sistema degrada conforme o número de entradas incorretas na tabela de apontadores cresce. Como a tabela de apontadores é bem pequena (tipicamente O ($\log n$) entradas), é muito fácil mantê-la atualizada através de algoritmos de estabilização. É o que o Chord faz. Testes mostraram que o Chord é capaz de manter o tempo de pesquisa em O ($\log n$) mesmo com 50% de probabilidade de falhas por nó [23,65]. O Chord, além de executar os algoritmos de estabilização a intervalos regulares de tempo, também funciona com múltiplos nós virtuais por nó real. Isto ajuda o balanceamento de carga dos nós, pois propicia uma distribuição mais uniforme das entradas da DHT pelos nós da rede [23].

2.2.4 Comparação entre as arquiteturas

A seguir apresentamos uma breve comparação entre as arquiteturas apresentadas. A comparação foi baseada nos seguintes critérios: tolerância a falhas, escalabilidade, busca e segurança.

2.2.4.1 Tolerância a falhas Apesar de termos apresentado a arquitetura com serviço de localização centralizado, tipicamente este tipo de rede não pode ser considerado como uma rede P2P verdadeira. Por não ser distribuída, esse tipo de rede possui uma tolerância a falhas muito baixa, pois há um ponto central de falha.

Por outro lado, as redes distribuídas, tanto as estruturadas quanto as não-estruturadas, têm uma alta tolerância a falhas. Ainda que ambas apresentem uma boa resistência a ataques, elas têm alguns pontos que poderiam ser explorados por um eventual adversário.

As redes não-estruturadas, apesar de não serem redes livres de escala[26] puras, se apresentam como uma rede que possui todas as vantagens e desvantagens de uma rede deste tipo [60]. Por possuírem essas características, a falha aleatória de nós da rede não lhes traz grandes impactos [2, 7], ainda que a taxa de falhas seja extremamente alta. Entretanto, a remoção de um número pequeno (em relação ao tamanho da rede) de *hubs* pode levar ao particionamento da rede.

Por construção, as redes estruturadas não são afetadas por fenômenos como a conexão preferencial de novos nós [6], ademais o número de conexões por nó é praticamente constante, tipicamente O($\log n$) onde n é o número de nós da rede. Tais características são próprias de redes aleatórias[27]. Pela inexistência de *hubs* nesse tipo de rede, o efeito criado pela retirada de nós escolhidos ao acaso é idêntico ao da retirada de nós escolhidos por ordem decrescente de conectividade. Mostra-se, entretanto, que redes aleatórias não oferecem o mesmo grau de robustez que as redes livres de escala [2] para falhas aleatórias de uma grande quantidade de nós.

Para tratar problemas de particionamento e manutenção da topologia da rede, existe um processo constante de manutenção das conexões. A manutenção da rede é uma operação dispendiosa. Vários traba-

[26] Uma rede é livre de escala (*scale-free network*) quando os números de conexões de alguns nós da rede são muito maiores do que o número de conexões médio dos demais nós da rede. Nós que possuem muito mais conexões que os demais nós são denominados *hubs*. Mais formalmente, nessas redes a probabilidade $P(k)$ de que um nó possua k conexões é dada por $P(k) = k^{-\gamma}$, onde γ é um número dependente da rede que normalmente vale entre 2 e 3 [8].

[27] Em uma rede aleatória (*random network*), o número de conexões de cada um dos nós é aproximadamente igual ao número médio de conexões dos demais nós da rede. Mais formalmente, a probabilidade $P(k)$ de que um nó possua k conexões é dada por uma distribuição Poisson. Portanto, após um certo valor de k a probabilidade de encontrar um nó com k conexões decresce exponencialmente. Por isso, redes aleatórias também são chamadas de redes exponenciais (*exponential networks*). Numa rede desse tipo, a existêcia de *hubs* é muito improvável [7].

lhos [15, 29, 60, 64] foram feitos a esse respeito, e aqui não nos aprofundaremos muito neste assunto. De fato, enquanto redes com serviço de localização centralizado são extremamente simples de se manter, as redes totalmente distribuídas não são. A constante entrada e saída dos nós da rede (*churn*[28]) é facilmente resolvida por um servidor central. Entretanto, no caso totalmente distribuído, como não existe um nó central, todos os nós precisam enviar mensagens a intervalos regulares de tempo para manter a consistência da rede, e no caso das redes estruturadas manter a DHT. Isto pode ser bastante custoso. Estima-se que, em junho de 2001, a rede Gnutella fosse composta de aproximadamente 50.000 nós e 170.000 conexões. Com essas dimensões, a manutenção da rede exigia que fossem trafegados, aproximadamente, 330TB/mês, excluindo-se as transferências de arquivos. Para se ter uma base de comparação, esse volume de tráfego representava cerca de 1.7% de todo o tráfego de rede nos *backbones* dos EUA em dezembro de 2000 [60]. Portanto, existe um compromisso claro que precisa ser adotado, no momento da escolha da arquitetura, entre tolerância a falhas e tráfego para manutenção da rede. Arquiteturas de busca descentralizada, como o E-donkey, assim como esquemas híbridos [15, 44] (com a presença de características de redes estruturadas e redes não-estruturadas), foram propostos e os resultados em alguns casos superam o desempenho dos esquemas puros.

2.2.4.2 Escalabilidade Redes com serviço de localização centralizado tendem a não ser tão escaláveis quanto redes descentralizadas ou distribuídas. Quase toda a carga de manutenção e busca de informações na rede fica a cargo de um ou alguns poucos nós. Numa rede descentralizada, este problema é contornado com a adição de novos servidores, mas ainda assim continuam a existir nós diferenciados na rede e geralmente estes nós especiais tem uma carga maior tanto de tráfego de rede quanto de processamento. Nas redes distribuídas não-estruturadas, todos os nós têm uma carga semelhante, entretanto no caso degenerado todos os nós necessitam processar todas as consultas enviadas por todos os outros nós, o que pode ser muito custoso. As redes estruturadas, por sua vez, não têm este tipo de problema: cada um dos nós da rede é responsável por uma fração pré-definida dos dados dispostos na rede e, portanto, acaba ocorrendo automaticamente uma

[28] *Churn* é uma palavra inglesa que significa agitar, bater ou gerar uma movimentação excessiva em algo. Neste contexto, representa a "movimentação" criada na rede pela entrada e saída dos nós. Denomina-se *churn rate* a razão entre o número de nós entrando e o número de nós saindo da rede em um determinado período de tempo.

distribuição de carga entre os nós. Um possível problema é um nó relativamente inferior em relação aos outros em termos de capacidade de processamento e tráfego de rede ficar responsável pela manutenção do índice de um recurso que é muito requisitado na rede. Este problema é tratado pela maior parte das redes estruturadas através de replicações. Cada um dos nós que tem a réplica do recurso acaba por responder por grande parte das requisições no lugar do nó "oficialmente" designado para o tratamento daquele recurso. Tanto as redes centralizadas quanto as distribuídas não-estruturadas oferecem resultados satisfatórios para tamanhos pequenos e médios da rede, mas acabam tendo o seu desempenho comprometido nas redes grandes. Nestes casos as redes descentralizadas e distribuídas estruturadas se destacam em relação às demais.

2.2.4.3 Busca A pesquisa por recursos é parte fundamental de sistemas P2P. É de senso comum que pesquisas em redes com servidores centralizados, redes descentralizadas ou redes distribuídas não-estruturadas são mais poderosas do que pesquisas em redes estruturadas, já que naquelas se pode efetuar pesquisas complexas de forma muito mais simples e eficientes do que nas redes distribuídas estruturadas. De fato, muitas propostas, como [12, 30, 57], foram feitas, mas a verdade é que as redes distribuídas estruturadas estão distantes da versatilidade oferecida pelas demais redes no que se refere à pesquisa de dados.

2.2.4.4 Segurança Quando se passa o controle total das responsabilidades de pesquisa e roteamento para os nós da rede, alguns problemas, como a privacidade do usuário, acabam aparecendo. Nós maliciosos podem estar conectados à rede, disseminando informações falsas ou monitorando o tráfego. Até onde foi possível averiguar, não existe nenhum sistema centralizado ou descentralizado que ofereça ao menos algum grau de anonimato na rede (a não ser pela utilização de *proxies*). Existem diversas implementações no caso das redes distribuídas, sendo que as mais conhecidas são o Freenet e o Neblo [17], no caso de redes não-estruturadas e estruturadas, respectivamente.

2.2.4.5 Conclusão A tabela 1 mostra de forma resumida os itens discutidos acima:

Não se pode concluir que exista um tipo de rede melhor do que outra. Cada uma delas, dependendo do tipo da aplicação, tem vantagens e desvantagens. Deve-se avaliar o tipo, complexidade, tamanho e

	Centralizada/Descentralizada	**Não-estruturada**	**Estruturada**
Pontos centrais de falha	Sim	Não	Não
Resistência a ataques	Ruim/Razoável	Boa	Boa
Escalabilidade	Ruim/Razoável	Razoável	Boa
Manutenção da rede			
Utilização de banda	Muito Boa	Ruim	Boa
Administração	Ruim	Boa	Boa
Busca			
Robustez	Boa	Boa	Razoável
Alcançabilidade	Muito Boa	Ruim	Boa
Tempo	Bom	Ruim	Bom

Tabela 1: Tabela comparativa entre as diferentes arquiteturas.

perfil de utilização do sistema que se deseja criar e ponderar estas avaliações em conjunto com as características de cada rede. Certos dados como, por exemplo, o perfil de utilização do sistema, podem não ser de fácil obtenção antes da efetiva implantação do sistema e isto pode dificultar bastante a decisão. Este capítulo mostrou apenas algumas das características presentes nos diferentes tipos de rede e implementações. Comparações mais aprofundadas sobre as características de cada arquitetura, suas vantagens, desvantagens e implicações sobre o desenho do sistema podem ser vistas em [45, 61], além das referências já feitas neste capítulo.

2.3 Conceitos e estruturas de dados

2.3.1 Filtros de Bloom

O filtro de Bloom [13] é uma estrutura de dados para a representação de um conjunto de elementos de uma forma condensada. Essa estrutura permite que sejam feitas, de uma forma eficiente, consultas sobre a pertinência de um elemento. Diferentemente de uma tabela de *hash* comum, essas consultas são feitas de uma forma probabilística, assim falsos positivos são possíveis enquanto falsos negativos não são. A probabilidade de falsos positivos aumenta conforme o número de elementos do conjunto aumenta. A vantagem de utilizar uma estrutura de dados desse tipo é o baixo consumo de memória.

Um filtro de Bloom é composto essencialmente por um vetor de m bits e por k funções de *hash* distintas. Uma função $h : A \longrightarrow B$ desse tipo deve ser tal que A seja um superconjunto dos elementos a serem adicionados ao filtro, e B um conjunto tal que $B \subset \{0, 1, 2, \ldots, m-1\}$. A princípio, todos os bits do vetor valem 0. Para cada elemento adicionado ao filtro, as k funções de *hash* são calculadas e os bits do vetor, cujo índice é igual ao valor calculado dessas funções, têm o seu valor alterado para 1. Para verificar se um elemento está no conjunto basta calcular o valor das k funções de *hash* e avaliar se os bits do vetor nessas posições estão com o valor igual a 1.

A grande vantagem de se utilizar um filtro de Bloom é ter o poder de se efetuar um balanceamento entre o tamanho do filtro m, e portanto da memória utilizada, o tempo de processamento (dependente do valor de k) e a probabilidade de falsos positivos F.

Para estimar o valor de F vamos assumir que as k funções de *hash* escolhidas são boas, ou seja, distribuem uniformemente os elementos no contradomínio, e que são independentes entre si. Note que após a inserção de n elementos no filtro, a probabilidade P de que um bit em particular ainda esteja com o valor 0 é:

$$P = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-\frac{kn}{m}} = \overline{P}$$

Seguindo nesse raciocínio, a probabilidade de um bit estar com o valor 1 é o valor complementar a P, ou seja, $1 - P$. Para o algoritmo acima fornecer um falso positivo para o teste de pertinência de um elemento, todos os valores dos bits do vetor nos índices calculados a partir das k funções de *hash* devem estar valendo

1. A probabilidade disso ocorrer é dada por:

$$F = (1-P)^k \approx \left(1-\overline{P}\right)^k$$

$$F = \left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^k \approx \left(1-e^{-\frac{kn}{m}}\right)^k = \overline{F}$$

É possível mostrar que o ponto de mínimo global de $\overline{F}$ ocorre quando $k = \frac{m}{n}\ln 2$. Assim, a probabilidade de ocorrer um falso positivo em um filtro que utiliza um número de chaves ótimo é aproximadamente igual a

$$\begin{aligned}
&\left(1-e^{-\frac{k}{nm}}\right)^k \\
&= \left(1-e^{-\ln 2}\right)^{\frac{m}{n}\ln 2} \\
&= (1/2)^{\frac{m}{n}\ln 2} \\
&\approx 0.6185^{\frac{m}{n}}
\end{aligned}$$

Os sistemas que se utilizarem de filtros de Bloom devem fazer uso dessa probabilidade para calcular um valor apropriado para o tamanho do filtro (o número de bits do vetor), levando em consideração o número de elementos que desejam adicionar.

2.3.2 n-gramas

Sejam $X = (x_1, x_2, \ldots, x_t)$ e $Y = (y_1, y_2, \ldots, y_s)$ seqüências (ou cadeias) com t e s elementos respectivamente. Y é uma subseqüência de X se $s \leq t$ e existe um índice $i, 0 \leq i \leq t-s$ tal que $\forall j, 1 \leq j \leq s, x_{i+j} = y_j$.

Denominamos *n-grama* de X qualquer subseqüência de X com tamanho n.

O número total de n-gramas de uma seqüência X com t elementos é dado por $t-n+1$.

2.3.3 Um algoritmo distribuído para o casamento aproximado de cadeias de caracteres

Sejam X e Y duas cadeias de caracteres. Dizemos que Y está contido em X se Y é uma subseqüência de X. Existem diversos algoritmos para determinar se uma cadeia está contida em outra, como por exemplo os

algoritmos de Rabin-Karp [34], Knuth-Morris-Pratt (KMP) [35] e Boyer-Moore [14]. Apesar de serem eficientes, esses algoritmos não funcionam muito bem de uma maneira distribuída. Em um sistema distribuído, quanto menos dados forem trafegados entre dois nós, mais rapidamente uma requisição poderá ser atendida. Abaixo propomos um algoritmo para a resolução desse problema.

Nos sistemas P2P atuais, as buscas por recursos se baseiam principalmente em duas estratégias: busca por inundação ou uso de uma DHT.

Em um sistema cujas buscas são baseadas em técnicas de inundação, o nó que iniciou a busca repassa aos seus vizinhos os critérios para a sua busca. Esses nós verificam se possuem os recursos desejados avaliando os critérios recebidos. Se possuírem algum recurso que encaixe nos critérios recebidos, respondem a requisição ao nó que iniciou a busca. Após isso, repassam a requisição de busca para os seus vizinhos. O processo é repetido até que um limite de repasses seja alcançado. Efetuar uma busca por casamento de cadeias de caracteres em um sistema P2P desse tipo é muito simples: basta que, junto com os critérios enviados de nó a nó, seja enviada a cadeia que se quer casar e aplicar um algoritmo como o KMP. De fato, é dessa maneira que sistemas como o Gnutella e o KaZaA funcionam. Outros sistemas baseados em inundação como o Freenet, em busca de anonimato, fazem as buscas através de chaves opacas. Isso faz com que nesse sistema seja impossível, sem comprometer o anonimato, fazer a busca da maneira exposta acima.

Os sistemas nos quais as buscas são feitas com o uso de uma DHT são, assim como o Freenet, baseados em chaves opacas. Essas chaves são geralmente calculadas utilizando alguma função de *hash* segura como o SHA-1. As buscas por palavras-chave nesse tipo de rede são feitas através de um índice invertido que mapeia as chaves calculadas com a função de *hash* para um valor. Em sistemas desse tipo, não é possível fazer diretamente uma busca por trechos de texto da mesma maneira como acontece nas redes não-estruturadas como o Gnutella.

A seguir propomos um algoritmo para a publicação e busca através de casamentos aproximados de cadeias de caracteres. Esse algoritmo é apropriado para sistemas que se baseiam em chaves opacas para efetuar uma publicação ou uma busca.

Para que possa ser posteriormente procurado através do casamento de cadeias de caracteres, qualquer

dado a ser publicado deve possuir uma descrição. Seja X a seqüência de caracteres dessa descrição. Seja $|X|$ o tamanho da cadeia X; Y a subcadeia procurada com tamanho $|Y|$, e n o tamanho dos n-gramas escolhido.

O algoritmo para a publicação, tanto para sistemas com busca baseada em DHTs quanto para sistemas baseados em inundação que utilizam chaves opacas, como o Freenet, é como se segue:

2.3.3.1 Algoritmo para a publicação

1. Calcule todos os n-gramas de X
2. Calcule as chaves de cada um dos n-gramas utilizando as ferramentas que o arcabouço escolhido disponibiliza. Dependendo do arcabouço escolhido, isso pode ser feito calculando o *hash* SHA-1 de cada n-grama;
3. Crie um filtro de Bloom e adicione todas as chaves calculadas no passo anterior;
4. Para cada uma das chaves calculadas no segundo passo, faça uma publicação na rede cujo valor seja um par contendo o filtro calculado no terceiro passo e o valor propriamente dito (a cadeia X).

Os três primeiros passos são, tipicamente[29], executados em tempo linear em relação ao tamanho da entrada, $|X|$, e não efetuam nenhuma operação de rede. O quarto passo, entretanto, faz, conforme vimos na seção 2.3.2, $|X| - n + 1$ operações de publicação na rede.

2.3.3.2 Algoritmo para a busca

1. Calcule todos os n-gramas de Y
2. Calcule todas as chaves de cada um dos n-gramas utilizando as ferramentas que o arcabouço escolhido disponibiliza;
3. Crie um filtro de Bloom e adicione todas as chaves calculadas no passo anterior;

[29] O passo 2, dependendo do algoritmo de cálculo de chaves utilizado, pode não executar em tempo linear. Entretanto, independentemente do algoritmo utilizado, esse passo é executado localmente.

4. Para cada uma das chaves calculadas no segundo passo, faça uma busca enviando junto o filtro de Bloom criado no passo anterior.

5. Filtre os resultados recebidos verificando o valor do resultado com o critério original de busca Y para eliminar eventuais falsos positivos.

Os passos um, dois, três e cinco são, tipicamente, executados em tempo linear em relação ao tamanho da entrada, $|Y|$ e não efetuam nenhuma operação de rede. O passo quatro efetua $|Y| - n + 1$ operações de busca.

Toda vez que um nó receber uma requisição de busca com um filtro de Bloom, ele valida se possui algum recurso cujo filtro de Bloom contenha o filtro de Bloom recebido[30]. Se possuir, ele responde à requisição da busca. Esse passo de validação entre os dois filtros vai ser executado em todos os nós alcançados pela busca. Em um sistema estruturado, esse passo vai ser executado, no máximo, em $|Y| - n + 1$ nós distintos; ou seja, apenas nos nós que tiverem um dado publicado cuja chave seja igual a chave de algum dos n-gramas de Y. Em um sistema não-estruturado, esse passo será executado por todos os nós alcançados pela busca, o que é determinado pelo TTL de repasses de requisições entre os nós vizinhos.

