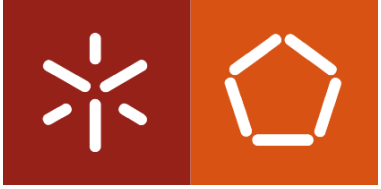

Universidade do Minho
Escola de Engenharia

Óscar Raul Vilela Bravo

Redes overlay peer-to-peer baseadas em SIP

Dissertação de Mestrado
Mestrado em Engenharia de Comunicações

Trabalho efectuado sob a orientação de:
Professora Doutora Maria João Nicolau
Professor Doutor António Costa

Outubro de 2011

Agradecimentos

Gostaria de agradecer à professora Maria João Nicolau pela orientação, dedicação e paciência durante a realização deste trabalho.

Ao co-orientador António Costa, pelos conselhos e pela ajuda com o *cluster*.

À Marta, aos meus pais, irmãos, avós e amigos por todo apoio.

À FCT pelo financiamento da bolsa de investigação no projecto UPCGSM-2010-3, no âmbito do qual foi desenvolvido este trabalho de dissertação.

Ao departamento de informática da Universidade do Minho pela disponibilização do *cluster* SeARCH no qual foram realizados alguns dos testes desta dissertação.

Resumo

As primeiras redes P2P, popularizadas por aplicações como o Napster, caracterizavam-se pelo facto de necessitarem de um servidor central para indexar os recursos disponibilizados pelos *peers* da rede. A utilização de um servidor central tornava a rede mais permeável a ataques de negação de serviço (DoS), e a problemas de escalabilidade. Com a evolução das redes P2P, surgiram novas formas de indexar os recursos: localmente ou de forma distribuída. Nas redes P2P que utilizam um mecanismo de armazenamento local, cada *peer* mantém uma lista dos recursos que possuí. A localização de recursos é feita recorrendo a mecanismos que inundam a rede com mensagens de localização. Este tipo de mecanismo gera muito tráfego e é pouco eficiente. Actualmente, as redes P2P mais populares armazenam a informação dos seus recursos de uma forma distribuída, recorrendo a Distributed Hash Tables(DHTs). Neste tipo de redes a forma como os *peers* são posicionados e os recursos pelos quais cada *peer* é responsável por indexar, é obtido de forma determinística. A localização de recursos é feita de uma forma muito mais rápida e eficiente.

A criação deste tipo de redes, baseando-se em soluções abertas como o protocolo SIP, pode facilitar a criação de novos tipos de serviços e permitir uma mais fácil integração de diferentes serviços. Para além disso, a utilização de uma solução madura, implementada em diversos dispositivos e cujo funcionamento é bem conhecido, é por si só uma vantagem. Adicionalmente, o facto de se usar apenas mensagens SIP na construção e utilização da rede P2P, permite ultrapassar as barreiras criadas por firewalls ou NATs, o que representa também uma mais-valia.

Neste trabalho foi desenvolvida uma implementação JAVA, capaz de criar redes P2PSIP com um ou dois níveis hierárquicos. A existência de uma hierarquia de dois níveis visa comprovar que em determinadas situações a rede *overlay* beneficia da existência de uma hierarquia deste tipo. A comunicação entre os nós da rede P2P é feita através de um protocolo totalmente baseado em SIP. Como algoritmos a utilizar pelo overlay P2P, foram implementados o algoritmo Chord e EpiChord. Para comprovar o funcionamento da implementação, foram efectuados testes num ambiente real, recorrendo numa primeira instância a uma topologia de rede emulada com o CORE, e posteriormente a um cluster no qual foram efectuados testes com um maior número de nós.

Abstract

The first P2P networks, popularized by applications like Napster, required a central server to index the resources provided by peers in the network. The use of a central servers makes the network more susceptible to denial of service (DoS) attacks, and creates scalability issues. With the evolution of P2P networks, two new ways for indexing resources were introduced: locally or distributed. In P2P networks that uses a local storage mechanism, each peer maintains a list of their resources. The location of resources in this kind of network is done with flood based mechanisms. This kind of mechanism floods the network with messages which generates generates a lot of traffic and is very inefficient. Currently, the most popular P2P networks store information of its resources in a distributed manner, using Distributed Hash Tables (DHTs). In this type of networks the way peers are positioned and resources in which each peer is responsible for indexing, is obtained deterministically. The location of resources is done in a much faster and efficient way.

The creation of such networks, based on open solutions like SIP, can facilitate the creation of new types of services and allow easier integration of different services. In addition, the use of a mature solution, implemented on multiple devices and whose operation is well known, is itself an advantage. Another advantage is the fact that using only SIP messages on the P2P network, can overcome the barriers created by firewalls or NATs.

In this work we developed a Java implementation, which can create P2PSIP networks with one or two hierarchical levels. The existence of a two-level hierarchy is aimed to prove that in certain situations the overlay network benefits from the existence of such a hierarchy. The communication between nodes in the P2P network is done through a protocol based entirely on SIP. As the algorithms used by the P2P overlay, Chord and EpiChord have been implemented. To prove the functioning of our implementation, tests were made in a real environment, using, in a first instance an emulated network topology with CORE, and, later a cluster in which tests were conducted with a larger number of nodes.

Conteúdo

Agradecimentos **iii**

Resumo **v**

Abstract **vii**

Lista de figuras **xi**

Lista de tabelas **xiii**

Acrónimos **xv**

1 Introdução **1**
1.1 Enquadramento 1
1.2 Objectivos 2
1.3 Contribuições 3
1.4 Estrutura da dissertação 3

2 Redes Peer-To-Peer **5**
2.1 *Overlays* Não estruturados 7
2.2 *Overlays* Estruturados 10
2.3 Protocolos P2P 11
2.3.1 Gnutella 11
2.3.2 Chord 15
2.3.3 EpiChord 20

3 Redes Peer-to-Peer baseadas em SIP **27**
3.1 *Session Initiation Protocol* (SIP) 27
3.1.1 Componentes Principais 28
3.1.2 Estrutura das mensagens 30
3.2 A Secure Architecture for P2PSIP-based Communication Systems 33
3.2.1 Arquitectura Proposta 33
3.2.2 Testes e Resultados 39
3.3 *Distributed Session Initiation Protocol* (dSIP) 41
3.3.1 Estrutura do *Overlay* P2P 41

3.3.2 Mensagens SIP . . . 42
3.4 Peer-to-Peer Protocol (P2PP) . . . 45
3.4.1 Arquitectura do Protocolo . . . 46
3.4.2 Formato das Mensagens . . . 47
3.5 REsource LOcation And Discovery (RELOAD) . . . 49
3.5.1 Arquitectura . . . 49

4 Uma proposta de arquitectura P2PSIP hierárquica 51
4.1 Hierarquia com dois níveis . . . 52
4.2 *Overlay* peer-to-peer . . . 54
4.3 Mensagens SIP utilizadas . . . 56
4.3.1 Cabeçalhos das mensagens . . . 57
4.3.2 Tipos de mensagens P2PSIP . . . 59

5 Implementação 63
5.1 Arquitectura . . . 63
5.1.1 Camada P2PSIP . . . 64
5.1.2 Camada DHT . . . 67
5.2 Chord . . . 71
5.2.1 Criação e manutenção do *Overlay* . . . 73
5.2.2 Gestão da tabela de encaminhamento . . . 76
5.2.3 Encaminhamento de mensagens . . . 77
5.2.4 Algoritmo de localização . . . 77
5.2.5 Admissão de *peers* . . . 79
5.2.6 Inserção/remoção e localização de recursos . . . 81
5.3 EpiChord . . . 84
5.3.1 Manutenção do *Overlay* . . . 85
5.3.2 Gestão da tabela de encaminhamento . . . 88
5.3.3 Algoritmo de localização . . . 89
5.3.4 Admissão de *peers* . . . 91
5.3.5 Inserção/remoção e localização de recursos . . . 92
5.4 Cliente . . . 94
5.4.1 Inserção/remoção e localização de recursos . . . 94