O número médio de palavras utilizadas para a busca no mecanismo de busca do Yahoo no ano de 2006 foi de aproximadamente 3,3 palavras por busca[31]. A tabela 2 mostra a lista completa.

Número de palavras por busca	**Percentual do número total de buscas**
1	22%
2	30%
3	24%
4	15%
5	9%
6 ou mais	8%

Tabela 2: Percentual do número de buscas em relação ao número de palavras utilizadas nas pesquisas feitas no mecanismo de busca do Yahoo.

[30] Sejam $A = (a_1, a_2, ..., a_m)$ e $B = (b_1, b_2, ..., b_m)$ dois filtros de Bloom com m bits cada. Dizemos que B está contido em A se $\forall i, 1 \leq i \leq m, b_i = 1 \Rightarrow a_i = 1$

[31] Dados obtidos a partir do relatório anual para acionistas que pode ser obtido em: `http://yhoo.client.shareholder.com/downloads/2006AnalystDay.pdf`

Com auxílio de alguns textos digitalizados de obras literárias em português[32], chegamos a um valor estimado do tamanho médio de uma palavra que é de 6,01 letras. Esse valor está bem próximo do valor do tamanho médio das pesquisas em português mais efetuadas no Google durante o ano de 2005[33], que é de 5,98.

Se levarmos em conta que as buscas feitas em um sistema utilizando este algoritmo seguem os mesmos padrões apresentados acima, e supondo que utilizemos n-gramas com $n = 3$, podemos ter uma estimativa do número médio de operações efetuadas durante uma busca. Com essas suposições, podemos dizer que uma busca típica efetua aproximadamente 6,01 . 3,3 - 3 + 1 = 17,833 operações de rede.

Durante uma busca, o uso de filtros de Bloom não traz muitas vantagens no que diz respeito a economia de utilização de banda no caso médio. Nessas circunstâncias, o caso médio do texto de busca teria 6,01 . 3,3 = 19,833 caracteres (ignorando os espaços), enquanto um filtro de Bloom razoável teria aproximadamente 160 bits, ou 20 bytes. O real ganho se dá na publicação, pois o filtro funciona como um resumo do conteúdo da descrição que pode ser muito maior do que 20 bytes. Discussões mais aprofundadas sobre as vantagens da utilização de filtros de Bloom em buscas distribuídas podem ser vistas em [10, 57].

2.3.4 Árvores de Merkle

Em um sistema P2P, freqüentemente, deseja-se distribuir algum dado entre os nós participantes para se ter distribuição de carga e/ou tolerância a falhas através de replicação e redundância. Nesse tipo de sistema também é comum que alguns nós, intencionalmente ou não, acabem corrompendo parte dos dados.

Se for desejável que exista sempre um número mínimo de réplicas de um dado na rede, é razoável supor-se que esse dado esteja dividido em múltiplos segmentos, de forma que a replicação possa ser feita de uma maneira eficiente de um nó para o outro. Por exemplo, se um nó que possui um determinado segmento sai da rede, os nós remanescentes que possuem esse segmento podem repassá-lo para um outro nó; ou seja, apenas

[32] Os textos analisados foram: O Crime do Padre Amaro de Eça de Queiros, Dom Casmurro e Memórias Póstumas de Brás Cubas de Machado de Assis, Clara dos Anjos de Lima Barreto, O Cortiço de Aluísio de Azevedo, Espumas Flutuantes de Castro Alves e Os Sertões de Euclides da Cunha. Os textos foram obtidos na Biblioteca Virtual do Estudante de Língua Portuguesa da Escola do Futuro da USP (`http://www.bibvirt.futuro.usp.br/`)

[33] `http://www.google.com/intl/en/press/intl-zeitgeist.html`

aquele segmento precisa ser transmitido pela rede para manter o número mínimo de réplicas. Supondo que os segmentos de um dado estão distribuídos pela rede de uma maneira uniforme, essa operação não é muito dispendiosa. Se a segmentação não fosse feita e o dado fosse muito grande, a cada vez que um nó saísse da rede os custos para manter o número de réplicas poderiam se tornar proibitivos. A segmentação traz consigo a vantagem de permitir que um eventual interessado no dado possa obter concorrentemente os segmentos do dado de diversos nós, aumentando assim a vazão.

Uma árvore de Merkle [50] é uma estrutura de dados que, quando adaptada para funcionar em um ambiente distribuído, como sistemas P2P de compartilhamento de dados, tem diversas características interessantes. Ela oferece a segmentação dos dados e validação de integridade do conteúdo dos dados obtidos. Essa validação dos dados pode ser feita de uma forma tal que cada segmento de dados tenha uma validação independente, e a validação de um ou mais segmentos possa ser feita antes de se obter o conjunto de todos os segmentos do dado da rede.

Uma árvore de Merkle é uma árvore (geralmente binária) onde as folhas são os segmentos do dado. Cada segmento do dado tem o seu *hash* calculado, e o *hash* da combinação (concatenação ou qualquer outra operação entre os *hashes*) desses *hashes* passa a ser o valor do nó superior a esses nós folhas. Através da combinação dos valores dos *hashes* dos nós irmãos, vai-se criando uma estrutura de árvore binária até chegar ao nó raiz, também chamado de *hash* mestre. A Figura 9 ilustra o processo.

A adaptação dessa estrutura para um sistema P2P baseado em publicações de pares chave-valor é praticamente direta. Após decidir qual é o tamanho do segmento desejado, basta calcular os *hashes* de cada segmento e montar a árvore. Para cada um dos nós dessa árvore, a começar pela raiz, publica-se na rede os pares chave-valor. O publicador precisa apenas guardar o valor do *hash* mestre para posteriormente obter o dado da rede.

Para obter o dado de volta da rede P2P, faz-se uma busca pelo valor publicado com o *hash* mestre. A partir dele já se tem acesso às chaves das publicações do segundo nível da árvore. Através da aplicação recursiva desse processo, chega-se às publicações folha da árvore de Merkle que contêm o dado desejado. Após a primeira busca pelo valor publicado com o *hash* mestre, pode-se fazer concorrentemente diversas

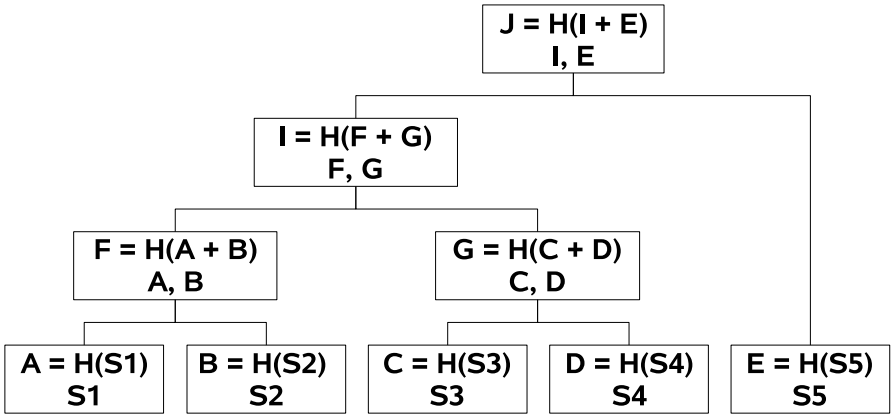

Figura 9: Uma árvore de Merkle. Na primeira linha de cada nó da árvore está a sua chave, ou o valor do *hash*, e na segunda os valores armazenados naquele nó da árvore. A árvore de Merkle é, essencialmente, constituída pela primeira linha dos nós. O valor na segunda linha é uma adição feita à árvore para fazer a sua implementação de uma maneira distribuída em uma rede P2P. S1, S2, S3, S4 e S5 são os segmentos do dado, H(X) é o valor da aplicação da função de *hash* sobre X, e o sinal de + denota a concatenação, ou qualquer outra operação que se desejar, entre dois valores de *hash*. Na figura, J é o valor do *hash* mestre.

buscas e obtenções de valores, o que, além de dividir a carga entre os nós da rede, acelera a obtenção do dado propriamente dito.

Redes P2P estruturadas geralmente utilizam funções de *hash* seguras para o cálculo das chaves. Isso garante que a distribuição dos segmentos se dê de uma forma uniforme entre os nós da rede. Garante também que não sejam feitas adulterações no conteúdo do dado, uma vez que, para fazer isso de forma que o nó que está obtendo o dado não perceba, ter-se-ia que gerar um segmento de dado cujo *hash* fosse o mesmo do dado original. Como a função de *hash* escolhida é segura, isso tem um custo proibitivo. Além disso, redes estruturadas geralmente contam com uma DHT que cuida automaticamente de todo o trabalho de replicação e manutenção de cada publicação/segmento.

Em uma rede P2P não-estruturada, ainda que seja possível utilizar uma árvore de Merkle para fazer o armazenamento distribuído dos dados, a implementação já não é tão direta. A garantia de distribuição uniforme dos segmentos do dado pelos nós da rede, assim como a replicação, não são tão fáceis de se obter. Ainda assim, com alguma distribuição por entre os nós da rede, é possível obter paralelismo para a obtenção

de cada um dos segmentos. A segurança quanto à integridade dos dados não é afetada. Todas as garantias que se tinha na rede estruturada continuam sendo válidas em uma rede não-estruturada.

3 O Hermes

A seguinte proposta também é adequada aos sistemas não-distribuídos. Entretanto, doravante nos concentraremos nos sistemas totalmente distribuídos (estruturados e não-estruturados), pois é neles que a implementação não é trivial.

3.1 Motivação

Conforme visto no capítulo 2, existem diversos tipos de rede. Cada um desses tipos possui certas peculiaridades. Dependendo do perfil da aplicação, um tipo de rede pode ser mais adequado que o outro. Determinar qual tipo de rede é o mais apropriado para a construção de um novo sistema pode não ser fácil e a escolha incorreta da arquitetura pode comprometer o bom funcionamento do sistema.

Todos os sistemas P2P utilizam duas operações básicas para funcionar: busca, por um nó ou um recurso publicado na rede, e troca de mensagens entre os nós [34]. Estas duas operações estão presentes em todas as redes já citadas.

Atualmente, cada um dos arcabouços de programação P2P disponibiliza uma API própria. Elas definem diversas operações e serviços semelhantes, entretanto trazem diferenças sutis no seu funcionamento. Para desenvolver uma aplicação, o desenvolvedor precisa avaliar, de antemão, qual dos arcabouços disponíveis é o mais apropriado para o seu caso. Essa avaliação deve ser feita levando em consideração as idiossincrasias de cada um dos arcabouços. Feita a escolha e a aplicação implementada, o desenvolvedor se vê preso a uma escolha feita no início do projeto, pois é custoso ou trabalhoso mudar de idéia após a implementação.

Freqüentemente, durante as fases iniciais de criação de uma aplicação, os desenvolvedores não têm todas as informações necessárias para fazer a melhor escolha dentre os arcabouços disponíveis. Dados sobre o perfil de utilização ou mesmo os requisitos da aplicação não estão completamente claros. Assim, não é desejável que o desenvolvedor gaste tempo se preocupando com os detalhes da infra-estrutura da rede. O

[34] Rigorosamente, apenas a troca de mensagens (o que inclui o seu roteamento) seria suficiente para a implementação de um sistema completo P2P (por exemplo, o Gnutella trabalha sobre HTTP, o Bamboo sobre TCP/IP e UDP/IP). Entretanto, praticamente todos os arcabouços de programação P2P fornecem uma funcionalidade de busca que geralmente está fortemente acoplada ao seu esquema de roteamento e topologia virtual da rede, de forma a obter resultados de busca melhores do que seriam possíveis através de uma implementação baseada apenas no seu serviço de troca e roteamento de mensagens.

ideal seria se ele pudesse se limitar a pensar na implementação das regras da sua aplicação. Para alcançar isto, é necessária a existência de uma camada de software que o isole das camadas de nível mais baixo, enfim, uma API para que a programação se desenvolva de forma natural e isolada da arquitetura de rede escolhida.

Além de agilizar o desenvolvimento, essa API possibilitaria ao desenvolvedor testar, *a posteriori*, qual tipo de rede mais se adequa ao seu sistema, ou seja, o desenvolvedor ganha o poder de adiar a decisão sobre a infra-estrutura de rede. Esse ganho é crucial pois a avaliação das características de utilização do sistema, antes da sua efetiva implantação, pode ser muito difícil de ser feita. Ademais, as características de utilização podem mudar com o tempo e decisões sobre a infra-estrutura tomadas anteriormente podem deixar de ser as mais apropriadas.

3.2 Proposta

A proposta é a criação de uma API poderosa o suficiente para suprir as necessidades dos desenvolvedores de aplicações P2P através de um mapeamento sobre os arcabouços P2P já existentes. Isto torna possível a troca de um arcabouço por outro, de uma forma totalmente transparente para a aplicação, a menos da alteração de um arquivo de configuração para a escolha do arcabouço. Assim, as aplicações que fizerem uso desta API poderão, sem alterações em seu código, se beneficiar de novas arquiteturas que possam ser criadas e de novas versões das arquiteturas já utilizadas. A API proposta dá ao desenvolvedor a liberdade de escolher, a qualquer momento, a arquitetura que melhor se adequa ao seu caso. Demos a essa API o nome Hermes.

A figura 10 mostra a arquitetura do Hermes, bem como a organização em camadas de uma aplicação que o utiliza.

Na camada mais alta fica a aplicação do usuário, a qual se comunica apenas com a camada inferior a ela, o Hermes. Este, por sua vez, emprega um módulo de mapeamento, que é específico para uma certa rede de sobreposição. O módulo de mapeamento converte chamadas à API do Hermes em chamadas à API da rede de sobreposição escolhida. Se esta rede não oferecer suporte a todas as operações disponíveis na

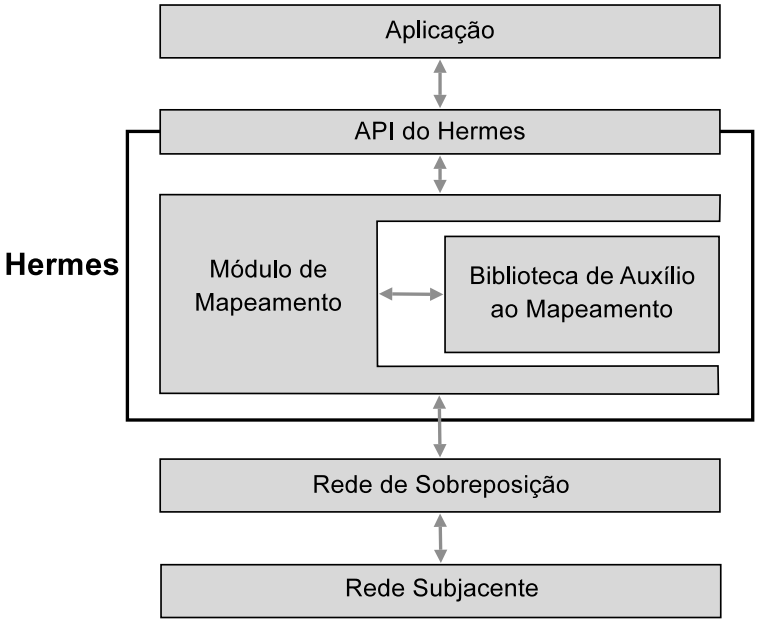

Figura 10: Arquitetura do Hermes

API do Hermes (e.g. busca aproximada), um mapeamento direto não poderá ser efetuado. Nesse caso, o módulo de mapeamento faz uso de uma biblioteca de classes auxiliares, que implementa soluções para os problemas mais comuns encontrados durante o mapeamento de uma nova rede de sobreposição para o Hermes. A biblioteca de auxílio ao mapeamento utiliza os serviços básicos oferecidos por todas as redes de sobreposição para implementar certas funcionalidades não disponíveis em algumas dessas redes. O acesso à rede subjacente (e.g. TCP/IP ou UDP/IP) fica exclusivamente a cargo da rede de sobreposição escolhida.

Como prova de conceito, foi desenvolvida uma API em Java que possui, entre outras, duas operações fundamentais: a troca de mensagens e a busca por recursos. Os detalhes dessa API estão descritos na seção 3.3. Foram implementados mapeamentos para quatro arcabouços diferentes: Bamboo[35], FreePastry[36], SimpleFlood[37] e JXTA [27]. Os dois primeiros são arcabouços para redes distribuídas estruturadas, enquanto

[35]The Bamboo Distributed Hash Table - `http://bamboo-dht.org/`

[36]Pastry - A substrate for peer-to-peer applications - `http://freepastry.org/`

[37]O SimpleFlood é uma implementação de arcabouço de programação P2P que faz parte deste projeto enquanto os demais são implementações de terceiros. O SimpleFlood é uma implementação bem simples, que não visa servir como infra-estrutura de rede para uma aplicação real. De fato, o seu funcionamento é bem semelhante ao de outros arcabouços não-estruturados como o Gnutella. Entretanto, diferentemente deste, o SimpleFlood não possui quaisquer otimizações, o que faz dele uma infra-estrutura de rede ainda menos escalável. A motivação por trás da criação do SimpleFlood é a demonstração e teste do funcionamento

os dois últimos são redes distribuídas não-estruturadas. Foram desenvolvidas também duas aplicações de demonstração, o Hermes Messenger, um *Instant Messenger* construído sobre o Hermes, e o TiWarriors, que é a adaptação de um jogo em rede simples que utilizava TCP/IP como camada de comunicação para passar a utilizar a API do Hermes. Nessas aplicações pode-se, sem alterações em seus códigos, experimentar a troca entre os mapeamentos dos quatro arcabouços disponíveis.

O código do Hermes, incluindo os mapeamentos para as quatro redes de sobreposicão e as duas aplicações de exemplo, pode ser obtido a partir de `http://sourceforge.net/projects/hermesp2p`.

3.3 A API do Hermes

Para possibilitar a criação dos mais diversos tipos de aplicação, a API do Hermes fornece serviços de troca de mensagens entre os nós da rede, busca por publicações, comunicação em grupo e armazenamento distribuído. A seguir esses serviços são descritos para em seguida apresentarmos como essa API foi mapeada para a linguagem de programação Java.

3.3.1 Serviços disponíveis

Apenas os serviços de troca de mensagens e a busca são suficientes para a criação de uma aplicação P2P. Ainda assim, visando facilitar o trabalho do programador de uma aplicação P2P que utiliza o Hermes, criamos alguns serviços adicionais cujas descrições vêm a seguir.

3.3.1.1 Troca de mensagens Em qualquer aplicação P2P pura não existe um nó participante da rede que seja mais importante que outro. Para manter o seu funcionamento a rede se baseia na troca de mensagens entre os seus participantes, sem distinção. Logo, um serviço de troca de mensagens entre os nós se torna indispensável. O Hermes fornece duas maneiras diferentes para um nó se comunicar com o outro. A primeira maneira é mais simples e rápida, não garantindo a entrega ou a ordem. O seu funcionamento é bem semelhante ao funcionamento do UDP/IP. O envio de pacotes de dados ocorre com quase nenhuma

da API do Hermes sem a ele ter associado o peso gerado pela utilização de um arcabouço mais robusto e completo como o Bamboo ou FreePastry.

sobrecarga para controle, funcionando com a política do melhor esforço. A segunda maneira garante a ordem e a entrega. Mas para isso gera uma pequena sobrecarga (assim como no caso do TCP/IP quando comparado ao UDP/IP).