6 Testes e Resultados 97
6.1 Ambiente de teste . . . 97
6.2 Validação da Implementação . . . 100
6.3 Peers com limitações . . . 103
6.4 Clientes com limitações . . . 105

7 Conclusão 109

Lista de Figuras

2.1 Topologia P2P não-estruturada 9
2.2 Topologia P2P não-estruturada com super-peers 9
2.3 Topologia P2P estruturada (DHT) 11
2.4 Chord - Exemplo de um *overlay* composto por 3 nós 17
2.5 Chord - Localização de recursos (simples) 18
2.6 Chord - Localização de recursos e exemplo de uma *finger table* 19
2.7 Escolha do destino das **p** mensagens a enviar [4] 22
2.8 Exemplo de um *overlay* com ciclos [4] 26

3.1 Exemplo do registo e localização de um utilizador 30
3.2 Formato de uma mensagem SIP 31
3.3 Exemplo da arquitectura proposta em [12] 34
3.4 Arquitectura de um *Chord Secure Proxy* [12] 34
3.5 Estrutura das mensagens *HelloRequest* e *HelloResponse* 37
3.6 Estratégia de routing semi-recursivo [12] 39
3.7 Comparação do número de saltos [12] 40
3.8 Níveis de confiança num sistema baseado em CSPs [12] 41
3.9 Exemplo de mensagens SIP do protocolo dSIP 44
3.10 Arquitectura de um *peer* P2PP [18] 47
3.11 Formato do cabeçalho das mensagens P2PP [18] 47
3.12 Arquitectura do protocolo RELOAD 50

4.1 P2PSIP - Hierarquia de dois níveis 54
4.2 Chord - Exemplo de um cabeçalho DHT-Link do protocolo dSIP 58
4.3 Exemplo de uma mensagem P2PSIP para o registo de um recurso 60
4.4 Exemplo de uma mensagem P2PSIP para o registo de um *peer* 61
4.5 Exemplo de uma mensagem P2PSIP para a localização de um *peer* 61

5.1 Arquitectura das aplicações - Camadas 64
5.2 Entidades principais da camada P2PSIP 64
5.3 Entidades principais da camada DHT 67
5.4 Exemplo no qual o *peer B* possuí informação de encaminhamento desactualizada 74
5.5 Fluxograma - Obtenção do *peer* responsável por um identificador 78
5.6 Exemplo - Admissão de um novo *peer* 80
5.7 Fluxograma - Processamento de pedido para registo de um recurso 83
5.8 Fluxograma - Processamento de pedido para localização de um recurso . . 84

5.9 Exemplo no qual o *peer B* actualiza a sua tabela de encaminhamento com base no trafêgo que recebe 87
5.10 Fluxograma - Algoritmo de localização do EpiChord 90
5.11 Fluxograma - Processamento de pedido para o registo de um recurso . . . 93
5.12 Fluxograma - Processamento de pedido para localização de um recurso . . 95

6.1 Topologia utilizada para efectuar os testes 98
6.2 Comparação dos resultados dos tempos médios de localização de recursos com e sem *peers* limitados 105
6.3 Comparação dos resultados dos tempos médios de localização de recursos com e sem clientes 107

Lista de Tabelas

6.1 Resultados do teste de *lookup* intensivo com 100 *peers* 102
6.2 Resultados do teste de *lookup* intensivo com 200 *peers* 102
6.3 Resultados do teste de *lookup* intensivo com 100 *peers* - Cenário 1, 2 e 3 . . 104
6.4 Resultados do teste de *lookup* intensivo com 200 *peers* - Cenário 1, 2 e 3 . . 104
6.5 Resultados do teste de *lookup* intensivo com 100 nós (peers e clientes) - Cenário 1, 2 e 3 106
6.6 Resultados do teste de *lookup* intensivo com 200 nós (peers e clientes) - Cenário 1, 2 e 3 106

Acrónimos

API - **A**pplication **P**rogramming **I**nterface

DHT - **D**istributed **H**ash **T**able

DNS - **D**omain **N**ame **S**ystem

DoS - **D**enial **o**f **S**ervice

dSIP - **D**istributed **S**ession **I**nitiation **P**rotocol

HTTP - **H**yper**t**ext **T**ransfer **P**rotocol

IP - **I**nternet **P**rotocol

NAT - **N**etwork **A**ddress **T**ranslation

P2P - **P**eer-**T**o-**P**eer

P2PP - **P**eer-**T**o-**P**eer **P**rotocol

P2PSIP - **P**eer-**t**o-**P**eer **S**ession **I**nitiation **P**rotocol

RELOAD - **RE**source **LO**cation **A**nd **D**iscovery

SeARCH - **Se**rvices and **A**dvanced **R**esearch **C**omputing

SIP - **S**ession **I**nitiation **P**rotocol

TTL - **T**ime **t**o **L**ive

UA - **U**ser **A**gent

UAC - **U**ser **A**gent **C**lient

UAS - **U**ser **A**gent **S**erver

URI - **U**niform **R**esource **I**dentifier

VoIP - **V**oice **o**ver **I**nternet **P**rotocol

Capítulo 1

Introdução

1.1 Enquadramento

O aumento do número de utilizadores na internet, assim como os serviços disponibilizados, faz com que as redes *peer-to-peer*, devido à sua natureza distribuída, assumam um papel cada vez mais importante na Internet. Actualmente existem diversas publicações científicas que abordam os vários aspectos destas redes, existindo inclusive um grupo de trabalho dedicado exclusivamente ao seu estudo: o *Internet Research Task Force - Peer-to-Peer Research Group (IRTF-P2PWG)* [1].

O facto de este tipo de redes estar habitualmente associado à partilha de ficheiros, através de protocolos como Gnutella, BitTorrent, Kazaa, etc, faz com que as redes *peer-to-peer* sejam para muitas empresas da indústria sinónimo de actividades ilegais. Contudo, existem diversas aplicações que beneficiam de um suporte *peer-to-peer*, como por exemplo o Skype.

Muitas das soluções existentes para implementar redes *peer-to-peer* são fechadas, existindo, no entanto, um esforço a nível académico para desenvolver redes *peer-to-peer* genéricas, baseadas em soluções abertas. A utilização de redes genéricas tem algumas vantagens, por não estarem limitadas a um tipo de serviço ou aplicação específica, podendo ser utilizadas para implementar diversos serviços ou suportar múltiplas aplicações. O facto de utilizarem soluções abertas, com uma grande maturidade, como por exemplo

o SIP [2], é também uma vantagem, pois existe um maior conhecimento sobre o funcionamento e desempenho dessas soluções, o que pode facilitar a sua implementação, assim como permitir que entidades diferentes consigam uma melhor interoperabilidade entre si já que o funcionamento da solução utilizada é bem conhecido.

O SIP (*Session Initiation Protocol*) é um protocolo de sinalização standard do IETF (*Internet Engineering Task Force*), que funciona ao nível da camada de aplicação. É bastante utilizado para estabelecer sessões entre um ou vários participantes, que permite que os participantes negoceiem os parâmetros a utilizar tanto no estabelecimento, como durante a sessão. É utilizado em diversas situações, como por exemplo no estabelecimento de chamadas telefónicas através da internet, distribuição de conteúdos multimédia, conferências, etc. Para além disso o SIP possuí um conjunto de mecanismos para ultrapassar alguns problemas relacionados com firewalls e NAT's.

1.2 Objectivos

Os objectivos deste trabalho podem ser sintetizados nos seguintes pontos:

- Compreender o funcionamento das redes *peer-to-peer*
- Avaliar o estado actual das redes *peer-to-peer*
- Estudar o protocolo *Session Initiation Protocol* (SIP)
- Estudar as propostas *peer-to-peer* totalmente baseadas em SIP existentes
- Especificar ou reutilizar um protocolo *peer-to-peer* que seja totalmente baseado em SIP
- Implementar um protótipo que permita avaliar a solução proposta
- Testar a solução proposta e comparar os resultados com outras soluções semelhantes

1.3 Contribuições

No trabalho realizado no âmbito da dissertação foi desenvolvida uma implementação capaz de criar uma rede *peer-to-peer* estruturada, totalmente baseada em SIP. A comunicação é feita através de um protocolo P2PSIP, tendo sido implementados dois algoritmos DHT: o Chord[3] e o EpiChord[4]. Para além da criação de um *overlay* P2PSIP utilizando o Chord ou EpiChord, a aplicação é capaz de criar um *overlay* com dois níveis hierárquicos. O primeiro nível hierárquico é composto exclusivamente por *peers*, formando estes o *overlay*. Num segundo nível encontram-se os clientes que se ligam a um ou vários *peers*, e utilizam os serviços disponibilizados pelo *overlay*, sem que participem activamente na sua construção e gestão. Actuando apenas em seu beneficio. A criação de uma hierarquia deste género visa comprovar que em alguns cenários pode ser melhor para o desempenho do *overlay*, que determinados *peers*, deixem de participar activamente no *overlay*, passando a ser clientes. Por participação activa no *overlay*, compreende-se a troca de mensagens entre *peers* para a gestão e construção do *overlay*, e também armazenamento de informação.

Os resultados preliminares do trabalho desenvolvido no âmbito desta dissertação foram publicados na 11ª Conferência sobre Redes de Computadores (CRC 2011).

1.4 Estrutura da dissertação

No capítulo 1 é feita uma introdução ao tema desta dissertação, descrevendo as motivações para o trabalho, e os respectivos objectivos a alcançar.

O capítulo 2 descreve o estado da arte das redes *peer-to-peer*, descrevendo o funcionamento dos vários protocolos P2P, desde os mais antigos, até alguns dos mais recentes.

No capítulo 3 são apresentadas as soluções *peer-to-peer* SIP actualmente existentes, descrevendo os tipos de mensagens SIP utilizadas a estruturas dos protocolos e os tipos de rede *peer-to-peer* suportadas.

No capítulo 4 é apresentada a descrição e análise do problema, na qual é apresentada a solução proposta para o problema, descrevendo os algoritmos *peer-to-peer* a implementar, o protocolo P2PSIP a utilizar, e as mensagens SIP utilizadas.

No capitulo 5 é descrita a implementação da solução, aprofundando um pouco mais a

forma como a solução foi implementada. São descritos os vários componentes desenvolvidos, a forma como os algoritmos DHT foram implementados, descrevendo alguns dos algoritmos como por exemplo para a localização de recursos.

O capítulo 6 serve para apresentar os testes e os resultados experimentais realizados à aplicação desenvolvida, descrevendo o ambiente no qual os testes foram realizados, os testes efectuados e os respectivos resultados. Foram efectuados vários testes, alguns dos quais para validar e comparar os resultados obtidos com resultados obtidos por outros autores. Enquanto que, outros testes, foram efectuados para verificar se a existência de uma hierarquia de dois níveis seria ou não benéfica para um *overlay peer-to-peer*.

Por fim, o capítulo 7 apresenta as conclusões desta dissertação, resumindo os objectivos cumpridos, as possíveis modificações a efectuar ao sistema e propostas de trabalho futuro.

Capítulo 2

Redes Peer-To-Peer

As redes *peer-to-peer* (P2P) são actualmente bastante populares na internet sendo utilizadas por diversas aplicações para implementar diferentes tipos de serviços, como por exemplo serviços de partilha de ficheiros e serviços de voz sobre IP (VoIP). O Skype é uma das aplicações VoIP mais populares e utiliza uma arquitectura P2P. Para além da partilha de ficheiros e de serviços de voz, muitos outros serviços podem ser implementados recorrendo a soluções P2P. Por exemplo, aplicações que necessitem de efectuar um elevado número de processamento de dados, no qual o processamento não necessite de ser efectuado sequencialmente podendo ser efectuado em paralelo, podem utilizar uma arquitectura P2P, dividindo o processamento dos dados pelos diversos nós da rede. O SETI é um exemplo de uma aplicação deste género que recorre a uma abordagem P2P.

As redes P2P caracterizam-se por serem distribuídas. Os nós (denominados '*peers*') que a compõem formam uma rede designada de *overlay* na qual todos eles devem cooperar entre si. Os nós do *overlay* devem disponibilizar parte dos seus recursos, como poder de processamento e/ou armazenamento, de forma a que em conjunto possam fornecer um determinado serviço. Cada nó desempenha funções de cliente e de servidor, contrariamente ao modelo tradicional cliente-servidor em que os servidores são utilizados para disponibilizarem serviços e onde os clientes actuam apenas como consumidores, actuando apenas para seu próprio beneficio. Um nó de uma rede *peer-to-peer* deve actuar das duas formas: como cliente quando pretende usufruir de uma funcionalidade oferecida pelo *overlay*, e também como servidor, quando necessita de efectuar operações para o beneficio da rede,

por exemplo, no encaminhamento de mensagens entre os nós, ou no armazenamento de informação do *overlay*.

Funções básicas de uma rede P2P

Apesar das funcionalidades que uma rede P2P deve implementar dependerem do tipo de serviço para o qual a rede será utilizada, existe pelo menos uma funcionalidade ou mecanismo, comum a qualquer tipo de serviço [1], que é a localização de nós da rede (*peer discovery*). É necessário que exista um mecanismo para a localização de nós na rede, de modo a que um novo nó possa localizar um ou vários nós pertencente à rede de forma a que ele se possa juntar à rede P2P. Existem diversos mecanismos para a localização de nós, sendo que habitualmente são utilizados mecanismos centralizados. Uma das técnicas mais utilizadas recorre a servidores denominados de *bootstrap servers* cuja função é disponibilizar um conjunto de endereços ip pertencentes a nós do *overlay* que podem estar disponíveis para receber o novo nó que deseja juntar-se ao *overlay*.
Para além da localização de nós na rede, uma rede P2P implementa habitualmente algumas das seguintes funcionalidades:

- Indexação de dados - Os recursos de dados armazenados no *overlay*, devem ser indexados de alguma forma, devendo existir um mecanismo responsável pela indexação dos recursos de dados da rede.
- Armazenamento de dados - Responsável por armazenar, assim como obter dados guardados no *overlay*
- Processamento de dados - Dependendo do tipo de serviço, os *peers* do *overlay*, podem colaborar no processamento de dados.
- Encaminhamento de Mensagens - Responsável pelo encaminhamento de mensagens entre os nós da rede.

Apesar de uma rede P2P se caracterizar pela sua natureza distribuída, algumas das suas funcionalidades podem ser implementadas recorrendo a soluções centralizadas, como por exemplo a indexação dos conteúdos de dados. Contudo, a utilização de servidores centralizados pode trazer alguns problemas, por exemplo problemas de escala e tolerância

a falhas, o que dependendo do tipo de aplicação que se pretende poderá não ser um problema.

Para alguns autores uma rede P2P pode ser classificada pela forma como os dados armazenados são indexados [1], podendo estes ser indexados de uma forma centralizada, localmente, ou de uma forma distribuída. Numa estratégia de indexação centralizada, um servidor central possui referências para todos os conteúdos de dados disponibilizados por todos os nós do *overlay*. O Napster [5], que foi uma das primeiras aplicações P2P para partilha de dados na Internet, utilizava esta abordagem para indexar os conteúdos partilhados pelos nós da sua rede.
Com uma estratégia de indexação local, cada nó referencia apenas os seus conteúdos de dados, não possuindo qualquer conhecimento sobre os conteúdos de outros nós. Esta estratégia é utilizada por diversos protocolos P2P, como por exemplo as primeiras versões do gnutella (até à versão 0.4 inclusive).
Numa estratégia distribuída, as referências para os conteúdos de dados estão distribuídas em vários nós do *overlay*. Exemplos de protocolos P2P que utilizam esta estratégia são os protocolos baseados em DHTs (*Distributed Hash Table*), como por exemplo o Chord ou o Kademlia.