3.3.1.2 Busca Uma aplicação P2P funciona por meio da troca de mensagens entre os nós participantes da rede. Entretanto, um problema claro é para quem enviar cada uma das mensagens. É razoável pensar que as mensagens devem ser enviadas para os nós que possuem um determinado recurso que nos interessa. No contexto de uma aplicação de IM (*Instant Messaging*) o recurso pode ser o usuário conectado, enquanto que, em uma aplicação de compartilhamento de arquivos, os recursos podem ser cada um dos arquivos compartilhados. Encontrar os nós que possuem os recursos desejados pode não ser uma tarefa muito fácil de ser feita apenas com a troca de mensagens entre os nós participantes da rede. Por isso, o Hermes disponibiliza uma interface de busca pelos nós que possuem um determinado recurso. O Hermes provê busca através de chaves únicas, criadas pela própria aplicação, ou através de buscas aproximadas de texto.

3.3.1.3 Comunicação em grupo Para diversas aplicações, a existência de um serviço de comunicação em grupo é de grande valia. Uma aplicação de *Instant Messaging*, por exemplo, pode utilizar esse recurso para montar uma conversa com diversos participantes, enquanto um jogo multi-jogadores pode utilizar-se disso para distribuir o estado do jogo entre os diversos jogadores de uma maneira otimizada. Para esses casos, o Hermes disponibiliza uma interface de comunicação em grupo simples, porém poderosa. A criação e a assinatura de um grupo são feitas através de uma única operação que recebe apenas o nome do grupo. O cancelamento da assinatura é igualmente simples. A entrega das mensagens enviadas para o grupo é feita com a política do melhor esforço, o que é relativamente comum nos serviços de comunicação em grupo disponíveis em outros arcabouços.

3.3.1.4 Armazenamento distribuído Em diversos tipos de aplicações, é muito comum querer salvar antes da finalização da execução alguns dados para serem utilizados durante a próxima execução. Em uma aplicação P2P como um IM, por exemplo, a lista de amigos é um desses dados. Nesse caso, é muito

comum o usuário se conectar de diferentes máquinas, o que faz o salvamento dos dados num dispositivo de armazenamento local uma opção não muito boa. Armazenar a lista de contatos na rede e possibilitar que o usuário a obtenha mais tarde por meio de qualquer um dos nós da rede é justamente o que esse serviço do Hermes provê. Quando um nó tem um conjunto de informações que interessam a muitos outros nós, esse serviço do Hermes possibilita também que o nó coloque as informações na rede, ao invés de transmití-las individualmente para cada um dos outros nós. Dessa forma a carga da distribuição das informações é dividida entre diversos nós da rede.

3.3.1.5 Serviços oferecidos por outros arcabouços

A Tabela 3 compara os serviços oferecidos pelo Hermes com os serviços oferecidos por outros arcabouços de programação de aplicações P2P.

	SimpleFlood	**JXTA**	**FreePastry**	**Bamboo**	**Hermes**
Troca de mensagens	•	•	•	•	•
Busca exata	•	•	•	•	•
Busca aproximada		•			•
Comunicação em grupo	•	•	•	•	•
Armazenamento distribuído			•	•	•
Autenticação de usuários		•			

Tabela 3: Comparação entre os serviços oferecidos pelo Hermes e por alguns outros arcabouços de programação de aplicações P2P.

O Hermes fornece serviços que não estão disponíveis em todas as infra-estruturas de redes mapeadas. Para fazê-lo, cada um dos mapeamentos feitos para o Hermes tem o trabalho adicional de fazer com que a API seja respeitada. Isso nem sempre é um trabalho tão simples, como será discutido mais adiante.

3.3.2 Detalhamento

A API é bem simples de ser utilizada e tem como principais integrantes as seguintes interfaces:

`Network` Representa a rede sobre a qual a aplicação está sendo executada. Aqui ficam definidas operações necessárias para a conexão e desconexão, assinaturas e cancelamento de assinaturas de grupos, além da publicação, retirada e busca de recursos publicados na rede.

`ID` Toda informação que é publicada na rede através da interface `Network` deve ter ao menos um identificador. Cada um destes identificadores é representado por essa interface e pode ser criado utilizando-se a interface `IDFactory`. Na maior parte das vezes, esse identificador acaba sendo, na verdade, um vetor de bytes gerado por uma função de *hash* segura. É a função desta interface esconder este tipo de detalhe do desenvolvedor da aplicação, de modo que ele veja estes `ID`s como objetos opacos.

`ResourceDescriptor` Todas as publicações e resultados de busca na rede feitas através das operações definidas pela interface `Network` se utilizam desta interface. Esta interface traz consigo informações sobre um recurso. Essas informações incluem os `ID`s do recurso, seu tipo e um texto com uma breve descrição. Objetos desse tipo são criados pela interface `ResourceDescriptorFactory`.

`Node` O resultado de uma busca com sucesso efetuada na rede é um objeto do tipo `SearchResult` que contém dois dados importantes: o `ResourceDescriptor` do recurso encontrado e um objeto `Node` que representa o nó da rede onde este recurso está localizado. É possível que vários `Node`s representem o mesmo nó físico da rede. Isso se deve ao fato de alguns arcabouços P2P criarem vários nós lógicos no mesmo nó físico, para que haja uma distribuição mais uniforme da carga pelos nós físicos da rede [57]. As únicas maneiras de se obter referências para objetos deste tipo são através da interface `Network`, que dá acesso ao nó local, e através da busca por algum recurso na rede pela mesma interface. Essa interface dispõe de operações para o envio de mensagens e requisição de recursos de um determinado nó.

`Group` O Hermes fornece um serviço de comunicação em grupo baseado em assinaturas. Quando se deseja participar de um grupo, basta assiná-lo informando ao Hermes o nome do grupo. A resposta a um pedido de assinatura é um objeto do tipo `Group`. Esse objeto pode ser utilizado para enviar mensagens aos assinantes do grupo. Além disso, esse objeto é necessário para efetuar o cancelamento da assinatura.

`ApplicationDataReceiver` Esta interface é a responsável pela comunicação entre o Hermes e a aplicação do usuário. Quando a aplicação se conecta à rede, ela deve fornecer ao Hermes um objeto

deste tipo. Esse objeto é notificado toda vez que uma mensagem ou requisição para o nó local for recebida.

Um diagrama simplificado, em UML, dessas interfaces pode ser visto na Figura 11.

Todas as operações definidas pela API que envolvem comunicação entre nós (e algumas outras operações relacionadas a estas tarefas) são não bloqueantes. Isso se deve ao fato da maioria dos arcabouços terem sido programados de uma forma orientada a eventos e utilizarem um modelo de comunicação assíncrona. Mais detalhes sobre o funcionamento desse tipo de sistema podem ser vistos em [59]. Um esquema de *callbacks*, fortemente inspirado naquele utilizado pelo FreePastry [31], foi implementado para evitar que o desenvolvedor da aplicação tenha a necessidade de manter um mecanismo (possivelmente com vários *threads*) de verificação de chegada de respostas a uma requisição. Por influência daquele sistema, os objetos de *callback* utilizados no Hermes receberam o nome de `Continuation`.

As definições das interfaces mais importantes são apresentadas a seguir. Algumas definições e trechos de código foram omitidos em prol da clareza.

3.3.2.1 A interface `ID`

```
interface ID extends java.io.Serializable {}
```

A interface `ID` representa o identificador de algum recurso. Esse identificador é gerado pelo arcabouço escolhido. A geração de `ID`s pode ser feita através da interface `IDFactory`. Esta interface possui operações de criação de `ID`s a partir de um vetor de bytes e a partir de um texto de descrição.

3.3.2.2 A interface `Network`

```
interface Network {
  void init(ApplicationDataReceiver dr);
  void connect();
  void disconnect();
  Node getLocalNode();
```

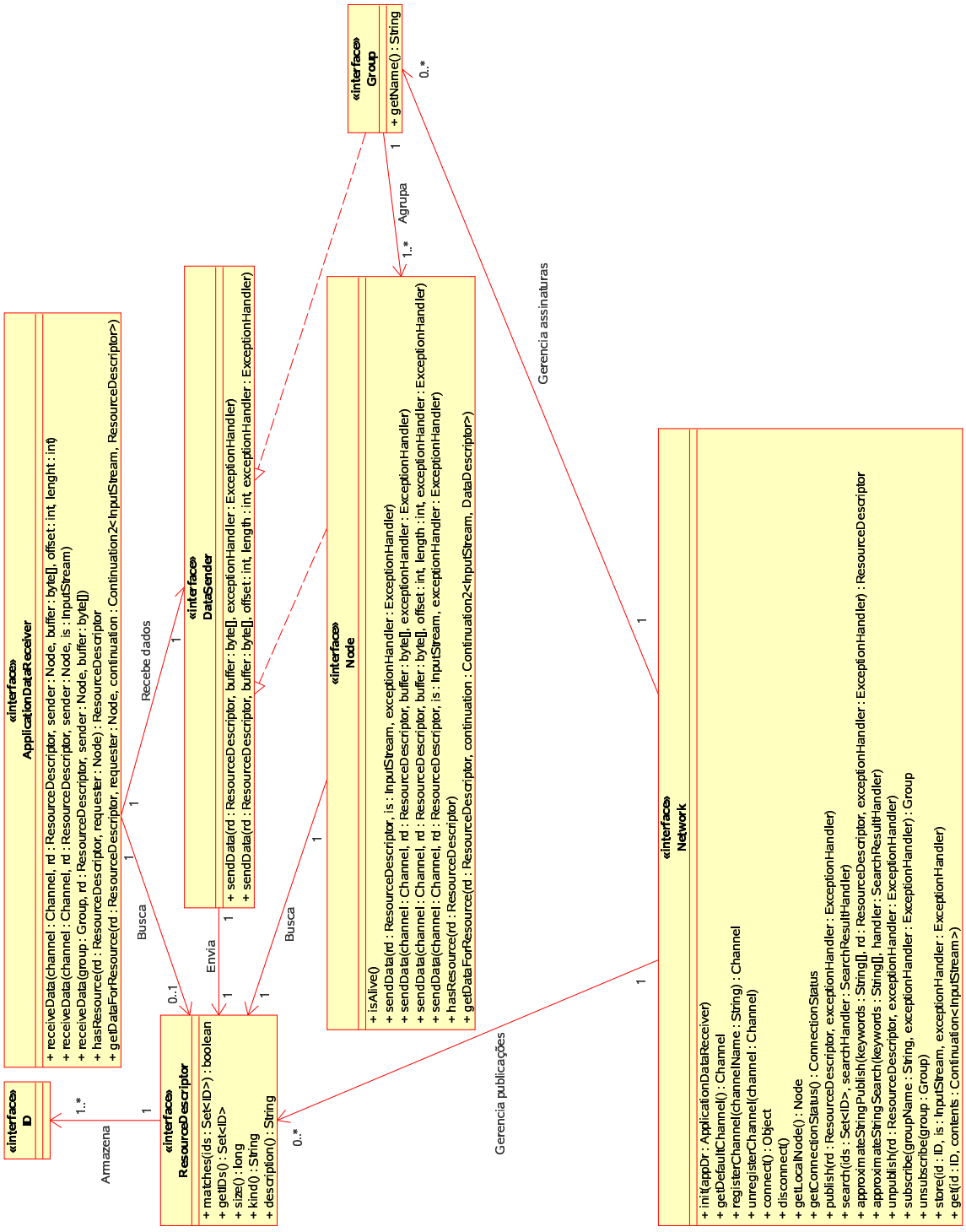

Figura 11: Diagrama simplificado, em UML, das principais interfaces componentes do Hermes.

```
    ConnectionStatus getConnectionStatus();

    Channel getDefaultChannel();
    Channel registerChannel(String channelName);
    void unregisterChannel (Channel channel);

    void publish (ResourceDescriptor rd, ExceptionHandler exHandler);
    ResourceDescriptor approximateStringPublish (String[] keywords,
          ResourceDescriptor rd, ExceptionHandler exHandler);
    void unpublish (ResourceDescriptor rd, ExceptionHandler exHandler);
    void search(Set<ID> ids, SearchResultHandler handler);
    void approximateStringSearch (String[] keywords,
          SearchResultHandler handler);

    Group subscribe (String groupName,
          ExceptionHandler exceptionHandler);
    void unsubscribe (Group group);

    void store (ID id, InputStream is, ExceptionHandler exHandler);
    void get (ID id, Continuation<InputStream> contents);
}
```

Operações da interface Network:

`init` Carrega o arcabouço escolhido pelo arquivo de configuração e faz quaisquer inicializações que esse arcabouço possa requerer antes de se conectar à rede. O canal de comunicação entre o Hermes e a aplicação do usuário é representado por uma instância de `ApplicationDataReceiver` que é recebida como parâmetro da operação.

`connect` Conecta-se à rede. Deve ser chamado após a chamada do `init`.

`disconnect` Desconecta-se da rede.

`getLocalNode` Devolve uma referência ao objeto `Node` que representa o nó local.

`getConnectionStatus` Devolve um objeto do tipo `ConnectionStatus` que representa o atual estado de conectividade da rede.

`getDefaultChannel` Toda comunicação feita através do Hermes se dá por meio de canais. Existe um canal padrão de comunicação que não exige um registro prévio para possibilitar o envio e o recebimento de mensagens. Este método devolve uma referência para um objeto do tipo `Channel` que representa esse canal.

`registerChannel` Uma aplicação construída sobre o Hermes pode escolher separar o envio e recebimento das mensagens por canais. Para fazê-lo, basta que ela registre um canal diferente para cada tipo de mensagem. Para registrar um novo canal e passar a ser capaz de enviar e receber mensagens por esse novo canal, este método deve ser utilizado. Ele recebe como parâmetro o nome do canal a ser registrado e devolve um objeto do tipo `Channel` que deve ser utilizado no momento do envio das mensagens.

`unregisterChannel` Se a aplicação em algum momento achar que é desnecessário o seu cadastro em qualquer um dos canais, o descadastramento pode ser feito utilizando este método. Recebe como parâmetro o objeto `Channel` representando o canal do qual se deseja ser descadastrado.

`publish` Publica na rede o recurso descrito pelo parâmetro. Após a publicação, esse dado pode ser encontrado pela operação `search`. Dependendo do arcabouço utilizado, a publicação não é instantânea e, portanto, a chamada não é bloqueante.

`approximateStringPublish` Publica na rede utilizando as palavras-chave do parâmetro `keywords` o recurso descrito pelo parâmetro `rd`. Os identificadores do `ResourceDescriptor` passado como parâmetro são ignorados. Os identificadores utilizados para a publicação são recalculados com base nas palavras-chave passadas como parâmetro. O `ResourceDescriptor` efetivamente utilizado

para a publicação é devolvido pelo método. A operação `approximateStringSearch` permite que o recurso, após sua publicação, seja encontrado na rede. Dependendo do arcabouço utilizado, a publicação não é instantânea e, portanto, a chamada desse método não é bloqueante.

`unpublish` Retira da rede algum dado publicado. A retirada do dado da rede não é instantânea e, dependendo do arcabouço sendo utilizado, pode levar alguns minutos até que ela seja completada. Por esse motivo, esta chamada não é bloqueante. Recebe como parâmetro o `ResourceDescriptor` utilizado para a publicação. Para retirar a publicação de um recurso publicado utilizando o método `approximateStringPublish`, deve-se utilizar o `ResourceDescriptor` devolvido durante a publicação, e não aquele passado como parâmetro.

`search` Efetua uma busca na rede por recursos publicados através da operação `publish`. Somente um recurso publicado pode ser encontrado pela busca. A publicação de um recurso envolve o fornecimento, pela aplicação, de um `ResourceDescriptor`. Uma das informações contidas nesse descritor é o conjunto de `IDs` que identificam o recurso sendo publicado. O comportamento padrão da operação `search` é a localização de quaisquer recursos publicados cujos conjuntos de `IDs` sejam superconjuntos do conjunto de `IDs` passado como parâmetro para a busca. Esse comportamento é definido pela implementação de `ResourceDescriptor` em utilização. O Hermes fornece uma implementação padrão, cujo funcionamento é o descrito acima. A aplicação, entretanto, tem a liberdade de alterar esse comportamento, podendo fazê-lo ser mais restritivo, definindo a sua própria implementação da interface `ResourceDescriptor`.

`approximateStringSearch` Efetua uma busca na rede por recursos publicados através da operação `approximateStringPublish` que contenham as palavras-chave passadas como parâmetro. Somente recursos publicados podem ser encontrados pela busca. Por exemplo, se uma publicação utilizar a palavra-chave “abacate” e outra a palavra-chave “alicate”, ambas serão encontradas por uma busca que utilize como palavra-chave “cate”. Entretanto, apenas a primeira será encontrada por uma busca feita com as palavras-chave “acate” ou “abacate”. Dependendo da implementação sendo utilizada, al-

guns falsos positivos podem ocorrer durante uma busca. Fica a cargo da aplicação do usuário avaliar se todos os resultados são apropriados para o seu uso.

`subscribe` O Hermes fornece um serviço de comunicação em grupo. Para assinar um grupo e passar a receber as mensagens que são enviadas a ele, utiliza-se este método passando como parâmetro o nome do grupo no qual se deseja ingressar. O método devolve um objeto do tipo `Group` que é utilizado para enviar as mensagens.

`unsubscribe` Se, a qualquer momento, a aplicação desejar cancelar a assinatura de algum grupo, basta utilizar esse método passando como parâmetro o objeto `Group` recebido no momento da assinatura.

`store` O Hermes fornece um serviço de armazenamento distribuído de dados. Para armazenar algo na rede, basta utilizar esse método informando o `ID` pelo qual esse dado será identificado e um `InputStream` que contenha o dado desejado.

`get` Este método é utilizado para recuperar os dados armazenados na rede através da operação `store`. Recebe como parâmetro o objeto `ID` utilizado para identificar o dado durante o armazenamento e um objeto de *callback* que será utilizado para notificar a aplicação do usuário quando o dado for obtido da rede.

3.3.2.3 A interface `ResourceDescriptor`

```
interface ResourceDescriptor extends java.io.Serializable {
  public boolean matches(Set<ID> ids);
  public Set<ID> getIDs();
  public long size();
  public String kind();
  public String description();
}
```

A interface `ResourceDescriptor` descreve os recursos da aplicação a serem trafegados na rede.

Uma implementação desta interface define como, durante uma busca, será feita a correspondência entre os IDs procurados e os IDs representantes de cada dado. Isso é feito através da operação `matches`. O resultado de uma busca pelo conjunto de IDs A, é o conjunto de todos os ResourceDescriptors encontrados tais que `matches`(A) é verdadeiro. Essa parte funciona como um filtro adicional ao filtro já feito por cada um dos arcabouços, ou seja, pode apenas restringir a gama de resultados mas nunca fazê-la ser mais abrangente.

O Hermes já fornece uma implementação padrão da interface `ResourceDescriptor`, ainda que, conceitualmente, a implementação dessa interface seja de responsabilidade da aplicação. Essa implementação padrão deve ser capaz de atender a maior parte das aplicações. Desta forma, o desenvolvedor só tem a necessidade de criar uma nova implementação em casos bem específicos.

A interface `ResourceDescriptor` define as seguintes operações:

`matches` Fica a critério do desenvolvedor da aplicação definir o comportamento deste método. Qualquer busca feita no Hermes utiliza alguns critérios de restrição. O Hermes, durante a execução de uma busca, utiliza este método para avaliar se um recurso está em conformidade com esses critérios. Para isso, a implementação de `ResourceDescriptor` deve avaliar se há correspondência entre o conjunto de IDs (passado como parâmetro a esta operação) e o conjunto de IDs contidos no `ResourceDescriptor`. As regras utilizadas nessa avaliação são dependentes da implementação do `ResourceDescriptor`. Como a maior parte das aplicações utiliza um comportamento semelhante, o Hermes já fornece uma implementação básica da interface `ResourceDescriptor`. A operação `matches` desta implementação avalia se o conjunto de IDs do `ResourceDescriptor` alvo é um superconjunto do conjunto de IDs passado como parâmetro à operação e, se este for o caso, devolve verdadeiro.

`getIDs` Devolve os IDs que representam este objeto.

`size` Devolve o tamanho, em bytes, do recurso representado por este objeto.

`kind` Devolve o tipo deste recurso. O tipo do recurso é representado de forma textual. O conceito deste

atributo é similar ao de um tipo MIME (*Multipurpose Internet Mail Extensions*). Entretanto, fica a cargo do desenvolvedor da aplicação a definição dos tipos a serem utilizados assim como a sua representação textual.

`description` Devolve uma descrição textual do recurso representado por este objeto.

3.3.2.4 A interface `Node`

```
public interface Node java.io.Serializable {
  boolean isAlive ();

  void sendData(ResourceDescriptor rd,
        byte[] buffer, ExceptionHandler exHandler);
  void sendData(Channel channel, ResourceDescriptor rd,
        byte[] buffer, ExceptionHandler exHandler);
  void sendData(ResourceDescriptor rd,
        InputStream inputStream, ExceptionHandler exHandler);
  void sendData(Channel channel, ResourceDescriptor rd,
        InputStream inputStream, ExceptionHandler exHandler);


  ResourceDescriptor hasResource (ResourceDescriptor rd);
  void getDataForResource(ResourceDescriptor rd,
        Continuation2<InputStream, ResourceDescriptor> continuation);
}
```

A interface `Node` representa um nó lógico participante da rede na qual se está conectado. Esta interface é definida como sendo `Serializable` pois desta maneira a aplicação do usuário tem a liberdade de poder enviar, de uma forma fácil, uma mensagem que contém uma referência a um nó. Isso é algo desejável em algumas aplicações. Esta interface define as seguintes operações:

`isAlive` Avalia se o nó representado pela instância alvo da chamada está respondendo a requisições.

`sendData` Envia os dados do recurso descritos pelo `ResourceDescriptor` e contidos em um vetor de bytes ou em um `InputStream` ao nó representado pela instância alvo da chamada. Existe uma diferença semântica bem clara entre as duas versões da operação `sendData`. No caso do vetor de bytes, o Hermes vai tentar enviar o buffer todo de uma só vez em um único pacote, até um certo limite de tamanho. Já a versão da operação que recebe um `InputStream` fará a transmissão utilizando quantos pacotes forem necessários, até enviar todos os dados. O nó destinatário receberá esses dados pelo método `receiveData` da interface `ApplicationDataReceiver`, que será descrita a seguir. Ocorrerá uma chamada a `receiveData` com um parâmetro do tipo `byte[]`, caso o envio tenha sido feito utilizando um vetor de bytes, ou com um parâmetro do tipo `InputStream`, caso o envio tenha sido feito utilizando este mesmo tipo. O objeto `Channel` enviado como parâmetro será enviado junto com os dados e repassado para a aplicação no nó destinatário da mensagem através do mesmo método, `receiveData`. As versões desta operação que não recebem um objeto `Channel` como parâmetro fazem o envio utilizando o canal padrão de comunicação cuja referência pode ser obtida através da operação `getDefaultChannel` da interface `Network`.

`hasResource` Apesar de um nó ser obrigado a publicar algum recurso na rede para que ele possa ser visível às operações de busca, cada nó pode, a qualquer momento, consultar um outro nó sobre a possessão de qualquer recurso. Isto serve para dois propósitos. O primeiro é permitir que um nó verifique se o resultado de uma busca feita anteriormente continua válido. O segundo é que um nó não necessariamente necessita publicar todos os recursos que ele deseja disponibilizar na rede. Um exemplo de utilização disto é uma aplicação de IM, onde o usuário simplesmente publica o seu nome na rede, mas deixa suas informações pessoais, ou mesmo sua foto, disponíveis apenas para quem as requisitar.

`getDataForResource` Esta é uma chamada não bloqueante que efetua uma conexão com o nó representado pela instância alvo da chamada e traz o dado que representa o recurso descrito pelo

`ResourceDescriptor` recebido, caso esteja disponível. O resultado da operação é informado ao chamador através da continuacão recebida como parâmetro.

3.3.2.5 A interface `Group`

```
public interface Group {
  String getName();
  void sendData(ResourceDescriptor dd, byte[] buffer,
       ExceptionHandler exHandler);
}
```

Uma instância da interface `Group` pode ser obtida através da assinatura de um grupo. Um objeto desse tipo representa a assinatura do grupo e permite o envio de mensagens para os assinantes do grupo. O envio das mensagens para os assinantes é feito de forma assíncrona e não há nenhuma garantia quanto a ordem ou a entrega das mensagens para todos os participantes do grupo. A política utilizada para o envio de mensagens é a do melhor esforço.

A interface `Group` define as seguintes operações:

`getName` Devolve o nome do grupo representado pelo objeto alvo da chamada.

`sendData` Envia os dados recebidos como parâmetro para os assinantes do grupo. Os assinantes serão notificados sobre o recebimento de uma mensagem através de uma chamada ao método `receiveData` da interface `ApplicationDataReceiver`, que é descrita a seguir.

3.3.2.6 A interface `ApplicationDataReceiver`

```
interface ApplicationDataReceiver {
  void receiveData (Channel channel, ResourceDescriptor rd,
       Node sender, byte[] buffer, int offset, int length);
  void receiveData (Channel channel, ResourceDescriptor rd,
       Node sender, InputStream inputStream);
```

```
    void receiveData (Group group, ResourceDescriptor rd,
          Node sender, byte[] buffer);

    ResourceDescriptor hasResource(ResourceDescriptor rd,
          Node Requester);
    void getDataForResource (ResourceDescriptor rd, Node requester,
          Continuation2<InputStream, ResourceDescriptor> continuation);
}
```

A interface `ApplicationDataReceiver` é o canal de comunicação entre o Hermes e a aplicação do usuário. A sua implementação é de responsabilidade do desenvolvedor da aplicação. Essa interface define as seguintes operações:

`receiveData` Existem três versões desta operação. Uma delas é responsável por notificar a aplicação do usuário sobre o recebimento de uma mensagem de um grupo do qual ela é assinante. Essa versão recebe como parâmetro o grupo pelo qual a mensagem foi recebida, além da mensagem propriamente dita. As duas versões remanescentes são responsáveis pela notificação da aplicação do usuário sobre o recebimento de uma mensagem enviada diretamente por outro nó. Existe uma diferença semântica bem clara entre essas duas versões. Uma recebe um vetor de bytes e a outra recebe um `InputStream`. Esta é utilizada para o envio de informações muito grandes que têm de ser fragmentadas (o Hermes garante a ordem e a entrega desses fragmentos de forma transparente ao usuário); aquela é utilizada para o envio de mensagens curtas de forma mais rápida e com menos sobrecargas com o envio de informações de controle, em contrapartida, não oferece a garantia de ordem ou entrega das mensagens. Ambas recebem como parâmetro um objeto do tipo `Channel` que indica o canal pelo qual a mensagem foi recebida.

`hasResource` O Hermes chama esta operação quando necessita saber se a aplicação possui um determinado recurso. A operação `hasResource` definida em `Node` é de fato uma chamada remota para esta operação.

`getDataForResource` O Hermes chama esta operação quando um nó remoto lhe requisita o envio de um determinado dado. Uma vez encontrado o dado, a aplicação executa um método da continuação passada como parâmetro, que por sua vez empacota o dado requisitado e o envia ao nó requisitante. A chamada a este método pode retornar (e tipicamente retornará) imediatamente.

3.4 Construindo uma aplicação sobre o Hermes

As aplicações construídas com o Hermes têm três estágios: inicialização, execução e finalização. A inicialização e finalização são estágios passageiros, enquanto a execução é um estágio mais duradouro. Durante a inicialização são carregados os arquivos de configuração que especificam a infra-estrutura P2P a ser utilizada, juntamente com suas configurações apropriadas. Em seguida, a rede é inicializada e a conexão com a rede é efetuada. Passado o estágio de inicialização, entra-se no estágio de execução. Nesse estágio é possível efetuar buscas na rede e trocar mensagens com os outros nós. É nele que a aplicação faz o seu trabalho. Quando a aplicação, por qualquer motivo, decidir que é hora de finalizar ou se desconectar, ela entra no estágio de finalização. Para a maioria das infra-estruturas, simplesmente finalizar a execução do programa não é a melhor opção: tabelas de apontadores acabam ficando sujas e algoritmos de estabilização geralmente desenhados para tratar falhas acabam tendo que entrar em ação, criando uma carga adicional, que poderia ser evitada, na rede. Por isso algumas infra-estruturas provêem mecanismos para a desconexão voluntária e o Hermes disponibiliza esses mecanismos para a aplicação.

3.4.1 Inicialização

Para inicializar o arcabouço escolhido e fazer as configurações necessárias, o Hermes utiliza-se das seguintes propriedades do ambiente de execução Java.

`hermes.NetworkFactoryImplementationClass` Define o nome da classe fábrica (*Abstract Factory* [25]) do objeto `Network`. Atualmente existem quatro fábricas[38] representando os diferentes mapeamentos para estes arcabouços:

[38] Estas fábricas estão contidas no pacote `org.hermes.networks`

- `simpleflood.mapping.SimpleFloodNetworkFactory`, para SimpleFlood
- `jxta.mapping.JxtaNetworkFactory`, para JXTA
- `freepastry.FreePastryNetworkFactory`, para FreePastry
- `bamboo.BambooNetworkFactory`, para Bamboo

`hermes.ResourceDescriptorFactoryImplementationClass` Define o nome da classe responsável pela criação de instâncias da implementação da interface `ResourceDescriptor`. O Hermes já oferece uma classe básica[39] com o comportamento mais comumente utilizado. Entretanto, a aplicação fica livre para criar a sua própria fábrica, caso queira definir sua própria implementação de `ResourceDescriptor`, para mudar o comportamento padrão das buscas.

`hermes.IDFactoryImplementationClass` Cada um dos arcabouços utiliza um método diferente para o cálculo dos identificadores dos recursos publicados na rede. Alguns, como o Bamboo, utilizam uma função de *hash* segura (como o SHA-1) sobre um conjunto de dados qualquer. Outros, como o SimpleFlood, utilizam simplesmente uma palavra-chave como um identificador. A geração dos identificadores pela aplicação precisa ser independente do arcabouço escolhido. Essa é a função desta fábrica. Já existem quatro fábricas disponíveis, representando os diferentes mapeamentos para estes arcabouços:

- `org.hermes.networks.simpleflood.mapping.SimpleFloodIDFactory`, para SimpleFlood
- `org.hermes.networks.jxta.mapping.JxtaIDFactory`, para JXTA
- `org.hermes.networks.freepastry.FreePastryIDFactory`, para FreePastry
- `org.hermes.networks.bamboo.BambooIDFactory`, para Bamboo

Essas são todas as configurações necessárias para uma aplicação inicializar o Hermes e se conectar à rede. O Hermes já traz consigo arquivos de configuração para cada um dos arcabouços. Esses arquivos de

[39] `org.hermes.implementation.BasicResourceDescriptorFactory`

configuração devem funcionar na maioria dos casos de aplicações. O desenvolvedor, salvo por algum motivo muito específico, não necessita alterar quaisquer configurações de cada um dos arcabouços mapeados para poder utilizá-los.

Escrevendo o exposto acima em código, o trecho da aplicação responsável pela inicialização é algo como:

```
/*
 * Carrega as configurações escolhidas
 * do arquivo hermes.properties
 */
ClassLoader cl = MinhaAplicacao.class.getClassLoader();
InputStream is = cl.getResourceAsStream("hermes.properties");
System.getProperties().load(is);

//Inicializa o Hermes e se conecta à rede.
NetworkFactory nf = Hermes.getNetworkFactory();
Network network = nf.create();
ApplicationDataReceiver appDR =
        new ApplicationDataReceiverDaMinhaAplicacao();
network.init(appDR);
network.connect();
```

Conforme o exemplo demonstra, a aplicação fica livre de qualquer referência direta à implementação da rede de sobreposição utilizada.

3.4.2 Execução

É neste estágio que a aplicação do usuário faz efetivamente todo o seu serviço. Durante este estágio, a aplicação é capaz de trocar mensagens com outros nós, publicar e buscar recursos na rede, além de armazenar e obter dados armazenados na rede.

Abaixo apresentamos diversos exemplos em código, mostrando como são feitas as operações mais comuns utilizando a API do Hermes. Evidentemente, uma aplicação real precisa fazer mais do que o que será apresentado aqui. Em prol da clareza, suprimimos diversos detalhes, como o tratamento dos dados, o tratamento de erros e o uso das diferentes modalidades de envio de mensagens.

3.4.2.1 Publicação e busca Durante a execução da aplicação, duas operações fundamentais são a publicação e a busca por recursos na rede. A publicação de recursos permite que participantes da rede possam encontrar um determinado recurso e, a partir daí, trocar mensagens entre o publicador e o interessado. O conteúdo da publicação, entretanto, é algo que varia de aplicação para aplicação. Um nó executando uma aplicação de IM pode, por exemplo, publicar o nome do seu usuário, enquanto uma aplicação de compartilhamento de arquivos pode publicar cada um dos seus arquivos.

A API do Hermes fornece duas modalidades de publicação e busca por recursos. A primeira maneira consiste na publicação e busca exatas, que são executadas de uma maneira rápida e muito eficiente. Essa busca geralmente é mapeada diretamente para algum serviço da rede de sobreposição subjacente. A segunda maneira é a publicação e busca aproximada de strings. Essa modalidade é um pouco mais custosa e geralmente não tem um mapeamento direto para a rede de sobreposição subjacente. A vantagem da segunda modalidade é o seu poder de efetuar buscas mais poderosas, capazes de encontrar resultados que de outra forma seriam dificilmente encontrados.

A publicação exata de um recurso pode ser feita da seguinte maneira:

```
/*
 * Publico as informações apropriadas para
 * que os outros nós me encontrem.  Isso é
 * feito através do objeto Network obtido
 * durante o estágio de inicialização.
 */
ResourceDescriptorFactory rdf =
   Hermes.getResourceDescriptorFactory();
```

```
IDFactory idFactory = Hermes.getIDFactory();
Set<ID> ids = new HashSet<ID>();
ids.add(idFactory.createID(ATRIBUTO_PUBLICADO_1));
ids.add(idFactory.createID(ATRIBUTO_PUBLICADO_2));
ResourceDescriptor rd =  rdf.create(
   ids, TAMANHO_DO_RECURSO, DESCRICAO, TIPO);
ExceptionHandler exHandler = new ExceptionHandler() {
  public void handleException(Exception e) {
    //trata a exceção no caso de erros durante a publicação
  }
};
network.publish(rd, exHandler);
```

A busca por um recurso publicado de forma exata pode ser feita da seguinte maneira:

```
/*
 * Toda busca na rede é feita através de IDs.
 * Aqui são criados os IDs que um objeto publicado
 * precisa ter para que seja devolvido pela busca.
 */
Set<ID> ids = new HashSet<ID>();
ids.add(Hermes.getIDFactory()
        .createID(ATRIBUTO_PROCURADO_1));
ids.add(Hermes.getIDFactory()
        .createID(ATRIBUTO_PROCURADO_2));
/*
 * Define qual é o tratamento a ser tomado
 * tanto no caso de sucesso quando de erro
 * durante a busca por um determinado dado.
 */
```

```
SearchResultHandler trataResposta =
  new SearchResultHandler() {
    public void handleResult (SearchResult result) {
      //Trata o resultado
    }
    public void handleException(Exception e) {
      //Trata o erro, caso ele ocorra
    }
    public boolean expired() {
      //Se por algum motivo a aplicação não desejar mais
      //receber os resultados dessa busca, basta fazer com
      //que esse método devolva verdadeiro
      return false;
    }
};
/*
 * Efetua a busca propriamente dita.
 */
network.search(ids, trataResposta);
```

Para publicar um recurso na rede de maneira que ele possa ser encontrado pelas buscas aproximadas, deve-se fazer:

```
ResourceDescriptorFactory rdf =
        Hermes.getResourceDescriptorFactory();
ResourceDescriptor rd =
        rdf.create(null, TAMANHO_DO_RECURSO, DESCRICAO, TIPO);
String[] keywords = {"palavras", "chave", "COM",
        "DISTINÇÃO ENTRE MAIÚSCULAS", "E", "minúsculas"};
ExceptionHandler exHandler = new ExceptionHandler() {
```

```
    public void handleException(Exception e) {
      //trata a exceção no caso de erros durante a publicação
    }
  };
  //É importante manter o ResourceDescriptor devolvido, pois é
  //ele que deve ser utilizado em uma eventual retirada de publicação
  //desse objeto da rede
  rd = network.approximateStringPublish(keywords, rd, exHandler);
```

Para buscar um recurso na rede utilizando os recursos de busca aproximada de strings que o Hermes fornece, deve-se fazer:

```
String[] keywords = {"palavras", "minús"};
SearchResultHandler trataResposta =
  new SearchResultHandler() {
    public void handleResult (SearchResult result) {
      //Trata o resultado
    }
    public void handleException(Exception e) {
      //Trata o erro, caso ele ocorra
    }
    public boolean expired() {
      return false;
    }
};
network.approximateStringSearch(keywords, trataResposta);
```

Independentemente da modalidade utilizada para a publicação de um recurso, basta executar o seguinte trecho de código para remover a publicação de um recurso da rede. Dependendo da rede de sobreposição utilizada, a retirada das publicações não é instantânea, e algumas buscas feitas, mesmo após a chamada para

efetuar a retirada de publicação, podem trazer como resultado esse recurso por algum tempo[40].

```
ExceptionHandler exHandler = new ExceptionHandler() {
  public void handleException(Exception e) {
    //trata exceções ocorridas durante a retirada da publicação
  }
};
network.unpublish(rd, exHandler);
```

3.4.2.2 Comunicação entre nós Para se enviar uma mensagem de um nó a outro utilizando a API do Hermes, é necessário ter-se uma referência ao nó destinatário. Existem duas maneiras de se obter uma referência a um nó diferente do nó local. A primeira maneira envolve a busca por algum recurso que esse nó publicou. O resultado de qualquer busca é representado por um objeto do tipo `SearchResult`. Este objeto possui uma operação (`getNode`) que devolve uma referência para o nó que publicou aquele recurso na rede. A segunda maneira é através do recebimento de uma mensagem de algum outro nó participante da rede. Sempre, juntamente com a mensagem, o Hermes informa à aplicação do usuário o nó remetente.