A forma como os dados são indexados e o modo como os nós são posicionados no *overlay*, leva a que alguns autores classifiquem as redes P2P de duas formas distintas [1, 6], não estruturadas e estruturadas.

2.1 *Overlays* Não estruturados

Neste tipo de rede os nós são posicionados de uma forma aleatória no *overlay*, formando uma topologia que habitualmente tem um ou dois níveis hierárquicos[6, 7]. Numa topologia sem hierarquia, os nós que formam o *overlay* estabelecem ligações com outros nós de uma forma aleatória, tendo todos eles igual importância na rede e desempenhando as mesmas funções. Enquanto que numa topologia com uma hierarquia de dois níveis, existe uma distinção entre os nós do *overlay*, existindo dois tipos de nós, denominados de *super-peers* e *peers*.
Um *super-peer* é eleito pela rede, normalmente pelo facto de disponibilizar uma maior

quantidade de recursos e/ou por estar mais tempo disponível comparativamente com a maioria dos restantes *peers*, podendo obviamente o processo de eleição derivar de outros parâmetros que não os aqui referidos. Neste tipo de hierarquia com *super-peers*, a topologia da rede é dividida em dois níveis. Um nível no qual os *peers* normais possuem ligações apenas com *super-peers*, estabelecendo uma ou mais ligações, e um outro nível no qual os super-peers estabelecem ligações entre si, formando um *overlay* no qual só os *super-peer* participam.
Uma vez que, devido às suas características, um *super-peer* possui uma maior importância na rede, estes são responsáveis pelo encaminhamento de mensagens no *overlay*. Actuando como *proxies* em favor dos *peers* normais, devem encaminhar para outros *super-peers* as mensagens enviadas pelos *peers* normais. Para além do encaminhamento de mensagens, os *super-peers* mantêm um índice no qual estão referenciados os recursos de dados partilhados pelos *peers* com os quais possuem uma ligação.

Numa rede P2P não estruturada, o mecanismo utilizado para a localização de recursos na rede, consiste habitualmente na utilização de técnicas de *flooding* [1, 6]. De acordo com estas técnicas a rede é inundada com mensagens para a localização do recurso pretendido (*queries*), existindo um parâmetro (TTL) que especifica o número máximo de saltos que uma mensagem pode dar. Desta forma limita-se a propagação excessiva de mensagens na rede. Este mecanismo de localização de recursos tem alguns problemas devido à grande quantidade de tráfego que pode ser gerado. Além disso, não permite garantir que um recurso que exista na rede seja encontrado, pois o recurso pode estar num nó ao qual as mensagens com a *query* não chegam devido ao número de saltos ter atingido o valor máximo. *Flooding* é um mecanismo considerado eficiente para a localização de recursos populares na rede, estando mais sujeito a falhar quando a popularidade do recurso pretendida é baixa. Este mecanismo quando utilizado numa topologia composta por *peers* e *super-peers*, é normalmente utilizado apenas no *overlay* formado pelos *super-peers*, uma vez que cada *super-peer* possui conhecimento sobre os recursos partilhados por cada um dos *peers* aos quais esta ligado, o que faz com que não haja necessidade de os *peers* normais participarem neste processo.
As figuras 2.1 e 2.2 mostram exemplos de duas topologias não estruturadas. Na figura 2.1 os nós formam um *overlay* no qual não existe qualquer tipo de hierarquia, enquanto que na figura 2.2 existe uma hierarquia de dois níveis, na qual os nós são distinguidos entre

peers normais e super *peers*.

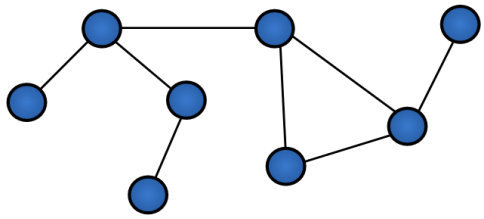

FIGURA 2.1: Topologia P2P não-estruturada

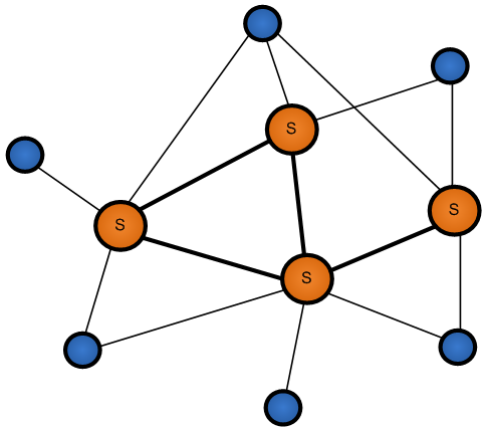

FIGURA 2.2: Topologia P2P não-estruturada com super-peers

2.2 *Overlays* Estruturados

Uma rede P2P estruturada caracteriza-se pelo facto da sua topologia ser bem definida, na qual os nós são posicionados de uma forma controlada, e os recursos são posicionados de uma forma determinística no *overlay* tornando mais eficiente a sua localização.

Este tipo de rede recorre a mecanismos baseados em DHTs para o posicionamento dos recursos no *overlay*. Numa DHT são atribuídos identificadores únicos aos nós da rede (NodeIDs), identificadores esses que podem ser utilizados para o *overlay* decidir onde o nó deverá ficar posicionado, assim como os nós com os quais este deve estabelecer ligações.

Os recursos de dados são também identificados através de identificadores únicos denominados por chaves "*keys*", sendo que esses identificadores podem ser obtidos através de técnicas de *hashing*. Através do identificador de um recurso, o *overlay* faz o mapeamento do nó que deve ficar responsável pelo seu armazenamento, escolhendo o nó com o identificador mais próximo do identificador do recurso a armazenar. De modo a tornar o sistema tolerante a falhas, alguns protocolos baseados em DHT utilizam técnicas de replicação de conteúdos nas quais os recursos são armazenados nos N nós com NodeID mais próximo da chave do recurso a armazenar.

Para efectuar a localização de um determinado recurso na rede, cada nó possui uma tabela com os endereços IP e os respectivos identificadores (NodeID) de alguns nós vizinhos, devendo enviar uma mensagem de *query* para o nó da sua tabela que possui o identificador mais próximo da chave do recurso a localizar. Este processo é repetido até que o nó responsável pelo recurso pretendido é localizado. Em teoria [6], a localização de recursos em DHT's requer em média O(log N) saltos, onde N é o número de nós do *overlay* P2P.

A figura 2.3 mostra uma topologia em anel, representando uma DHT, na qual é possível verificar os identificadores de cada nó, e os identificadores de recursos que cada nó é responsável por gerir.

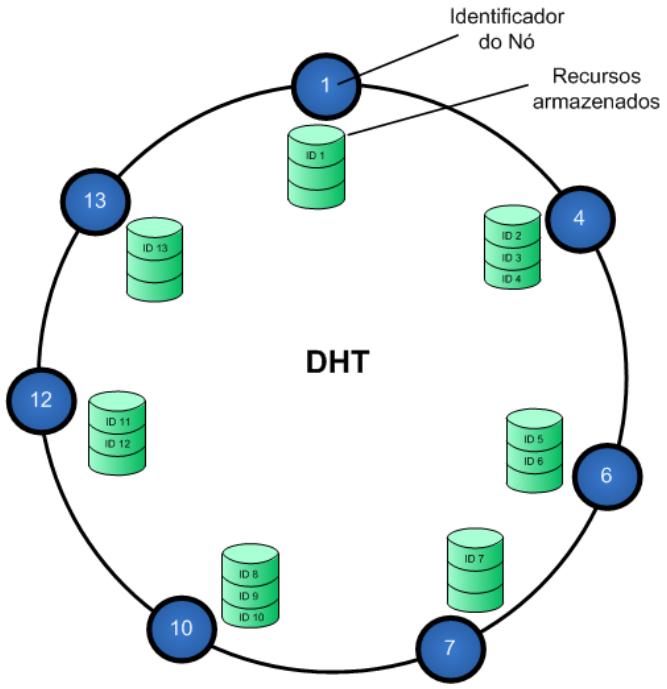

FIGURA 2.3: Topologia P2P estruturada (DHT)

2.3 Protocolos P2P

Nesta secção serão apresentados alguns dos protocolos P2P mais populares.