Todas as mensagens transmitidas pelo Hermes têm um canal de transmissão associado. Mensagens enviadas sem a especificação de um canal são enviadas por um canal padrão. Se a aplicação do usuário desejar separar o recebimento e o envio de mensagens em canais diferentes do canal padrão, basta fazer o registro do canal desejado da seguinte maneira:

```
Channel canal = network.registerChannel("NomeDoCanal");
```

Para efetuar o cancelamento do registro em um canal de transmissão de dados, basta fazer:

```
network.unregisterChannel(canal);
```

A interface `ApplicationDataReceiver` é a responsável por notificar a aplicação do usuário sobre o recebimento de mensagens. Essa notificação inclui o canal de transmissão, o remetente e um descritor da

[40]Nos casos em que a retirada da publicação não é instantânea, esse tempo é configurável por cada um dos mapeamentos.

mensagem, além da mensagem propriamente dita. Não será recebida nenhuma mensagem por um canal não registrado que for diferente do canal padrão.

Para efetuar o envio de uma mensagem a um nó cuja referência é representada pela variável `no`, basta fazer:

```
Node no = UM_NO;
byte[] buffer = MENSAGEM;
InputStream is = MENSAGEM_LONGA;

ExceptionHandler exHandler = new ExceptionHandler() {
  public void handleException(Exception e) {
    //trata a exceção no caso de erros durante o envio
  }
};

//Conjunto 1 - mensagens curtas
no.sendData(rd, buffer, exHandler); //1
no.sendData(network.getDefaultChannel(), rd, buffer, exHandler); //2
no.sendData(canal, rd, buffer, exHandler); //3

//Conjunto 2 - mensagens longas
no.sendData(rd, is, exHandler); //4
no.sendData(network.getDefaultChannel(), rd, is, exHandler); //5
no.sendData(canal, rd, is, exHandler); //6
```

O exemplo de código acima ilustra as diferentes maneiras de se enviar uma mensagem. O primeiro conjunto de linhas mostra como o envio de mensagens curtas pode ser feito. A linha 1 envia uma mensagem contida em `buffer` com descritor `rd` ao nó `no` pelo canal padrão, já que nenhum canal foi especificado. A linha 2 é equivalente à linha 1. A linha 3 envia a mesma mensagem enviada pelas linhas 1 e 2, entretanto, utiliza um canal diferente do canal padrão para fazer a transmissão. O nó destinatário deve ter se registrado

nesse mesmo canal para poder receber essa mensagem. O segundo conjunto demonstra essas mesmas características, mas agora utilizando, ao invés de um buffer curto, um `InputStream` de tamanho arbitrário. A linha 4 envia os dados contidos em `is` descritos por `rd` ao nó `no` pelo canal padrão. A linha 5 é equivalente a linha 4, e a linha 6 envia o mesmo conjunto de dados utilizando o canal `canal`. Todas as versões de envio de dados recebem um objeto `ExceptionHandler` para serem notificados de algum erro durante o envio da mensagem.

3.4.2.3 Comunicação em grupo O Hermes fornece um serviço de comunicação em grupo baseado em assinaturas. Apenas assinantes podem mandar mensagens a outros participantes do grupo. Para efetuar a assinatura de um grupo, basta fazer:

```
ExceptionHandler exHandler = new ExceptionHandler() {
  public void handleException(Exception e) {
    //trata a exceção no caso de erros durante a assinatura
  }
};
Group group = network.subscribe("NomeDoGrupo", exHandler);
```

O resultado do método requisitando uma assinatura de um grupo é um objeto do tipo `Group` que pode ser utilizado para enviar mensagens aos assinantes daquele grupo. Para enviar mensagens, deve-se fazer:

```
byte[] buffer = MENSAGEM;
group.sendData(rd, buffer, exHandler);
```

A chamada acima envia a mensagem contida em `buffer` descrita por `rd` a todos os assinantes do grupo `group`.

Através da interface `ApplicationDataReceiver`, a aplicação do usuário é notificada do recebimento de mensagens de grupos que ela assina. Na notificação estão contidos o grupo pelo qual a mensagem foi recebida, o nó remetente da mensagem, um descritor do conteúdo da mensagem e a mensagem propriamente dita.

Para cancelar a assinatura de um grupo, a aplicação do usuário pode fazer:

```
network.unsubscribe(group);
```

3.4.2.4 Armazenamento distribuído Para utilizar o serviço de armazenamento distribuído do Hermes precisa-se, primeiramente, gerar um identificador para o dado que se quer armazenar. O identificador pode ser gerado a partir de um texto ou simplesmente através de um vetor de bytes, utilizando-se a interface `IDFactory`. De posse do identificador e dos dados que se deseja armazenar, basta fazer a chamada do método `store` da interface `Network`. O exemplo abaixo demonstra como isso pode ser feito.

```
IDFactory fabrica = Hermes.getIDFactory();
ID identificador = fabrica.createID("FotoDoMeuCachorro");
FileInputStream file = new FileInputStream("FotoDoMeuCachorro.jpg");
ExceptionHandler exHandler = new ExceptionHandler() {
  public void handleException(Exception e) {
    //trata a exceção no caso de erros durante o armazenamento
  }
};
network.store(identificador, file, exHandler);
```

A obtenção de um dado armazenado na rede é igualmente fácil, o exemplo a seguir demonstra isso.

```
network.get(identificador, new Continuation<InputStream>() {
  public void handleResult(InputStream is) {
    try {
      FileOutputStream fos =
        new FileOutputStream("FotoRecebida.jpg");
      byte[] buffer = new byte[4096];
      while (is.available() > 0) {
        int read = is.read(buffer);
```

```
            fos.write(buffer, 0, read);
          }
          fos.close();
        } catch (Exception e) {
          handleException(e);
        }
      }
      public void handleException(Exception e) {
        //trata quaisquer exceções ocorridas durante o recebimento
      }
    });
```

No código acima, o dado armazenado na rede identificado por `identificador` é descarregado da rede e salvo em um arquivo com nome `FotoRecebida.jpg`.

3.4.3 Finalização

Muitos dos arcabouços disponíveis fornecem aos seus usuários alguma interface para a finalização e desconexão. Essas interfaces são expostas pelo Hermes através do método `disconnect` da interface `Network`. A aplicação deve fazer.

```
network.disconnect();
```

antes de finalizar a execução.

3.4.4 Aplicações de exemplo

Para validar a API disponibilizada pelo Hermes e demonstrar sua utilização, foram criadas uma aplicação de IM, o HermesMessenger, e uma versão P2P de um jogo multi-jogadores, o TIWarriors.

3.4.4.1 HermesMessenger O HermesMessenger é uma aplicação de IM que demonstra a utilização do Hermes e as suas diversas APIs. Ele permite que os usuários conectados façam troca de mensagens, troca de arquivos, busca aproximada pelos nomes dos arquivos dos outros usuários e tenham conversas em salas de bate-papo. Além disso, permite que os diversos usuários façam o armazenamento de seus dados na rede de forma que, quando se conectarem novamente, tenham acesso a eles. A Figura 12 é um *screenshot* que contém as telas mais comuns do HermesMessenger.

A localização dos usuários na rede é feita utilizando-se os serviços de busca exata fornecidos pelos Hermes. Cada um dos usuários conectados publica na rede o seu nome de usuário utilizando o método `publish`, da interface `Network`, que é pesquisado pelos demais utilizadores utilizando o método `search` da mesma interface. Um processo semelhante ocorre para a busca de arquivos dos usuários na rede. Durante a inicialização da aplicação, os arquivos contidos no diretório compartilhado são varridos e seus nomes publicados na rede de forma aproximada. Assim, posteriormente, os outros usuários poderão buscar esses arquivos utilizando buscas aproximadas e obtê-los dos outros nós, caso assim desejem.

As trocas de mensagens entre os usuários são feitas pela API de envio de mensagens curtas (método `sendData` com parâmetro `byte[]`, da interface `Node`). Durante as trocas de mensagens entre os usuários, uma foto de exibição do usuário é mostrada na janela de conversa, caso tal foto esteja disponível. A disponibilidade da foto é verificada utilizando o método `hasResource`, e a sua obtenção é feita através do método `getDataForResource`, ambos da interface `Node`. O envio de arquivos é feito utilizando a API de troca de mensagens longas (método `sendData` com parâmetro `InputStream`, da interface `Node`).

Um usuário pode, também, criar ou se juntar a uma sala de bate-bapo. Para isso, basta que digite o nome da sala desejada. Se a sala já existir, ele se juntará a ela, se ela ainda não existir uma nova sala será criada. Isso é feito através da API de comunicação em grupo do Hermes. Em suma, para cada sala um grupo de comunicação é criado.

A lista dos usuários conhecidos de cada um dos utilizadores dessa aplicação fica armazenada na rede, de forma que o usuário, independentemente do nó onde estiver conectado, terá acesso a essa lista. O usuário também tem a possibilidade de salvar algum arquivo na rede para seu uso posterior.

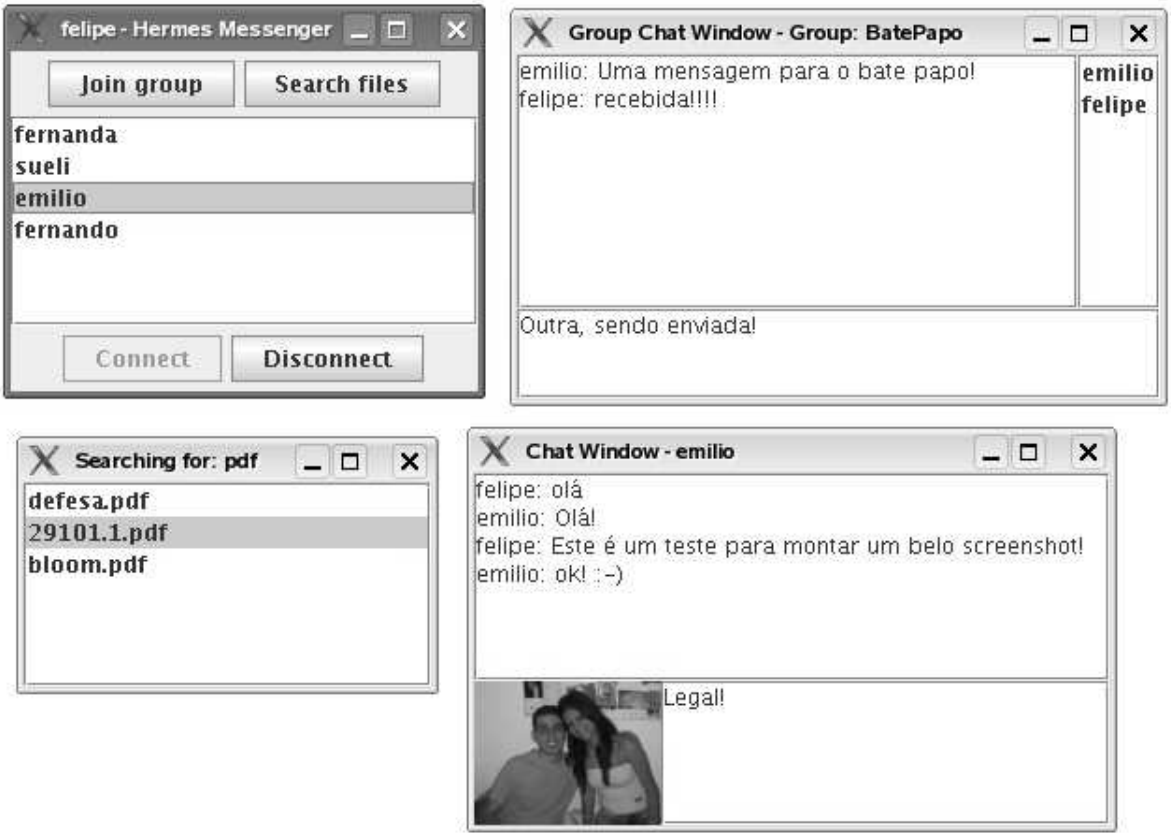

Figura 12: *Screenshot* do HermesMessenger. No canto superior esquerdo a tela que lista os contatos que estão *online*. Nesta tela também é possível encontrar os controles de conexão, desconexão, busca por arquivos e assinaturas de salas de bate-papo. No canto superior direito uma tela de um grupo de bate-papo, onde é possível ver os usuários presentes neste grupo, as mensagens já trocadas e enviar novas mensagens. No canto inferior esquerdo uma tela de resultado de uma busca por arquivos cujo nome contenham "pdf" e no canto inferior direito uma tela de troca de mensagens entre usuários.

Por possuir a maioria das funcionalidades presentes nos IMs disponíveis no mercado hoje, acreditamos que o HermesMessenger seja uma aplicação que demonstra a viabilidade da utilização da API do Hermes para a criação de aplicações P2P desse tipo.

3.4.4.2 TIWarriors O TIWarriors é um jogo multi-jogadores em rede, originalmente desenvolvido por Tiago Motta Jorge e licenciado sob a GPL. O jogo original pode ser obtido pela Internet[41]. A Figura 13 mostra um *screenshot* da aplicação.

A implementação original utiliza soquetes TCP/IP puros para efetuar a comunicação entre os nós participantes do jogo. O esquema de funcionamento é centralizado em um nó, o criador do jogo. Esse nó é responsável pelo recebimento de cada uma das ações de cada um dos jogadores, pelo cálculo do efeito de cada uma dessas ações e pelo posterior envio, a todos os participantes, do estado atualizado completo do jogo. Estes, então, atualizam suas telas.

O TIWarriors foi alterado para que passasse a utilizar, no lugar de soquetes TCP/IP, as interfaces dispostas pelo Hermes para a comunicação entre nós. Na implementação original, o nó criador do jogo se tornava o servidor, e os demais participantes deveriam conhecer o endereço IP do servidor para se juntar a um jogo já criado. Nesta nova versão, qualquer nó que deseje participar de um jogo multi-jogadores se conecta à rede P2P através das interfaces do Hermes. Depois de conectado, o nó faz uma pesquisa na rede pelos jogos já criados. O usuário, então, tem a opção de se juntar a um desses jogos ou criar o seu próprio jogo. A partir desse ponto, o esquema de funcionamento passa a ser igual ao esquema da implementação original, com a única diferença de utilizar as interfaces do Hermes para a troca de mensagens no lugar de soquetes TCP/IP.

As alterações feitas no TIWarriors foram pontuais. O desempenho do jogo não teve alterações perceptíveis após a substituição do mecanismo de troca de mensagens. Isso mostra que, neste caso, a utilização do Hermes é viável e traz ganhos como, por exemplo, a ausência da necessidade de se conhecer o endereço IP do servidor para se juntar a um jogo pré-existente[42]. Uma alteração mais abrangente do que a feita poderia

[41] `http://sourceforge.net/projects/tiwarriors/`

[42] Poder-se-ia argumentar que o mesmo poderia ser feito através da utilização de IP multicast. Para isto, bastaria fazer uma pequena alteração na implementação original. Contudo, o problema dessa solução é que, dependendo da rede utilizada pelos participantes (*e.g.* Internet), o uso de IP multicast não é viável. A utilização do Hermes supre esta deficiência.

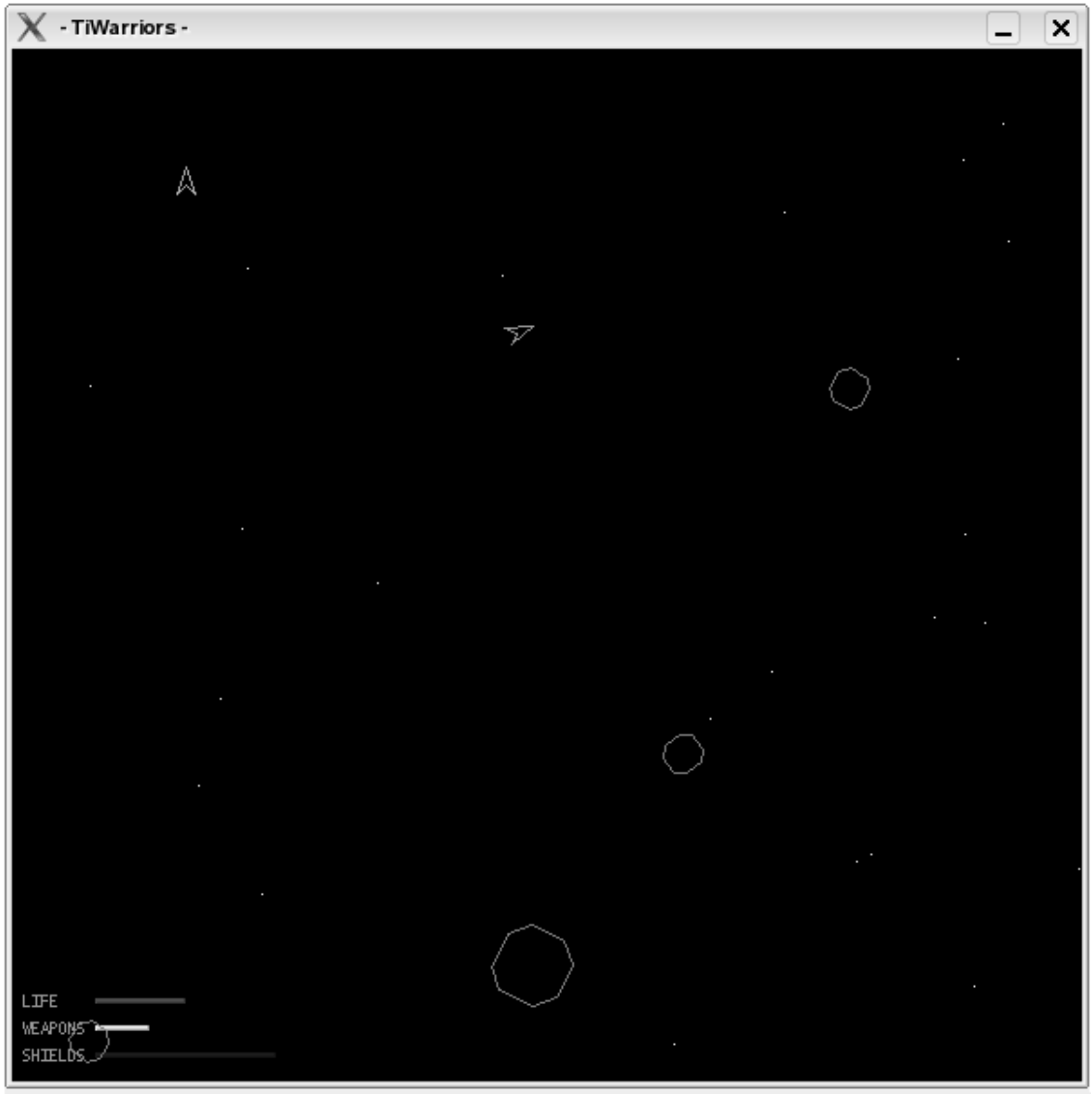

Figura 13: *Screenshot* do TIWarriors. Na figura, um jogo entre dois jogadores.

utilizar o mecanismo de troca de mensagens em grupo, oferecido pelo Hermes, para garantir a disseminação do estado atual do jogo a todos os participantes de forma otimizada. Isso, possivelmente, traria ganhos de escalabilidade e desempenho.

3.5 Completude da API do Hermes

O Hermes fornece serviços de troca de mensagens curtas e longas entre os nós da rede. A maior parte dos arcabouços apenas fornece um serviço de envio de mensagens curtas.

Aplicações distribuídas não raramente necessitam de comunicação em grupo (*multicast*). Dependendo do tipo da aplicação sendo construída, este tipo de serviço pode servir aos mais diversos propósitos. Ele pode ser usado para algo simples, como a distribuição de eventos para um grupo de interessados, ou para algo mais complexo, como a transmissão otimizada de dados a um grupo de nós. Embora seja possível implementar a comunicação em grupo usando apenas o envio de mensagens diretas entre dois nós, isso não é uma tarefa trivial. Para simplificar a implementação da aplicação do usuário, uma API com este tipo de serviço é fornecida pelo Hermes. Arcabouços como o JXTA[43] e o FreePastry (através do Scribe [16]) oferecem serviços deste tipo. O Hermes mapeia estes serviços de modo a esconder suas peculiaridades.

Uma aplicação distribuída normalmente não tem um só tipo de mensagem. Cada tipo de mensagem tipicamente requer um tipo diferente de tratamento. Fazendo uma analogia com TCP/IP, cada serviço abre sua própria porta no nó da rede onde está executando, a fim de tratar cada tipo de requisição de forma diferente. Aplicações P2P têm necessidades semelhantes. Assim, o conceito de porta, aqui denominado canal, se faz útil a essas aplicações. Por exemplo, um jogo multi-jogadores poderia criar um canal para a transmissão de cada uma das ações tomadas por cada um dos jogadores e outro para mensagens de texto ou voz. Isso evitaria que, no momento do recebimento de cada uma das mensagens, a aplicação tivesse que fazer um tratamento das mensagens antes de efetivamente processá-las. A criação de canais traz outras vantagens além da simples conveniência para o implementador da aplicação. Com este conceito, torna-se possível a diferenciação entre os canais no que diz respeito à sua confiabilidade, além da atribuição de

[43] `http://www.jxta.org`

prioridades[44] e parâmetros de qualidade de serviço (QoS) diferentes a cada um deles.

A busca por objetos na rede é uma das operações básicas de qualquer aplicação P2P. De forma geral, as buscas podem ser classificadas em dois tipos: buscas exatas e buscas aproximadas. Certas redes não-estruturadas, como Freenet, e as redes estruturadas em geral, oferecem apenas a busca exata por chaves. Nelas não é possível (de forma direta) fazer uma busca por uma palavra-chave contendo caracteres coringas (*wildcards*), expressões regulares ou expressões booleanas, como é de praxe na rede Gnutella, por exemplo. Este problema típico das redes estruturadas é bem conhecido e diversas soluções têm sido propostas. Propostas de esquemas híbridos, como [44], e esquemas estruturados puros com utilização de filtros de Bloom, como [57, 68], são exemplos de que ainda existe bastante campo para avanços neste sentido. Seguindo a linha de pensamento proposta até agora, o desenvolvedor de uma aplicação não deve se preocupar com as últimas novidades sobre busca aproximada de objetos em redes estruturadas (ou em qualquer outro tipo de rede). A idéia é que o Hermes cuide de tais problemas e que dessa forma deixe tempo livre para o desenvolvedor da aplicação se preocupar com coisas que para ele são mais importantes, como os últimos avanços na área da sua aplicação. Quando existir alguma solução mais apropriada disponível ele simplesmente trocará o arquivo de configuração da aplicação para que esta passe a utilizar o mapeamento do Hermes para a nova solução. O Hermes fornece uma API de busca exata e de busca aproximada que é mapeada para os arcabouços de forma transparente ao desenvolvedor da aplicação. Mesmo que o arcabouço empregado não ofereça busca aproximada, tal serviço estará disponível para os usuários do Hermes.

O Hermes fornece um serviço de armazenamento distribuído. Enquanto isso é algo quase intrínseco às implementações de rede estruturadas, não ocorre o mesmo para as redes não-estruturadas. JXTA por exemplo, apesar da sua rica API, não oferece nenhum serviço desse tipo. Para boa parte das aplicações P2P é essencial dispor de alguma maneira de armazenar dados na rede. Nós entram e saem da rede, e as informações relativas ao usuário, por exemplo, em muitos casos não podem ficar restritas ao armazenamento local de algum nó. Tais informações devem ser distribuídas de modo a não restringir o usuário a utilizar

[44] Alguns arcabouços, como o Bamboo, oferecem, ainda que não de forma direta e simples, suporte a este tipo de funcionalidade. No Bamboo, toda aplicação é composta de estágios. Cada estágio registra o tipo de mensagem que quer tratar e como esse tratamento deve ser feito. No caso do Bamboo, cada mensagem, e não o seu tipo, possui uma prioridade.

sempre os mesmos nós para poder ter acesso às suas informações.

A autenticação de usuários, a segurança das mensagens trocadas e a segurança dos dados armazenados na rede são inexistentes no Hermes. Dos arcabouços de redes P2P mapeados, apenas o JXTA oferece algum suporte para autenticação de usuários e envio de mensagens de forma segura. De forma mais clara, faltam maneiras intrínsecas ao Hermes para autenticar usuários, autenticar a origem e o destino de mensagens sigilosas (já que muitas das mensagens serão repassadas - *relayed* - pelos demais nós participantes da rede), garantir a integridade das mensagens, enfim, para lidar com os problemas básicos de segurança de dados em um ambiente de rede.

Duas aplicações foram criadas como provas de conceito e, para esses exemplos, a API existente supriu todas as necessidades. O HermesMessenger, entretanto, para garantir que apenas o destinatário lerá a mensagem, utiliza meios próprios, externos à rede, tanto com o objetivo de assegurar a autenticidade de origem e destino, como para cifrar a mensagem a ser enviada. Tais serviços claramente poderiam ser oferecidos pelo Hermes, mas que no entanto não são fornecidos por infra-estruturas P2P como o Bamboo ou o FreePastry.

A API oferecida pelo Hermes é mais completa do que a maioria das APIs oferecidas pelas redes de sobreposição mais comuns. Claramente, a segurança é um ponto para trabalhos futuros. Contudo, segurança não aparenta ser um serviço indispensável às aplicações P2P, dada a ausência deste serviço em boa parte das infra-estruturas disponíveis hoje. Se imaginarmos que as APIs das redes de infra-estrutura mapeadas são completas o suficiente para a implementação de uma aplicação P2P real, o Hermes também é, já que fornece todos os serviços oferecidos por tais infra-estruturas. Essa suposição é bem razoável, dada a complexidade das aplicações já construídas [32, 36, 53, 58] sobre essas infra-estruturas.

4 Mapeamento das redes de sobreposição para o Hermes

O mapeamento de uma nova implementação de rede de sobreposição para as APIs do Hermes consiste na criação de classes que implementem as seguintes interfaces: `ID`, `IDFactory`, `Node`, `Group`, `Network` e `NetworkFactory`.

O uso de um objeto `ID` e da sua respectiva fábrica se faz necessário pois o espaço de `ID`s (além dos algoritmos utilizados para a sua construção) de cada um dos arcabouços é diferente. Um mapeamento, através da API do Hermes, permite à aplicação ficar independente do arcabouço utilizado durante as operações de geração de `ID`s. Esse mapeamento é geralmente direto, pois a maior parte dos arcabouços já dispõem de classes para a construção desse tipo de objeto.

A criação de implementações para a interface `Network` já não é tão direta. Apesar da implementação da interface `NetworkFactory` ser relativamente simples, a implementação de uma classe que represente a rede é mais trabalhosa. Nessa classe devem estar contidas as regras de inicialização, execução e finalização para o arcabouço sendo mapeado. Algumas de suas operações, como `getLocalNode` e `search`, normalmente, são de implementação trivial, enquanto outras, como `publish` e `unpublish`, não raramente demandam esforço do implementador para manter a semântica definida pela API do Hermes. Um exemplo claro disto é a implementação do `unpublish`. Alguns dos arcabouços disponíveis, incluindo o Bamboo e o JXTA, não oferecem uma interface para a retirada de algo que foi publicado[45]. Esse tratamento tem que ser feito totalmente pelo mapeamento.

A implementação da interface `Node` merece mais atenção. Ela é, na verdade, um dos pontos centrais da API – juntamente com a interface `Network`. Naquela interface estão contidos os métodos que possibilitam a troca de mensagens entre os nós da rede. Muitos arcabouços possuem limites para o tamanho das mensagens enviadas e é papel da implementação do `Node` fazer o tratamento da fragmentação dos dados para o envio de forma transparente à aplicação. Também faz parte de das tarefas do `Node` garantir a entrega, a

[45] Apesar de tanto o Bamboo quanto o JXTA não possuirem interfaces explícitas para a retirada de um par chave-valor publicado em suas redes, ambos têm outros mecanismos para lidar com este problema. Essencialmente, para cada publicação é especificado o espaço de tempo pelo qual ela é válida. Através da utilização controlada deste mecanismo, é possível implementar a operação de retirada de publicação simplesmente fazendo com que esta não seja renovada automaticamente.

ordem e consistência dos dados enviados.

A comunicação em grupo de cada um dos arcabouços funciona de maneira diferente. Alguns exigem que um nó efetue a criação do grupo antes de assiná-lo, enquanto em outros a simples assinatura implica na criação do grupo. É parte das responsabilidades do implementador de um mapeamento para o Hermes esconder esse tipo de peculiaridade do implementador da aplicação através da criação de uma implementação da interface `Group`. Aqui devem ficar encapsulados também todos os trâmites necessários para se efetuar o envio de uma mensagem para os assinantes de um grupo.

Freqüentemente, a criação de classes auxiliares se faz necessária para que a implementação do mapeamento seja completa. Para facilitar o trabalho do implementador de um novo módulo de mapeamento, o Hermes fornece uma biblioteca que traz algumas destas classes auxiliares, evitando assim que ele gaste seu tempo em tarefas já resolvidas. Esta biblioteca está descrita na seção 4.1.

Em alguns dos mapeamentos feitos, algumas estruturas de dados, algoritmos e conceitos pouco comuns (descritos na seção 2.3) foram utilizados. Com o uso destes, detalharemos os mapeamentos de cada uma das infra-estruturas de rede nas seções 4.2 a 4.5, para então, na seção 4.6, detalharmos como o mapeamento pode ser feito para as demais redes.

4.1 Biblioteca de classes auxiliares

Diversas tarefas que fazem parte da criação de um mapeamento são recorrentes. Para, então, facilitar o trabalho do escritor de um mapeamento, o Hermes fornece uma biblioteca de classes auxiliares que são capazes de resolver algumas das tarefas mais comuns com que esse escritor se deparará.

Nesta biblioteca estão disponíveis classes para o auxílio à fragmentação e envio de mensagens longas, para o empacotamento e seriação de objetos, para o cálculo e verificação de filtros de Bloom, para a segmentação e consolidação de dados utilizando árvores de Merkle e classes para o auxílio à implementação de buscas aproximadas. Abaixo segue uma breve descrição de cada uma dessas funcionalidades oferecidas pelo Hermes:

Fragmentação e envio de mensagens longas Grande parte das infra-estruturas de rede não oferece

um serviço de envio de mensagens longas. Os usuários que utilizam essas infra-estruturas diretamente precisam fazer o controle da fragmentação, e posterior consolidação dos fragmentos, sem o auxílio da infra-estrutura de rede. A API do Hermes disponibiliza uma interface que se propõe a fazer justamente este serviço. Assim, o Hermes retira a responsabilidade da fragmentação e consolidação dos dados do escritor da aplicação e a atribui ao escritor do mapeamento. A biblioteca oferece, ao escritor do mapemento, classes que cuidam da fragmentação dos dados, do envio dos fragmentos e da consolidação dos fragmentos no destinatário. Essas classes necessitam apenas de duas operações primitivas, uma para o envio de mensagens curtas e outra para o recebimento destas mensagens. Durante a transferência, em segundo plano, essas classes cuidam da garantia de entrega, garantia de ordem e também controlam um *buffer* (preenchido automaticamente) de envio e de recebimento de forma a acelerar a escrita e leitura dos dados pelos nós envolvidos.

Empacotamento e seriação de objetos Durante a comunicação entre os nós em uma aplicação P2P é muito comum que sejam enviados dados de uma forma estruturada. A forma mais natural é através do envio de objetos de transferência de dados (*data objects*). Para fazer o empacotamento e seriação desses objetos, o Hermes oferece classes com operações simplificadas que auxiliam o empacotamento e a seriação.

Filtros de Bloom A utilização de filtros de Bloom (vide seção 2.3.1) é muito comum durante a implementação de um mapeamento. Eles são utilizados para fazer a filtragem de um conjunto de dados sem que para isso seja necessário trafegar esse conjunto todo pela rede. O Hermes fornece classes que tratam da criação de filtros de Bloom de tamanho arbitrário, assim como operações para efetuar testes de pertinência de um elemento a um conjunto (representado por um filtro de Bloom) e para avaliar se um conjunto está contido em outro (com ambos os conjuntos representados por filtros de Bloom).

Segmentação e consolidação de dados com árvores de Merkle Através da utilização de árvores de Merkle (vide seção 2.3.4) o Hermes oferece classes que auxiliam a segmentação e consolidação

de dados de tamanho arbitrário. Essas classes auxiliares são utilizadas pelos mapeamentos para fazer o armazenamento distribuído de dados. Para isto essas classes auxiliares precisam de algumas primitivas, são elas: cálculo do *hash* de um vetor de bytes, armazenamento de um segmento endereçado por um identificador (esse identificador é gerado pela função de *hash* fornecida) e obtenção de um segmento previamente armazenado dado um identificador.

Buscas aproximadas O Hermes fornece classes que implementam o algoritmo de busca aproximada descrito na seção 2.3.3. Esse algoritmo se baseia em n-gramas (vide seção 2.3.2) e busca exata de pares chave-valor. Os escritores dos mapeamentos podem utilizar essas classes auxiliares para implementar a busca aproximada caso a infra-estrutura de rede sendo mapeada não ofereça este serviço.

4.2 SimpleFlood

O SimpleFlood é uma implementação minimalista de uma rede de sobreposição não-estruturada. A implementação conta apenas com funcionalidades para a troca de mensagens curtas, publicação e busca exata (baseada em inundação) e comunicação básica em grupo (baseada em IP Multicast). Ela não garante a entrega de mensagens nem a alcançabilidade dos dados. O SimpleFlood é uma rede de sobreposição adequada apenas para testar aplicações feitas utilizando-se o Hermes sem a sobrecarga (necessária para uma aplicação real) das demais implementações.

O mapeamento do serviço de troca de mensagens entre os nós participantes da rede é direto. O SimpleFlood conta com uma abstração de um nó da rede que é bem semelhante ao conceito de nó existente na API do Hermes. A troca de mensagens maiores que o tamanho limite de mensagens do SimpleFlood, que exigem a fragmentação dos dados, é controlada pelo mapeamento[46]. O mapeamento utiliza-se do artifício de enviar junto com cada uma das mensagens um marcador indicativo do seu tipo. Os tipos das mensagens podem ser: mensagem curta, fragmento de mensagem longa ou mensagem de controle (como mensagens de ping e outras requisições entre nós que não sejam trocas de mensagens). Desta maneira, utilizando apenas

[46] A biblioteca de classes auxiliares ao mapeamento do Hermes fornece um serviço de fragmentação de mensagens. Para utilizá-lo basta que os mapeamentos criem algumas classes de ligação entre o Hermes e a rede de sobreposição utilizada. Os mapeamentos do FreePastry, JXTA e Bamboo também utilizam esse serviço.

uma interface simples de troca de mensagens, como o SimpleFlood oferece, o Hermes é capaz de tratar cada uma das mensagens de maneira distinta. Artifício semelhante é utilizado para o tratamento de canais.

A busca do SimpleFlood é baseada em identificadores, e não existe uma opção de busca aproximada de strings. A implementação da publicação e busca exata é feita de maneira direta, utilizando-se as interfaces do SimpleFlood. A publicação e a busca aproximadas foram implementadas utilizando-se as técnicas descritas na seção 2.3.3.

O SimpleFlood fornece um serviço de comunicação em grupos simples, baseado em assinaturas, tal qual o oferecido pela API do Hermes. O mapeamento desse serviço é direto.

O SimpleFlood não fornece um serviço de armazenamento distribuído. O mapeamento tem que fazer esse trabalho por si só. Quando um usuário requisita o armazenamento de algum dado na rede, esse dado é salvo localmente e uma publicação é feita. Qualquer nó que requisitar esse dado faz uma busca na rede e, caso encontre a publicação, obtém uma cópia do dado desejado. Caso o nó que fez a busca seja um nó diferente do nó que fez o primeiro armazenamento, este nó também armazena uma cópia desse dado localmente e após isso o publica na rede. Essa maneira de trabalhar faz com que dados muito requisitados sejam automaticamente distribuídos pela rede e distribuam a carga automaticamente. Entretanto, no caso de dados não muito requisitados, essa técnica de replicação passiva tem algumas limitações. Se o dado publicado for uma informação pessoal, por exemplo, apenas o usuário do nó que armazenou a informação vai acessá-la e, portanto, caso ele o faça de outro nó enquanto o nó onde foi feita a publicação estiver desconectado, o dado não será encontrado. Alguns esquemas de replicação e *caching* ativos para redes não-estruturadas [19, 46] foram propostos como soluções para este problema.

4.3 JXTA

Assim como o SimpleFlood, o JXTA é uma implementação de rede de sobreposição não-estruturada. Entretanto, diferentemente do SimpleFlood, ela não é minimalista. Com o objetivo de oferecer uma API poderosa e completa, os criadores do JXTA acabaram produzindo uma rede de sobreposição cuja utilização não é muito simples.

O JXTA não possui uma abstração que represente um nó participante da rede. Nesse sistema cada um dos nós da rede pode ter um ou mais *pipes* associados a ele. A comunicação de um nó a outro se dá através de *pipes* que têm versões para a leitura ou escrita de dados. Existe um tipo especial de Pipe chamado `PropagateType`. Neste tipo de *pipe* é permitida que a leitura seja realizada por mais de um utilizador ao mesmo tempo. A criação de *pipes* é feita pelo serviço `PipeService`.

Qualquer recurso que se deseje utilizar em uma rede JXTA deve ser publicado para que seja localizado pelos demais nós. Toda publicação (*Advertisement*) no JXTA pode ser representada em XML. A publicação dos recursos na rede JXTA é feita pelo `DiscoveryService`. O serviço de busca do JXTA já provê suporte para busca aproximada de strings.

O mapeamento do JXTA para o Hermes foi feito de uma maneira simples e direta. O conceito de um nó no Hermes representa uma entidade de rede com a qual pode-se interagir através do envio de mensagens. Esse é justamente o mesmo conceito de um *pipe* no JXTA. Logo, o mapeamento da interface `Node` do Hermes para o JXTA é simplesmente o encapsulamento de um *pipe*. Para obedecer a interface definida pelo Hermes, que faz com que um objeto do tipo `Node` seja `Serializable`, o mapeamento, na verdade, guarda a representação em XML da publicação do *pipe* ao invés do *pipe* propriamente dito. Quando um nó utiliza uma referência a um objeto do tipo `Node` pela primeira vez, o Hermes usa o `DiscoveryService` para fazer uma busca pelo *pipe* descrito pelo XML contido no `Node` e, em seguida, instancia um pipe concreto para permitir a comunicação entre os nós. O mapeamento de envio de mensagens curtas utiliza as interfaces do JXTA diretamente, enquanto o mapeamento para o envio de mensagens longas utiliza o serviço de fragmentação de mensagens fornecido pela biblioteca de classes auxiliares ao mapeamento do Hermes.