2.3.1 Gnutella

O gnutella foi o protocolo utilizado nas primeiras redes puramente P2P, tendo sido desenvolvido com o objectivo de permitir efectuar pesquisas em ambientes distribuídos [5, 8]. A versão 0.4 do protocolo é considerada a última versão estável, existindo uma nova versão (0.6) que implementa algumas melhorias no protocolo, nomeadamente ao nível da estrutura do *overlay*.
Nesta secção serão abordadas ambas as versões, sendo descrita mais pormenorizadamente a versão 0.4.

Este protocolo é utilizado por diversas aplicações para a criação de redes *peer-to-peer*, maioritariamente para implementar serviços de partilha de ficheiros. Os nós que compõem uma rede gnutella são designados por *servents*. Um *servent*, implementa funcionalidades

habitualmente associadas a clientes, e a servidores, vindo dai a origem do seu nome (SERVidor e cliENTes).

Nas primeiras versões deste protocolo, até à versão 0.4 inclusive, um *servent* que pretenda juntar-se à rede gnutella, necessita de estabelecer uma ligação com um outro *servent* que pertença à rede. Para tal, deverá conhecer o endereço ip de pelo menos um *servent* que esteja na rede. A forma como os endereços ip dos *servents* pertencentes à rede são obtidos, não se encontra especificada no protocolo, contudo existem várias técnicas que podem ser utilizadas. Por exemplo a utilização de servidores de *bootstrapping* aos quais um *servent* se pode ligar de modo a obter uma lista com endereços IP de servents pertencentes à rede.

O protocolo gnutella, especifica um conjunto de mensagens, que os *servents* devem utilizar para a descoberta de outros *servents*, assim como para a localização de recursos na rede. Todas as mensagens deste protocolo são compostas por um cabeçalho e pelo corpo da mensagem.
O cabeçalho é composto pelos seguintes campos:

- *Message ID* - String de 16 bytes que serve para identificar inequivocamente uma mensagem na rede.
- *Payload Descriptor* - Parâmetro composto por um byte que identifica o tipo de mensagem.
- *TTL* - Este parâmetro contêm o número de vezes que a mensagem pode ser reencaminhada de um *servent* para outro. Cada vez que um *servent* retransmite a mensagem o valor deste parâmetro é decrementado. Uma vez atingido o valor zero, a mensagem é removida da rede.
- *Hops* - Este parâmetro, contêm o número de vezes que a mensagem foi reencaminhada.
- *Payload Length* - Parâmetro que especifica o tamanho, em bytes do corpo da mensagem.

Os parâmetros que compõem o corpo da mensagem variam consoante o tipo de mensagem, existindo mensagens que possuem apenas cabeçalho.
Os tipos de mensagem base deste protocolo são:

- PING - Utilizado para descobrir nós na rede. Um *servent* que recebe uma mensagem deste tipo, deverá enviar pelo menos uma mensagem do tipo PONG.
- PONG - Esta tipo de mensagem é enviado como resposta a uma mensagem do tipo PING, inclui o endereço de um *servent* da rede, assim como informação sobre os recursos de dados que o *servent* disponibiliza.
- QUERY - Este tipo de mensagem é utilizado para efectuar pesquisas na rede gnutella, um *servent* que receba uma mensagem deste tipo, deverá responder com uma mensagem do tipo QueryHit no caso de possuir a informação pretendida pela pesquisa.
- QueryHits - Este tipo de mensagem é enviado como resposta a uma pesquisa, esta mensagem contém informação suficiente para que o receptor da mensagem possa estabelecer ligações de modo a obter o recurso pretendido.
- Push - Este tipo de mensagem é utilizado por *servents* que estejam protegidos por uma *firewall* para que possam partilhar ficheiros de dados com a rede.

Encaminhamento de Mensagens

Em termos de encaminhamento de mensagens (*routing*) na rede gnutella, o protocolo específica que todos os *servents* devem participar no encaminhamento das mensagens. Não existe qualquer distinção entre *servents* com maior ou menor capacidade de recursos, estando o encaminhamento das mensagens sujeito a um conjunto de regras que devem ser seguidas de modo a que o bom funcionamento da rede possa ser assegurado. Todas as mensagens enviadas possuem um cabeçalho no qual existe, entre outros, um parâmetro TTL que especifica o número máximo de saltos que uma mensagem pode atingir. Cada vez que uma mensagem é retransmitida o valor desse parâmetro decresce, e uma vez atingindo o valor 0, a mensagem não poderá ser retransmitida para outro servent.

Localização de Recursos

Para a localização de recursos na rede, um *servent* deverá enviar uma mensagem do tipo *Query* para todos os *servents* aos quais se encontra ligado, sendo que na mensagem enviada, é especificado qual o recurso que se pretende localizar. Um *servent* que receba uma mensagem do tipo *Query*, deverá fazer o seguinte:

- Verificar se possui o recurso pretendido, se possuir deverá enviar uma mensagem do tipo *QueryHit* para o *servent* que lhe enviou a mensagem do tipo *Query*.
- O valor do parâmetro *TTL* da mensagem deve ser decrementado, e se o novo valor for superior a zero, a mensagem de *Query* recebida, deve ser retransmitida para todos os *servents* com os quais mantém uma ligação, com excepção do *servent* que lhe enviou a mensagem de *Query*.

O mecanismo para localização de recursos utilizado necessita de inundar a rede com mensagens para efectuar a pesquisa de um determinado recurso, o que pode levar a uma grande quantidade de tráfego gerado na rede. O parâmetro *TTL* é utilizado para limitar a propagação das mensagens, contudo este mecanismo não consegue garantir a não existência de um recurso na rede. Em determinados casos, mesmo quando a pesquisa não retorna qualquer resultado válido, é possível que o recurso pretendido esteja disponível num *servent* ao qual a mensagem de *Query* não chega pelo facto do *TTL* atingir o valor zero antes de a mensagem lá chegar. Nesse caso a pesquisa não retorna qualquer resultado válido, dando erradamente a ideia de que o recurso não existe, quando de facto ele existe, apenas se encontra num ponto mais distante da rede.

Modificações Gnutella 0.6

A versão 0.6 do gnutella trouxe melhorias significativas no protocolo, tendo adoptado o conceito de *super-peers*, modificando a forma como os *servents* se ligam entre si. Enquanto que nas versões anteriores um *servent* estabelecia ligações de forma aleatória com outros *servents*, nesta versão, existe uma estrutura hierárquica [9] denominada de *ultra-peer system*, na qual os *servents* podem ser caracterizados de duas formas distintas, como *leaves*

ou *ultrapeers*.

Nesta nova estrutura, os *servents* denominados por *leaves* são caracterizados por possuírem menores capacidades em termos de recursos disponíveis e/ou limitações ao nível de conectividade (firewalls, NATs..), e devem estabelecer ligações apenas com *ultrapeers*. Os *ultrapeers* por sua vez são *servents* que possuem uma maior capacidade em termos de recursos, devendo estabelecer ligações com outros *ultrapeers* formando dessa forma uma arquitectura de rede com dois níveis. Um nível no qual os *ultrapeers* se encontram ligados entre si, e um outro nível no qual os *leaves* se encontram ligados a um ou vários *ultrapeers*. A figura 2.2 na página 9 mostra um exemplo de como os *servents* se podem ligar nesta estrutura.

Para além da utilização de uma estrutura baseada em *ultra-peers*, o protocolo de encaminhamento de mensagens, foi também ele melhorado, sendo agora designado por *Query Routing Protocol* (QRP) [6]. Neste protocolo os nós menos importantes *leaves*, enviam para os *ultra-peers*, o *hash* de algumas informações, como por exemplo palavras-chaves relativas aos recursos de dados que possuem, de modo a que os *ultra-peers* possam indexar esses recursos. Isto permite reduzir o tráfego gerado pelas mensagens de localização de recursos, nomeadamente nos nós com menores capacidades (*leaves*), uma vez que os *ultra-peeres* em conjunto possuem informação suficiente para conseguirem localizar os nós que possam possuir o recurso pretendido. Uma mensagem de localização de um recurso apenas é retransmitida para um nó do tipo *leaf* se o recurso pretendido corresponder a um dos recursos disponíveis nesse nó, fazendo com que estes nós não tenham de receber mensagens para a localização de um recurso que não possuem.

2.3.2 Chord

O protocolo Chord foi um dos primeiros protocolos P2P a recorrer a *Distributed Hash Tables* (DHTs), tendo sido desenvolvido de forma a oferecer às aplicações as seguintes características [3]:

- Balanceamento de carga - As chaves que identificam os recursos, são distribuídas uniformemente pelos nós da rede, o que de algum modo contribui para o balanceamento da carga.

- Descentralização - É totalmente distribuído, não existe distinção entre os nós da rede, não havendo nós com uma maior importância que outros.
- Escalabilidade - O custo da localização de um recurso numa rede chord aumenta com o logaritmo do número de nós, o que torna o sistema escalável mesmo com um grande número de nós.
- Disponibilidade - Os nós actualizam automaticamente as suas tabelas de modo a reflectir a entrada ou saída de nós na rede, assegurando que mesmo que alguns nós falhem, o nó responsável por uma determinada chave, é sempre encontrado.

Este protocolo baseia-se em algoritmos de *hashing* consistente, como o SHA-1, de modo a criar um espaço de endereçamento circular de modulo 2^m, no qual existem 2^m identificadores disponíveis. A cada nó do *overlay* é atribuído um identificador único (node ID), obtido através da aplicação do algoritmo de *hashing* sobre o endereço IP do nó, posicionando cada nó numa posição específica no anel do *overlay*. O identificador é composto por **m** bits do resultado da função de *hash*. Os recursos de dados são identificados também eles por um identificador, possuindo cada nó um conjunto de pares chave-valor, em que a chave é um identificador único do recurso, e o valor é informação sobre o recurso de dados em si. O identificador do recurso pertence ao mesmo espaço de endereçamento dos identificadores dos nós da rede, podendo estes ser obtidos através do *hashing* dos dados do recurso, do seu nome, ou a partir de outros parâmetros.

Cada nó na rede é responsável por armazenar um conjunto de pares chave-valor, sendo que uma chave **k** é atribuída a um nó, se o identificador do nó no espaço de endereçamento for igual ou superior ao identificador **k**, ficando este nó conhecido como o sucessor da chave **k** (sucessor(k)).

A figura 2.4 mostra um exemplo de uma rede chord (m = 3), na qual é possível verificar o comportamento do algoritmo relativamente à distribuição das chaves pelos nós da rede. Neste exemplo, como o valor de m é 3, o *overlay* pode ter no máximo 8 nós, uma vez que existem apenas 8 identificadores possíveis tanto para os nós como para identificar os recursos. É possível verificar na figura a existência de recursos no *overlay* com chaves 1, 2, 5 e 7 e que apenas os nós 0, 1 e 3 fazem parte do *overlay*. O facto de o *overlay* ser composto apenas por esses três nós leva a que estes tenham de guardar informação

sobre recursos que possuem um identificador diferente do seu. Por exemplo, o nó 0, é o responsável por armazenar a informação relativa aos recursos com chaves 5 e 7, visto que este nó é primeiro sucessor de ambas as chaves, pois segundo a especificação, uma chave K deve ser atribuída ao nó com identificador igual ou superior ao da chave K.

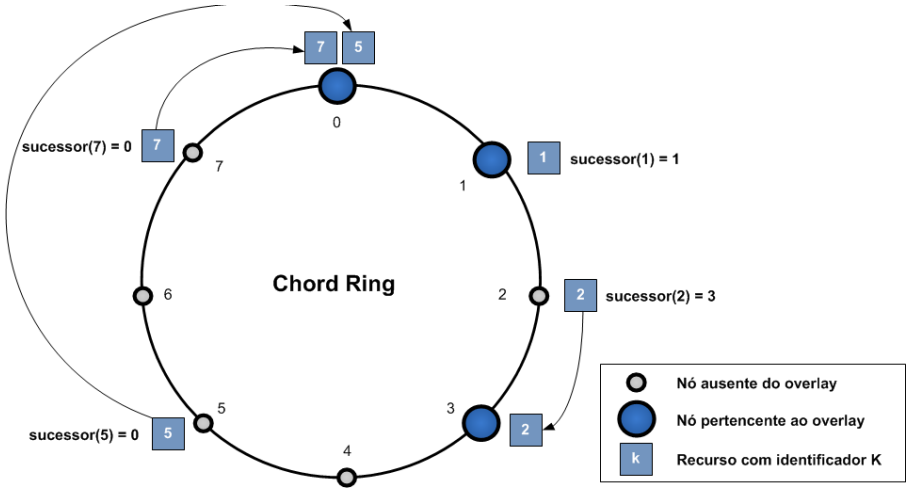

FIGURA 2.4: Chord - Exemplo de um *overlay* composto por 3 nós

Quando um novo nó **n** se junta ao *overlay*, este passa a ser responsável por armazenar algumas das chaves que estavam sobre a responsabilidade do seu sucessor. Por exemplo, utilizando a topologia da figura 2.4, se ao novo nó **n** fosse atribuído o identificador 6, este passaria a ficar responsável pelo recurso com identificador **5**, que anteriormente estava sobre a responsabilidade do nó 0.

Assim que um determinado nó abandona o *overlay*, todas as chaves que lhe tinham sido atribuídas, são atribuídas ao nó que é seu sucessor.

Localização de recursos

Para efectuar a localização de um recurso numa rede chord, basta que cada nó mantenha actualizada a informação sobre o seu sucessor no espaço de endereçamento da topologia, sendo que as mensagens para a localização do recurso vão circulando de sucessor em sucessor, até que o nó responsável pela chave que identifica o recurso pretendido é encontrado [3].

A figura 2.5 mostra um exemplo da utilização deste método de localização de recursos, no qual é possível verificar o trajecto de uma mensagem para a localização do recurso com

chave 54, enviada pelo nó 8.

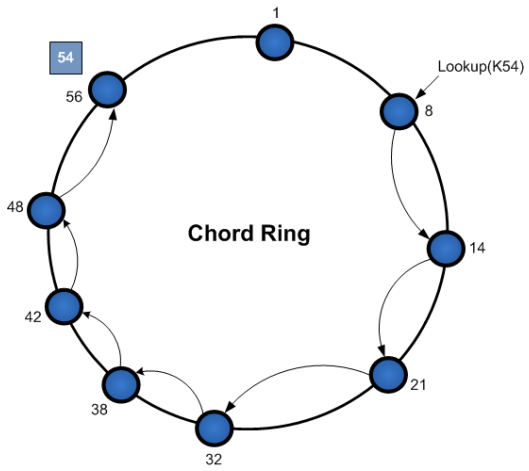

FIGURA 2.5: Chord - Localização de recursos (simples)

Uma vez que no mecanismo descrito anteriormente, a localização de recursos é feita de modo linear, uma mensagem pode ter de percorrer a rede quase toda até ser encontrado o nó com a chave pretendida, necessitando em média de O(n) saltos até a mensagem chegar ao destino.

Para contornar esse problema, o chord implementa um outro mecanismo [3] no qual cada nó mantêm uma tabela, designada por *finger table* que contém informação sobre um conjunto de nós que se encontram a uma distância bem definida no anel do *overlay*.