A implementação da publicação e busca, tanto exata quanto aproximada é direta. Para publicar um recurso é criado um `PipeAdvertisement` associado aos parâmetros recebidos para a publicação referenciando o *pipe* ligado ao nó publicador. Essa publicação é feita na rede através do `DiscoveryService` do JXTA. A busca é igualmente direta e feita através do mesmo serviço do JXTA, o `DiscoveryService`, que permite que sejam feitas buscas tanto exatas quanto aproximadas.

A comunicação em grupo no JXTA é realizada através de um pipe do tipo `PropagateType`. A

implementação da interface `Group` no Hermes é, na verdade, um encapsulamento deste *pipe*. Quando um nó deseja assinar um grupo, ele pesquisa por publicações de *pipes* na rede com o nome procurado. Caso encontre, ele se liga ao *pipe* vinculado a essa publicação. Caso contrário ele cria um *pipe* e o publica. Para evitar o particionamento das comunicações em grupo, caso mais de um nó assine (e acabe criando ao mesmo tempo) o mesmo grupo, cada um dos participantes na rede utiliza o `DiscoveryService` para buscar, a intervalos regulares de tempo, grupos com os mesmos nomes que ele assina. Caso encontre grupos diferentes dos grupos que ele possui atualmente, ele mantém apenas o grupo cujo criador possui o menor identificador.

Assim como o SimpleFlood, o JXTA não oferece um serviço de armazenamento distribuído. Existem, entretanto, diferentes projetos como o *Distributed Storage Leasing* (DSL)[47] e o *do*[48] que poderiam ser úteis na implementação do mapeamento do serviço de armazenamento distribuído para o JXTA. Nenhum desses serviços, porém, está maduro o suficiente para ser utilizado. Por esse motivo, o mapeamento para o JXTA do serviço de armazenamento distribuído foi feito de uma maneira bem parecida com o mapeamento feito para o SimpleFlood, tendo as mesmas vantagens e desvantagens deste. Quando se requisita o armazenamento de um dado, este é armazenado localmente e uma publicação dele é colocada na rede. Caso algum outro nó solicite a recuperação desse dado, o dado é salvo e publicado pelo nó solicitante.

4.4 FreePastry

FreePastry é uma implementação de uma rede de sobreposição estruturada que conta com serviços de roteamento de mensagens, notificação de eventos, busca e armazenamento distribuído.

O FreePastry disponibiliza a API proposta por [22] (vide a seção 5). Portanto, existe um objeto que representa um nó da rede (`NodeHandle`). O mapeamento da interface `Node` do Hermes, neste caso, é um simples encapsulamento deste objeto. O mapeamento para as operações de envio de mensagens utiliza-se do serviço de fragmentação oferecido pelo Hermes no caso do envio de mensagens longas.

A DHT oferecida pelo FreePastry não permite a utilização do mesmo identificador para duas publicações

[47] `http://sourceforge.net/projects/dsl`
[48] `http://do.jxta.org/`

distintas. Em outras palavras, a DHT FreePastry não possui um tratamento para colisões de chaves nas operações de inserção na sua tabela de *hash* distribuída. Uma chave deve apontar para um único valor. A API do Hermes, no entanto, permite que dois valores diferentes sejam publicados com a mesma chave. A alternativa, então, seria fazer um objeto lista que contém os objetos com a mesma chave e publicar a lista no lugar do objeto propriamente dito. Bastaria, a cada vez que se fizesse uma publicação, verificar se já existe um objeto com a mesma chave publicada, e caso existisse, adicionar o objeto na lista do objeto já publicado. Essa abordagem, entretanto, traz outras complicações. Publicações concorrentes devem tomar o cuidado de não sobrescrever publicações de outros nós. A remoção de objetos da lista sofreria problemas semelhantes. O *caching* e a replicação automática das publicações feitas na DHT também criam alguns obstáculos, uma vez que a busca, publicação e retirada de objetos da DHT podem alcançar nós diferentes da rede e, portanto, acabar acessando versões diferentes da lista.

A DHT do FreePastry não possui uma operação de remoção de publicações. Para evitar que a DHT comece a acumular publicações não mais necessárias, um processo de coleta de lixo é executado a períodos regulares (ou quando for necessário, por exemplo, quando o espaço para armazenamento de dados se esgotar), retirando todas as publicações cuja validade expirou. O período entre as coletas, entretanto, não é configurável, e é tipicamente da ordem de dias e não de minutos como na interface do Hermes. A publicação de algo na DHT, ainda que ajustada como válida apenas por alguns minutos, pode facilmente permanecer disponível por vários dias. Poder-se-ia contornar isso avaliando o momento da publicação de cada objeto antes de devolvê-lo como possível resultado de uma busca. Mas outros problemas, como sincronismo de relógios e desperdício de banda e de espaço, acabariam aparecendo.

Ao invés de tentar contornar esses problemas, propomos uma solução escalável mais simples, como alternativa ao uso da DHT do FreePastry. A proposta é utilizar o serviço de comunicação em grupo oferecido pelo FreePastry, o Scribe [16], para a criação de uma rede semântica de sobreposição[49] . O Scribe foi criado para lidar com grupos cuja composição é muito dinâmica e nos quais a entrada e saída de participantes são

[49]Uma rede semântica de sobreposição (*semantic overlay network* - SON) [21] é uma rede onde os nós com recursos ou interesses semelhantes são aglomerados (*clustered*) em conjuntos lógicos de conteúdos e interesses. Um nó pode participar de mais de um conjunto e tipicamente o faz.

muito frequentes. Ele é capaz de lidar com um número muito grande de grupos e de participantes, de forma eficiente e escalável. Assim como a DHT, o Scribe se baseia na topologia da rede para efetuar a manutenção dos grupos criados na rede.

Toda publicação exata através do Hermes é feita com um conjunto de identificadores. A idéia é que cada publicação implique na assinatura dos grupos de comunicação cujos identificadores sejam os mesmos da publicação. Assim, todos os nós que tiverem um recurso publicado com o mesmo identificador farão parte do mesmo grupo de comunicação.

As buscas exatas no Hermes são feitas com base em um conjunto de identificadores. Toda vez que, utilizando o mapeamento para FreePastry, um nó fizer uma busca, ele na verdade estará enviando uma mensagem para cada um dos grupos que tenham identificadores iguais aos identificadores recebidos como parâmetro para a busca. Cada mensagem de busca contém um filtro de Bloom montado a partir dos identificadores recebidos como parâmetro para a busca. Suponha que (a, b) sejam os identificadores a serem procurados. Neste caso serão enviadas duas mensagens, uma para o grupo com identificador a e outra para o grupo com identificador b. Apenas os nós que tiverem alguma publicação cujos identificadores contenham a ou b serão alcançados (inclusive nós que possuem publicações que não sejam resultados aceitáveis para a busca como, por exemplo, (a), (b) ou (a, c)). Assim como aconteceria se estivéssemos utilizando a DHT, a utilidade do filtro de Bloom se demonstra no momento da resposta ao remetente da busca. Antes de cada um dos nós responder à requisição de busca, ele avalia se alguma de suas publicações satisfaz o filtro de Bloom recebido e, só nessas circunstâncias, responde ao nó que originou a pesquisa. A utilização desse filtro faz com que apenas alguns poucos falsos positivos (cuja probabilidade pode ser parametrizada, conforme visto na seção 2.3.1) sejam devolvidos como resultado de uma busca. A utilização de um filtro de Bloom também diminui significativamente o tráfego adicional criado na rede pela utilização de um grupo de comunicação para a criação de uma rede semântica de sobreposição.

A utilização do esquema acima para a publicação e busca permite que a publicação e a remoção de publicações da rede sejam feitas de forma simples e eficiente. A implementação da busca aproximada foi feita utilizando-se o princípio acima mesclado com a técnica descrita na seção 2.3.3.

A implementação do serviço de comunicação em grupos é direta através da utilização do Scribe.

O Past é o sistema de armazenamento distribuído fornecido pelo FreePastry. Ele é na verdade a DHT do FreePastry e, portanto, oferece o serviço de mapeamento de chaves em valores. Apesar desta DHT estar preparada para lidar com dados relativamente grandes, ela assume que esses dados devem caber na memória. A API do Hermes não faz qualquer suposição a esse respeito. Além disso, a publicação de dados enormes é prejudicial ao bom desempenho de certos serviços da DHT, como por exemplo replicação e *caching*. Para manter a funcionalidade prevista pela API do Hermes e, ao mesmo, tempo utilizar o sistema de armazenamento distribuído fornecido pelo FreePastry sem comprometer o seu desempenho, o mapeamento do FreePastry para o Hermes faz uso das técnicas descritas na seção 2.3.4.

4.5 Bamboo

O Bamboo é uma implementação de rede de sobreposição estruturada que conta com serviços de roteamento de mensagens, armazenamento distribuído e comunicação em grupo bem como publicação e busca exata de pares chave-valor.

Todo nó participante da rede do Bamboo possui um identificador de 160 bits. O Bamboo oferece um serviço de encaminhamento de mensagens que entrega a mensagem para o nó cujo identificador seja o mais próximo (no espaço de identificadores) do identificador do destinatário da mensagem. Isso significa que uma mensagem nunca deixa de ser entregue (no caso degenerado o nó local recebe todas as mensagens que ele mesmo envia). Logo, o receptor de uma mensagem não é necessariamente igual ao destinatário, sendo diferente do nó destinatário caso este nó não esteja disponível na rede. Esse serviço de encaminhamento de mensagens, que utiliza a topologia da rede de sobreposição para efetuar o seu roteamento foi utilizado para implementar o serviço de trocas de mensagens no Hermes. Assim como no FreePastry, o tamanho da mensagem é limitado e, por essa razão, os serviços de fragmentação fornecidos pelo Hermes foram utilizados para efetuar o envio de mensagens longas.

A DHT do Bamboo, ao contrário daquela fornecida pelo FreePastry, está preparada para lidar com colisões de chaves durante a publicação. Ela permite que múltiplos valores sejam publicados com a mesma chave e,

portanto, as buscas nessa DHT podem devolver mais de um resultado. Além disso, todas as inserções feitas nessa DHT possuem um período de validade. A intervalos de tempos regulares (configurável e geralmente na ordem de alguns poucos minutos) todas as publicações da DHT cujo período de publicação tenha expirado são descartadas.

O serviço de publicação e busca exata do Hermes foi implementado através da utilização da DHT fornecida pelo Bamboo. A inserção na DHT é feita de forma trivial: uma inserção na DHT para cada um dos identificadores fornecidos através da interface do Hermes. Junto com os dados propriamente ditos, a inserção armazena na DHT um filtro de Bloom calculado a partir do conjunto dos identificadores fornecido como parâmetro para a publicação. A busca é feita da seguinte maneira: calcula-se um filtro de Bloom a partir do conjunto dos identificadores fornecidos como parâmetro para a busca. Para cada identificador desse conjunto é feita uma busca enviando-se o filtro de Bloom calculado no passo anterior. Cada um dos nós atingidos pela busca (que são os possíveis detentores de algum dado) avalia quais das publicações mantidas por ele possuem filtros de Bloom que contenham o filtro de Bloom recebido com a requisição de busca. Esses nós então enviam, ao nó que efetuou a busca, mensagens contendo os resultados. A retirada de publicações é feita através da utilização controlada do mecanismo de publicação com validade oferecido pelo Bamboo. Todas as publicações são feitas com um período de validade relativamente curto. Instantes antes da publicação ter o seu período vencido, uma publicação de renovação é efetuada automaticamente. Assim, quando através da interface do Hermes uma requisição de retirada de publicação for feita, apenas a renovação daquela publicação é cancelada, permitindo que o Bamboo descarte as cópias espalhadas pela rede automaticamente logo que estas expirarem.

O serviço de busca aproximada foi implementado sobre a DHT com uso da técnica descrita na seção 2.3.3.

O Bamboo fornece um serviço de comunicação em grupo chamado SoftScribe, que é muito semelhante ao serviço Scribe do FreePastry. Esse serviço de comunicação em grupo foi utilizado para a implementação do serviço de comunicação em grupo do Hermes.

O serviço de armazenamento distribuído oferecido pelo Hermes foi mapeado utilizando-se de um serviço

de armazenamento de grande quantidade de informações já existente no Bamboo. Para armazenar as informações na rede, o serviço do Bamboo usa a DHT em conjunto com árvores de Merkle. O mapeamento consistiu na criação de uma camada que faz a tradução das requisições feitas pela interface do Hermes para as interfaces do Bamboo.

4.6 Mapeamento para outras redes de sobreposição

Com os mapeamentos para SimpleFlood, JXTA, Bamboo e FreePastry, esperamos ter mostrado que a API do Hermes é razoavelmente simples de se mapear para cada uma dessas redes, que formam um conjunto representativo das diferentes infra-estruturas de rede de sobreposição disponíveis. Mostraremos, a seguir, que os únicos serviços necessários para fazer um mapeamento completo das interfaces do Hermes são a busca exata de pares chave-valor e o roteamento de mensagens entre os participantes da rede.

A implementação do serviço de envio de mensagens curtas é direto. O serviço de envio de mensagens longas, caso não esteja disponível na infra-estrutura de rede sendo mapeada, pode ser feito através da utilização do serviço de fragmentação de mensagens fornecido pelo Hermes. O serviço de fragmentação utiliza apenas o serviço de envio de mensagens curtas para fazer o seu trabalho.

A comunicação em grupo em redes não-estruturadas, caso não esteja disponível na infra-estrutura de rede sendo mapeada, pode ser feita através do uso do JGroups[50]. O JGroups permite a criação de grupos de comunicação entre participantes de uma rede. Para tanto ele requer apenas primitivas de envio de mensagens de um nó a outro. Os mapeamentos já disponíveis para o JGroups incluem UDP (IP Multicast), TCP e JMS[51]. A criação de um novo mapeamento é uma tarefa relativamente simples.

A comunicação em grupo em redes estruturadas, caso não esteja disponível, pode ser feita tanto com o uso do JGroups, como foi proposto acima para as redes não-estruturadas, como através de uma adaptação do Scribe. O esquema de comunicação em grupo do Scribe pode ser facilmente adaptado para outras redes estruturadas. No Bamboo, por exemplo, a sua implementação consiste apenas de sete classes.

A implementação da busca exata geralmente é direta: a maioria das redes de sobreposição fornece algum

[50] `http://www.jgroups.org/`
[51] Java Message Service - `http://java.sun.com/products/jms/`

mecanismo para a busca exata que pode ser mapeado para o Hermes. A exceção notável é o FreePastry que, apesar de dispor de interfaces de busca exata que seriam facilmente mapeáveis para o Hermes, não permite a publicação de mais de um valor com a mesma chave. No caso do mapeamento de sistemas de redes estruturadas com esse tipo de limitação, poder-se-ia criar uma rede semântica de sobreposição utilizando as mesmas idéias utilizadas pelo mapeamento do FreePastry ou, se o desempenho da SON se tornar um ponto crítico, a implementação da publicação poderia ser feita através de um objeto do tipo lista que contém todas as publicações que possuem uma determinada chave. Esta última solução, entretanto, tem diversos detalhes que precisam ser tratados, tais como versionamento e acesso exclusivo para a edição da lista. A solução eficiente destes problemas pode não ser tão simples de se obter em um sistema totalmente distribuído.

Caso a infra-estrutura de rede já não disponha de um serviço de busca aproximada, o mapeamento da busca aproximada pode ser feito aplicando-se a técnica descrita na seção 2.3.3 ao serviço de busca exata.

Nas redes não-estruturadas, o armazenamento distribuído pode ser feito da mesma maneira que no mapeamento do SimpleFlood e do JXTA, que não oferecem um serviço desse tipo. Nas redes estruturadas, a própria maneira pela qual a DHT é implementada acaba trazendo consigo um serviço de armazenamento distribuído. Caso haja uma limitação no tamanho dos valores armazenáveis na DHT, e essa limitação seja um problema, o uso de uma árvore de Merkle é uma solução.

5 Trabalhos relacionados

Frank Dabek et alii [22] propõem a criação de uma API comum a todos os sistemas P2P baseados em redes de sobreposição estruturadas. A sua proposta trata da criação de uma API comum tanto para o envio, roteamento e entrega de mensagens quanto para busca e comunicação em grupo (*multicast*). Para a definição desta API é utilizada uma linguagem de programação fictícia e facilmente mapeável a qualquer linguagem de programação real. Por ser dirigida exclusivamente às redes distribuídas estruturadas, esta API diferencia-se do Hermes no sentido de que ela deixa claro ao desenvolvedor da aplicação qual o tipo de rede utilizada. Por isso a troca por outro tipo de rede não pode ser feita de forma direta. Além disso, a utilização de uma linguagem fictícia para a especificação da API, sem a definição de regras de mapeamento para uma linguagem real (como aquelas existentes para a IDL de CORBA), impossibilita que dois arcabouços, ainda que na mesma linguagem de programação, forneçam uma API comum. Um exemplo claro é a implementação do FreePastry em Java. Apesar de ela disponibilizar a API definida em [22], ela contém classes cujo nome do pacote é `rice.pastry.p2p.commonapi`. Qualquer aplicação que escolher este arcabouço como sua solução se vê obrigada a fazer referências a estas classes. Isso cria um acoplamento forte entre a aplicação e o arcabouço escolhido e dificulta a troca do arcabouço depois da implementação da aplicação.

Giuseppe Ciaccio [18] propõe uma API mais flexível, genérica, simples e concisa do que [22]. A API proposta tenta levar em consideração os diferentes tipos de rede P2P disponíveis. Esse trabalho parece uma lapidação da proposta feita por Dabek et alii: Ciaccio retira e simplifica diversas características (como o tratamento do encaminhamento de pacotes) e adiciona outras (como o conceito de portas de comunicação). Esta proposta não especifica o tipo de rede. Nesse sentido, ela é bem próxima a API do Hermes. Ela se limita a especificar o envio, o roteamento e a entrega de mensagens. Ciaccio se utiliza de uma linguagem fictícia, similar à utilizada por [22], para definição da API. As regras para o mapeamento da linguagem para uma linguagem real não foram definidas e, portanto, essa proposta possui os mesmos problemas encontrados em [22].

Jörg Liebeherr et alli [42] propõem a criação de uma API para programação de aplicações sobre redes de sobreposição que é muito parecida com a API de soquetes de Berkeley. Ela não especifica o tipo de rede a

ser utilizado e permite que as trocas de arcabouços sejam feitas de maneira transparente para a aplicação. Esta proposta não se preocupa com busca, mas simplesmente com o envio e o recebimento de mensagens. Quando comparado ao Hermes e às duas propostas anteriores, esta proposta é de nível bem mais baixo, porém muito mais simples.