Segundo a especificação[3] a *finger table* é composta por no máximo **m** entradas, onde cada entrada da tabela possuí informações relativas a um nó especifico, tais como o seu identificador, endereço IP e porta. A entrada **i** de uma *finger table* (onde i está compreendido entre 1 e m) pode ser obtida através da seguinte formula:

$entrada[i] = sucessor((n + 2^{i-1}) \ mod \ 2^m)$

Com a utilização da *finger table* o encaminhamento de mensagens é feito de uma forma mais eficiente, visto que num único salto é possível alcançar um nó que está pelo menos a meio do caminho restante para atingir o destino da mensagem. O preenchimento das entradas da *finger table* é feito periodicamente de modo a manter a tabela actualizada. Para tal são feitas pesquisas pelas chaves correspondentes a cada entrada da tabela.

A figura 2.6 contêm um exemplo de um *overlay* chord (m = 6), no qual os identificadores dos nós e dos recursos são compostos por 6 bits, permitindo que o *overlay* possa ter

no máximo 64 nós. A primeira figura(a) mostra a tabela do nó 8, onde a primeira entrada da tabela possui um apontador para o nó 14, visto que este é o primeiro nó no espaço de endereçamento que sucede a $(8 + 2^0) \bmod 2^6 = 9$. Da mesma forma, a última entrada da tabela possui informações sobre o nó 42, uma vez que este é o primeiro nó que sucede a $(8 + 2^5) \bmod 2^6 = 40$.

A segunda figura (b) contêm um exemplo idêntico ao da figura 2.5, utilizando neste exemplo uma *finger table*, sendo possível verificar que esta permite reduzir substancialmente o número de saltos necessários para que uma mensagem chegue ao destino.

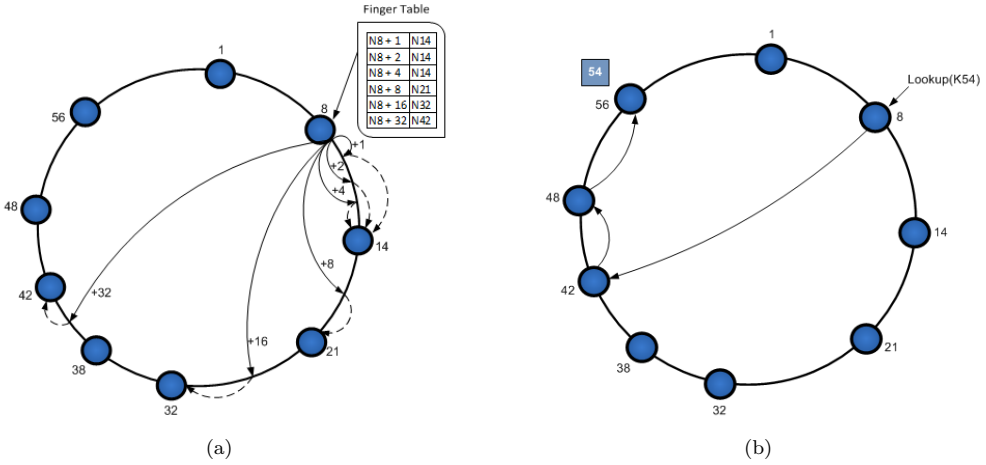

FIGURA 2.6: (a) Conteúdo da tabela do nó 8. (b) O percurso efectuado por uma mensagem enviada pelo nó 8 para localizar o recurso com chave 54. [3]

Uma das características deste mecanismo de localização de recursos é o facto de que numa rede composta por **N** nós, o número de saltos que uma mensagem dá até o recurso ser localizado, é em média O(log N) [3].

Visto que numa rede P2P, existem constantemente nós a entrar e a sair, o chord implementa mecanismos para que os seus nós mantenham as suas tabelas devidamente actualizadas.

Estabilização do protocolo

De forma a manter a estabilidade do *overlay*, um *peer* Chord necessita de guardar informação sobre os seus vizinhos, o *peer* que o sucede e o que o antecede. A informação sobre os seus vizinhos deve ser actualizada periodicamente. O Chord define um mecanismo de estabilização no qual cada *peer* contacta periodicamente o seu sucessor, pedindo-lhe

informação sobre o seu antecessor. Esta informação é importante para que um *peer* possa detectar a entrada ou saída de *peers* do *overlay*. Assim que o *peer* recebe a resposta do seu sucessor, verifica se o antecessor do seu sucessor é o próprio *peer*, se não for, significa que um novo *peer* entrou no *overlay*, e ficou posicionado entre eles, significando isto que deverá ser este o seu novo sucessor. Neste caso, o *peer* deverá contactar o novo sucessor de forma a que este tome conhecimento da sua existência considerando-o seu antecessor. Periodicamente cada *peer* envia também uma mensagem para o seu antecessor, de modo a verificar se este ainda está no *overlay*, removendo-o de seu antecessor, caso este não responda.

2.3.3 EpiChord

O EpiChord é um algoritmo de localização de recursos para redes *peer-to-peer* baseadas em *Distributed Hash Tables* (DHTs), desenvolvido por Ben Leong et al. [4].
Uma das principais características que diferencia o EpiChord de outros algoritmos DHT existentes, é a utilização de uma tabela de encaminhamento (denominada pelos autores por *cache*) sem limite máximo de entradas. Outros algoritmos DHT como por exemplo, o Chord possuem um limite máximo de O(log N) entradas.
Devido ao facto de o número de entradas na tabela de encaminhamento poder afectar o número de saltos necessários para localizar um recurso no *overlay*, os autores do EpiChord afirmam que este permite reduzir em média o número de saltos necessários para a localização de um recurso.

Estratégia de Encaminhamento

Para além da utilização de uma *cache* com tamanho indefinido, o EpiChord implementa uma nova estratégia de encaminhamento denominada por *reactive routing*. Esta estratégia de encaminhamento utiliza as mensagens de localização de recursos para transportar informação útil para a manutenção da *cache* dos nós do *overlay*. Desta forma, os nós do *overlay*, conseguem manter parte da sua *cache* actualizada observando apenas o tráfego de localização de recursos que vão recebendo, adicionando informação de encaminhamento às mensagens de resposta que enviam. Contrariamente, outros DHT's como por exemplo

o Chord, necessitam de enviar periodicamente mensagens para as várias entradas das suas tabelas de encaminhamento (*finger table* no caso do Chord), de forma a verificar a validade das mesmas. No EpiChord, este comportamento é apenas necessário, se o tráfego na rede for demasiado baixo, pois nesse caso o número de mensagens que cada nó recebe pode não ser suficiente para manter a sua *cache* minimamente actualizada. Nesse caso torna-se necessário enviar mensagens para apenas algumas entradas da *cache*, de modo a manter a *cache* minimamente actualizada.

Com esta estratégia de encaminhamento, a informação contida na *cache* nem sempre é a mais actualizada quando comparada com outras estratégias de encaminhamento. Pelo que para a localização de recursos é necessário recorrer ao envio de mensagens em paralelo, de modo a minimizar o efeito da existência de entradas inválidas na *cache*. O envio de mensagens em paralelo, nem sempre é uma boa solução pelo tráfego extra que gera, contudo como o EpiChord possui uma *cache* muito grande o número de saltos necessários para localizar o recurso é mais reduzido o que reduz também o número de mensagens de localização a enviar e o respectivo tráfego na rede.
Os autores [4] afirmam que o envio de mensagens em paralelo do EpiChord quando comparado com uma implementação Chord tradicional, com tráfego de localização de recursos semelhante, consegue melhorar em média a performance da localização de recursos tanto em número de saltos como na latência.