A plataforma JXTA, acrônimo para a palavra inglesa *juxtapose*, é uma especificação de um conjunto de protocolos para permitir a comunicação entre diferentes tipos de dispositivos através da criação de uma rede P2P de sobreposição. Existem implementações dessas especificações feitas para diferentes linguagens, tais como Java e C. Essas implementações são compatíveis entre si, ou seja, utilizam o mesmo protocolo para comunicação. O JXTA especifica, além dos serviços disponíveis, como as implementações devem ser feitas. A rede de sobreposição criada por JXTA é uma rede não-estruturada e as suas APIs são, apesar de completas, complexas. Dentre as infra-estruturas de rede para a programação de aplicações P2P, a plataforma JXTA é uma das mais conhecidas. Essa plataforma, entretanto, não tem uma API simplificada e é de difícil utilização. Isso incentivou a criação de diversos softwares satélites à plataforma JXTA, como por exemplo o myJXTA2[52], para auxiliar o desenvolvimento de aplicações. Embora essa plataforma não sofra dos mesmos problemas das soluções já citadas, o desenvolvedor que a utiliza se vê obrigado a amarrar fortemente sua aplicação à rede de sobreposição não-estruturada especificada pelo JXTA. Isso faz com que uma troca da arquitetura em um momento avançado do desenvolvimento seja muito trabalhosa.

O projeto cP2Pc [39] (lê-se "*copy to pc*") é uma abordagem que visa a criação de um *ÜberClient* conforme proposto por Brandon Wiley em [67]. Uma aplicação *ÜberClient* é uma aplicação que provê uma única interface para os usuários, entretanto se conecta nas diferentes redes simultaneamente de forma transparente. Para isso os projetistas do cP2Pc criaram uma API comum de programação para a criação de aplicações de compartilhamento de arquivos utilizando uma linguagem real de programação, C. Foram feitos mapeamentos para as redes Gnutella e GDN. O projeto cP2Pc, apesar de definir uma API comum e ter mapeamentos para diferentes tipos de rede, se diferencia do Hermes por ser uma API voltada apenas para a criação de aplicações de compartilhamento de arquivos. Dessa forma a sua API é um pouco limitada, não tendo, por

[52] `http://myjxta2.jxta.org/`

exemplo, suporte à comunicação em grupo ou à troca de mensagens entre nós participantes da rede. Outro ponto que o diferencia da nossa proposta é que, com o cP2Pc, a aplicação do usuário pode se conectar (e normalmente o faz) a mais de uma rede ao mesmo tempo, com o objetivo de permitir o compartilhamento de arquivos entre as diferentes redes. Isso mostra uma diferenciação bem clara entre as intenções dos criadores do cP2Pc e a intenção do Hermes, que é possibilitar a troca da infra-estrutura de rede utilizada sem alterações no código da aplicação.

O Hermes define a semântica da sua API em uma linguagem de programação real. Isso cria a possibilidade da troca de infra-estruturas de rede e da experimentação com as várias infra-estruturas disponíveis. O desenvolvedor fica livre para escolher, a qualquer momento, qual infra-estrutura é a mais adequada para o seu caso.

Fica claro que em [18,22,42] houve uma preocupação com o aprendizado de uma API comum pelo desenvolvedor. Com uma API comum, o desenvolvedor passa a ser capaz de implementar a sua aplicação sobre qualquer arcabouço que julgar apropriado sem ter que se preocupar com as idiossincrasias do arcabouço escolhido. Não fica claro, entretanto, se os autores desses trabalhos chegaram a definir os mapeamentos de suas linguagens fictícias para linguagens de programação reais. Esses trabalhos se propõem a resolver um problema em um nível mais baixo do que o Hermes. Eles são, na verdade, bem mais úteis para um desenvolvedor que está criando um mapeamento de uma infra-estrutura de rede para o Hermes do que para o desenvolvedor de uma aplicação final que, freqüentemente, não tem como avaliar *a priori* com que perfil de uso, tamanho da rede ou carga sua aplicação deverá lidar.

A plataforma JXTA e o cP2Pc se preocuparam com o mapeamento das suas APIs para uma linguagem de programação real, sendo que o JXTA vai além disso ao especificar, inclusive, como as implementações dos serviços devem funcionar.

Dos trabalhos relacionados, o cP2Pc é o que mais se aproxima da proposta do Hermes, que é ser um arcabouço para a implementação de aplicações P2P independentes da rede sobre a qual elas executarão. No entanto, a API do cP2Pc é limitada e não oferece diversos dos serviços disponíveis no Hermes, como a troca de mensagens entre os nós e a comunicação em grupo.

6 Considerações finais

6.1 Trabalhos futuros

Apesar das diferentes implementações de rede fornecerem, na sua maioria, apenas a funcionalidade de busca exata de chaves ou busca aproximada de cadeias de caracteres, por vezes as aplicações P2P podem requerer buscas mais elaboradas do que essas. Buscas na rede por faixas de valores, ou aproximadas através da utilização de vetores de características, são exemplos desse tipo de requisito. Enquanto tais buscas são facilmente implementáveis em redes não estruturadas, sua implementação em redes estruturadas é mais trabalhosa [10, 12, 43, 68]. Como a proposta do Hermes é facilitar o trabalho do implementador de uma aplicação P2P e isolá-lo da rede de sobreposição escolhida, o oferecimento de uma API para consultas avançadas se faz necessário.

As mensagens trocadas pelo Hermes não tem nenhum tipo de prioridade associada a elas. Com a adição de prioridades para o controle do envio de mensagens, tornar-se-ia possível o oferecimento de QoS. Isso possibilitaria a criação de canais contínuos de comunicação, como os que seriam necessários para a implementação de uma aplicação de telefonia, por exemplo.

O serviço de armazenamento distribuído do Hermes não é eficiente para a replicação de dados nas redes não estruturadas. O esquema passivo de replicação atualmente implementado não é apropriado para o armazenamento e posterior descarregamento de dados que são de baixa popularidade. Um esquema de replicação ativo para a melhoria da eficiência desse serviço está sendo implementado.

Apesar da construção de um sistema P2P totalmente seguro estar fora do escopo do Hermes, alguns serviços básicos de segurança seriam úteis se disponíveis. Um serviço de envio de mensagens assinadas e criptografadas pressupõe a existência de um serviço de autenticação de usuários. Ambos seriam serviços cuja importância se destacaria. Sem uma entidade certificadora, os nós precisariam confiar um nos outros para efetuarem tarefas de autenticação. A figura de um nó certificador poderia surgir na rede através de um esquema de controle de reputação, como o proposto por [33]. A possibilidade de implementar um serviço básico de segurança nesses moldes está sendo estudada.

O Hermes não impõe quase nenhuma sobrecarga à aplicação que o utiliza quando comparado à utilização da implementação de rede diretamente. Isso se deve ao fato do Hermes apenas traduzir a sua API para chamadas à API da rede de sobreposição sendo utilizada. Nos casos onde o Hermes utiliza os serviços básicos da rede de sobreposição para oferecer algum serviço que está disponível no Hermes mas não na implementação da rede propriamente dita, não há base para comparações de desempenho. Ainda assim, é necessário fazer um estudo aprofundado sobre as sobrecargas, que acreditamos ser desprezíveis, que o Hermes impõe às aplicações que o utilizam. Também é necessário avaliar o grau de escalabilidade e desempenho que os serviços do Hermes, que se utilizam dos serviços básicos das implementações de rede, podem alcançar.

6.2 Conclusão

O Hermes oferece uma API definida numa linguagem de programação real (Java) e faz da implementação da rede de sobreposição uma parte das aplicações P2P que é facilmente substituível. Para usar uma certa rede de sobreposição através do Hermes é necessário um módulo de mapeamento para essa rede. Lembrando a similaridade entre o Hermes e camadas de portabilidade como ODBC (ou JDBC), JNDI e JMS, vemos que a função do módulo de mapeamento é similar à de um *driver* ODBC (ou JDBC), ou à de um *JNDI provider*, ou ainda à de um *JMS provider*. A existência de módulos de mapeamento para quatro redes de sobreposição distintas, a saber, Bamboo, FreePastry, JXTA e SimpleFlood, é uma forte indicação de que API do Hermes é mapeável para todas as redes de sobreposição usuais, sejam elas estruturadas ou não estruturadas. Até onde pudemos averiguar, esta é a única proposta com essas características.

Algumas redes de sobreposição não fornecem todos os serviços que são oferecidos pela API do Hermes. Por isso, nos mapeamentos dessas redes é necessário um passo além da simples tradução de chamadas à API do Hermes em chamadas à API da rede de sobreposição escolhida. Esse passo adicional seria um trabalho extra com o qual o implementador de uma aplicação P2P teria que se preocupar, caso sua aplicação precisasse desse serviço e fosse utilizar diretamente a API da rede de sobreposição. O Hermes elimina esse problema. Por isolar o desenvolvedor de aplicações P2P da rede de sobreposição, o Hermes remove do

desenvolvedor a preocupação com certos serviços básicos e permite que ele se concentre na resolução dos problemas inerentes às aplicações.

A API do Hermes oferece todos os serviços das redes de sobreposição usuais, exceto funções de segurança (disponíveis apenas no JXTA), que devem tratadas externamente ao arcabouço. Sendo tão completa quanto as APIs das redes mapeadas, ela dá ao desenvolvedor de aplicações P2P tanto poder quanto a utilização direta das APIs dessas redes. O uso do Hermes, no entanto, tem a vantagem de evitar o acoplamento direto entre a aplicação P2P e a rede de sobreposição. Dessa forma, o Hermes dá ao criador de uma aplicação P2P a liberdade de adiar a decisão sobre qual a rede de sobreposição mais apropriada para a sua aplicação. Além de permitir que tal decisão seja tomada apenas no momento da implantação da aplicação, o Hermes permite que essa decisão seja revista com a aplicação já em produção, sem a necessidade de se alterar uma linha de código sequer.

Nos sistemas P2P atuais é relativamente comum a utilização de n-gramas, para a busca de padrões de cadeias de caracteres, de filtros de Bloom, para filtrar conjuntos de dados, e de árvores de Merkle, para a distribuição de um conjunto de dados. A implementação dos mapeamentos das redes de sobreposição exigiram a utilização dessas técnicas em conjunto e de forma distribuída, o que não é algo comum nos sistemas de hoje. Acreditamos ser essa uma das contribuições deste trabalho.

Referências

[1] Secure Hash Standards - Secure Hash Signature Standard (SHS) (FIPS PUB-180-2). http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf, AUG 2002.

[2] Reka Albert, Hawoong Jeong, and Albert-Laszlo Barabasi. Error and attack tolerance of complex networks. *Nature*, 406(6794):378–382, July 2000.

[3] Yariv Aridor and Mitsuru Oshima. Infrastructure for mobile agents: Requirements and design. In *MA '98: Proceedings of the Second International Workshop on Mobile Agents*, pages 38–49, London, UK, 1999. Springer-Verlag.

[4] Arno Bakker, E. Amade, Gerco Ballintijn, Ihor Kuz, P. Verkaik, I. van der Wijk, Maarten van Steen, and Andrew S. Tanenbaum. The Globe distribution network. In *Proceedings of the USENIX Annual Conference*, pages 141–152, 2002.

[5] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, February 2003.

[6] A. L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, October 1999.

[7] Albert-László Barabási. *Linked: The new science of networks*. Perseus Publishing, 2002.

[8] Albert-László Barabási and Eric Bonabeau. Scale-free networks. Scientific American, May 2003.

[9] Paul Baran. On distributed communications - i. introduction to distributed communications networks. Research Memoranda RM-3420-PR, RAND Corporation, 1964.

[10] Daniel Bauer, Paul Hurley, Roman Pletka, and Marcel Waldvogel. Bringing efficient advanced queries to distributed hash tables. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*, pages 6–14, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Albert Benschop. Peer-to-peer: Networks of unknown friends - the power of sharing. `http://www.sociosite.org/p2p_en.php`, March 2004.

[12] Bobby Bhattacharjee, Sudarshan Chawathe, Vijay Gopalakrishnan, Pete Keleher, and Bujor Silaghi. Efficient peer-to-peer searches using result-caching. In *The 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, February 2003.

[13] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[14] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[15] Miguel Castro, Manuel Costa, and Antony Rowstron. Peer-to-peer overlays: structured, unstructured, or both? Technical Report MSR-TR-2004-73, Microsoft Research, 2004.

[16] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), October 2002.

[17] G. Ciaccio. NEBLO: Anonymity in a structured overlay. Technical Report DISI-TR-05-05, Universita' di Genova, May 2005.

[18] Giuseppe Ciaccio. A pretty flexible API for generic peer-to-peer programming. Technical Report DISI-TR-05-06, DISI, Università di Genova, May 2005.

[19] Ian Clarke, Scott G. Miller, Theodore W. Hong, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[20] Danilo Conde and Emilio Francesquini. 4-ação. PLoP-IME, IME-USP, São Paulo, SP, 2002.

[21] Arturo Crespo and Hector Garcia-Molina. Semantic overlay networks for P2P systems. In Gianluca Moro et al., editors, *AP2PC*, volume 3601 of *LNCS*, pages 1–13. Springer, 2004.

[22] Frank Dabek, Ben Zhao, Peter Druschel, John Kubiatowicz, and Ion Stoica. Towards a common API for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, February 2003.

[23] Diego Doval and Donal O'Mahony. Overlay networks: A scalable alternative for P2P. *IEEE Internet Computing*, 7(4):79–82, 2003.

[24] Elmasri and Navathe. *Fundamentals of database systems (3rd edition)*. Addison-Wesley Longman, August 1999.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1994.

[26] Andrei Goldchleger. Integrade: Um sistema de middleware para computação em grade oportunista. Master's thesis, Instituto de Matemática e Estatística da USP, São Paulo, SP, December 2004.

[27] Li Gong. Project JXTA: A technology overview. `http://www.jxta.org/project/www/docs/jxtaview_01nov02.pdf`, 2002.

[28] Steve Gribble. Public review for "Design choices for content distribution in P2P networks", by A. Al Hamra and P. A. Felber. *ACM SIGCOMM Comput. Commun. Rev.*, 35(5):29–40, 2005.

[29] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 314–329, New York, NY, USA, 2003. ACM Press.

[30] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex queries in DHT-based peer-to-peer networks. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 242–259, London, UK, 2002. Springer-Verlag.

[31] Jeff Hoye. The freepastry tutorial. Versão 1.3; 1o. de agosto de 2005. Para o FreePastry versão 1.4.2.

[32] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: a decentralized peer-to-peer web cache. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 213–222, New York, NY, USA, 2002. ACM Press.

[33] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The Eigentrust algorithm for reputation management in P2P networks. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 640–651, New York, NY, USA, 2003. ACM Press.

[34] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[35] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[36] Manolis Koubarakis, Iris Miliaraki, Zoi Kaoudi, Matoula Magiridou, and Antonis Papadakis-Pesaresi. Semantic Grid Resource Discovery using DHTs in Atlas. In *3rd GGF Semantic Grid Workshop*, Athens, Greece, February 2006.

[37] John Kubiatowicz. Extracting guarantees from chaos. *Commun. ACM*, 46(2):33–38, 2003.

[38] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[39] Ihor Kuz and Maarten van Steen. cP2Pc: Integrating P2P networks. *Linux Journal*, 110, June 2003.

[40] Steve Lawrence and C. Lee Giles. Searching the World Wide Web. *Science*, 280(5360):98–100, 1998.

[41] David Liben-Nowell, Hari Balakrishnan, and David Karger. Observations on the Dynamic Evolution

of Peer-to-Peer Networks. In *1st Workshop on P2P Systems and Technologies*, Cambridge, MA, March 2002.

[42] Jörg Liebeherr, Jianping Wang, and Guimin Zhang. Programming overlay networks with overlay sockets. In Burkhard Stiller, Georg Carle, Martin Karsten, and Peter Reichl, editors, *Networked Group Communication*, volume 2816 of *Lecture Notes in Computer Science*, pages 242–253. Springer, 2003.

[43] L. Liu and K.D. Ryu. RHT: Supporting range queries in DHT-based P2P systems. In *Proceedings of the 2006 IASTED Conference on Parallel and Distributed Computing and Systems*. Acta Press, 2006.

[44] Boon Thau Loo, Ryan Huebsch, Ion Stoica, and Joseph M. Hellerstein. The case for a hybrid P2P search infrastructure. In Geoffrey M. Voelker and Scott Shenker, editors, *IPTPS*, volume 3279 of *Lecture Notes in Computer Science*, pages 141–150. Springer, 2004.

[45] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, pages 72–93, 2005.

[46] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 84–95, New York, NY, USA, 2002. ACM Press.

[47] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st annual ACM symposium on Principles of distributed computing*. ACM Press, 2002.

[48] Andreas Mauthe and David Hutchison. Peer-to-peer computing: Systems, concepts and characteristics. *Praxis in der Informationsverarbeitung & Kommunikation (PIK), K. G. Sauer Verlag, Special Issue on Peer-to-Peer*, 26(03/03), June 2003.

[49] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the

xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, London, UK, 2002. Springer-Verlag.

[50] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO '87: A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, pages 369–378, London, UK, 1988. Springer-Verlag.

[51] Michael Miller. *Discovering P2P*. Editora Sybex, 2001.

[52] D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing. Technical report, Hewlett-Packard, 2002.

[53] A. Mislove, C. Reis, A. Post, P. Willmann, Dru P. Schel, D. Wallach, X. Bonnaire, P. Sens, J. M. Busca, and L. Arantes. Post: A secure, resilient, cooperative messaging system. In E. Iee, editor, *Ninth IEEE Workshop on Hot Topics in Operating Systems (HotOS-IX)*, Kaui (USA), June 2003. IEEE Society Press.

[54] Object Management Group. IDL to Java Language Mapping Specification, August 2002. Version 1.2, formal/02-08-05.

[55] Manoj Parameswaran, Anjana Susarla, and Andrew B. Whinston. P2P networking: An information-sharing alternative. *Computer*, 34(7):31–38, 2001.

[56] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[57] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. In *International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[58] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: a public DHT service and its uses. In *SIGCOMM '05: Procee-*

dings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications, pages 73–84, New York, NY, USA, 2005. ACM Press.

[59] Sean C. Rhea. A programmer's tutorial on event-driven programming, asynchronous input/output, and the Bamboo DHT. Documentação do Bamboo DHT.

[60] Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), Aug 2002.

[61] John Risson and Tim Moors. Survey of research towards robust peer-to-peer networks: search methods. Technical Report UNSW-EE-P2P-1-1, University of South Wales, September 2004.

[62] Vladimir Moreira Rocha. Protocolos par-a-par para interligação de aglomerados em grades computacionais. Master's thesis, Instituto de Matemática e Estatística da USP, São Paulo, SP, December 2005.

[63] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329, 2001.

[64] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. *IEEE/ACM Trans. Netw.*, 12(2):219–232, 2004.

[65] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM Press.

[66] Shkapenyuk V. and Suel T. Design and implementation of a high-performance distributed web crawler. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, page 357, Washington, DC, USA, 2002. IEEE Computer Society.

[67] Brandon Wiley. Interoperability through gateways. In Andy Oram, editor, *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., 2001.

[68] Feng Zhou, Li Zhuang, Ben Y. Zhao, Ling Huang, Anthony D. Joseph, and John D. Kubiatowicz. Approximate object location and spam filtering on peer-to-peer systems. In *Proc. of Middleware*, pages 1–20, Rio de Janeiro, Brazil, June 2003. ACM.