Localização de Recursos

A localização de recursos no EpiChord é feita recorrendo ao envio de mensagens (denominadas por *queries*) em paralelo, sendo que para se iniciar a localização de um recurso são enviadas **p** mensagens em paralelo, onde **p** é um parâmetro configurável do sistema.
O algoritmo utilizado para determinar o destino de cada uma das **p** mensagens a enviar é simples, é consultada a *cache* para se obter um conjunto de nós mais próximos do nó responsável pelo recurso a localizar. Por exemplo, denominando por **id** o identificador do recurso a localizar, o envio das **p** mensagens é feito da seguinte forma: é enviada uma mensagem para o nó da *cache* que é o sucessor imediato do **id** a localizar e de seguida, envia-se para os **p-1** nós da *cache* que antecedem o **id** do recurso. A figura 2.7 mostra um

exemplo do algoritmo descrito anteriormente, neste exemplo um nó **x** pretende localizar o recurso identificado por **id**.

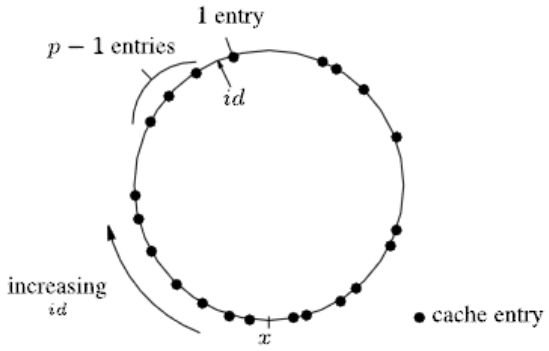

FIGURA 2.7: Escolha do destino das **p** mensagens a enviar [4]

Quando um nó **x** inicializa a localização de um recurso identificado por **id** através do envio de **p** mensagens, os nós que receberem mensagens para a localização do recurso, devem enviar uma mensagem de resposta, de acordo com as seguintes condições:

- Se o nó for responsável pelo recurso pretendido, a resposta deve conter o valor associado ao recurso (caso exista), assim como deve também enviar informação de *routing*, contendo os dados relativos ao seu antecessor.
- Se o nó é antecessor de **id** relativamente ao nó **x** a mensagem de resposta deve conter informação sobre o sucessor do nó, assim como os **l**[1] melhores próximos saltos obtidos através da *cache*, onde **l** é um parâmetro configurável do sistema.
- Se o nó é sucessor de **id** relativamente ao nó **x** a mensagem de resposta deve conter informação sobre o antecessor do nó, assim como os **l** melhores próximos saltos obtidos através da *cache*.

Um dos motivos pelo qual um *peer* responde com informação sobre o seu sucessor ou antecessor, para além dos L contactos, é para que no caso de os L contactos estarem desactualizados, quem originou a localização consiga sempre aproximar-se um pouco mais do destino. Isto porque, a informação sobre o sucessor e antecessor de cada *peer* está sempre mais actualizada do que a informação que armazena em *cache*.

[1]Os **l** melhores próximos saltos referem-se ao *peer* que sucede o identificador do recurso, e os **l-1** nós que antecedem o recurso

À medida que as respostas vão sendo recebidas pelo nó **x**, este verifica a informação de *routing* nelas contidas, actualizando a sua *cache* com essa informação. Para além da actualização da *cache*, **x** envia novas mensagens de localização em paralelo para os nós contidos nas mensagens de resposta e que se encontram mais próximos do destino comparativamente aos nós que já responderam. Isto é, à medida que os nós vão respondendo, são actualizados dois parâmetros que identificam os dois nó mais próximos (sucessor e antecessor do identificador a localizar) do destino, sendo apenas enviadas novas mensagens se o destinatário estiver mais próximo do destino que o actual melhor sucessor ou antecessor.
O algoritmo de localização utiliza uma estratégia de *routing* iterativo, o que permite que o nó que inicia a localização possa verificar se está a aproximar-se do nó pretendido, assim como manter um histórico das mensagens enviadas. Desta forma é possível detectar casos em que um nó é referenciado múltiplas vezes nas mensagens de resposta de outros nó, evitando-se o envio da mesma mensagem múltiplas vezes para o mesmo nó.

Gestão da *Cache*

Os registos guardados na *cache* são compostos pelo endereço IP e porta do nó, o número de falhas no envio de mensagens, e um parâmetro que guarda o instante de tempo em que o nó foi contactado na altura em que o registo foi originalmente criado. Este parâmetro permite verificar num dado instante, há quanto tempo o nó associado ao registo foi contactado pela última vez, podendo ser utilizado para descartar registos que ultrapassam um determinado valor máximo de tempo.
Sempre que um nó recebe uma mensagem, este verifica na sua *cache* se já existe algum registo associado ao emissor da mensagem. No caso de não existir, é adicionado um novo registo à *cache* sendo guardado o instante de tempo em que o registo foi adicionado. Caso já exista na *cache* uma entrada associada ao nó que enviou a mensagem, o número de falhas no envio de mensagens é reposto a zero e é actualizado o instante de tempo associado ao registo.

Quando um nó envia mensagens de resposta, cada registo da *cache* que é enviado na resposta, contêm o parâmetro *lifetime*, que indica à quanto tempo (em segundos) o registo

foi obtido[2].

Um nó ao processar a informação de *routing* recebida numa mensagem, verifica se para cada registo existente na mensagem já existe um registo na sua *cache*. No caso de não existir, é adicionado um novo registo na *cache* e é utilizado o parâmetro *lifetime* contido na mensagem para calcular o instante de tempo no relógio local do nó em que o registo foi obtido originalmente. Caso já exista na *cache* é calculado o *lifetime* do registo contido na *cache* e comparado com o da mensagem recebida. Se o valor do *lifetime* da mensagem for menor que o da *cache* do nó, significa que o registo existente na mensagem é mais recente do que o registo existente na *cache* do nó. Nesse caso o nó actualiza o registo repondo a zero o número de falhas no envio de mensagens, e actualiza o instante de tempo do registo da sua *cache* utilizando o valor do *lifetime* da mensagem como referência.

A verificação de registos expirados na *cache* é feita periodicamente, sendo removidos os registos cujo seu *lifetime* ou número de falhas no envio de mensagens ultrapassem um valor máximo definido.

Estabilização

A saída abrupta (quando um o nó abandona o *overlay* sem anunciar a sua saída aos seus vizinhos directos) de nós do *overlay*, ou a entrada de múltiplos nós aproximadamente na mesma localização, pode criar inconsistências temporárias. Por exemplo um nó pode não saber que passou a ser responsável por gerir uma determinada zona do espaço de endereçamento. Para evitar esses problemas, o EpiChord, como o Chord, utiliza mecanismos para estabilizar o *overlay*.

O EpiChord implementa dois mecanismos de estabilização do *overlay* denominados por *Weak Stabilization protocol* e *Strong Stabilization protocol*.

Weak Stabilization protocol

Este protocolo de estabilização, tem como objectivo garantir que nó possui a informação relativa ao seu sucessor e antecessor actualizada. Periodicamente são enviadas mensagens

[2]O parâmetro *lifetime* é obtido através da subtracção do instante de tempo na altura do envio da mensagem com o valor do instante de tempo da última actualização do registo existente na *cache*.

para os seus vizinhos directos (sucessor e antecessor), de modo a verificar se estes se encontram activos. Estes enviam na mensagem de resposta informação sobre a sua *cache*, podendo enviar apenas informação sobre os vizinhos directos, ou enviar um conjunto de K sucessores e K antecessores.

Segundo este protocolo, cada nó é responsável por manter actualizada a informação sobre quem é o seu sucessor e antecessor. Por isso, quando um nó recebe uma mensagem de um outro nó que se encontra mais próximo de si do que o seu sucessor ou antecessor actual, este actualiza o registo associado ao seu sucessor ou antecessor (consoante a localização) para o novo nó.
Quando um nó descobre indirectamente, através de outros nós, por exemplo, observando tráfego de localização a existência de um nó que se encontra mais próximo de si do que o seu sucessor ou antecessor actual, é enviada uma mensagem para esse nó. A mensagem enviada serve para verificar se esse nó se encontra disponível, passando a ser o novo sucessor ou antecessor (consoante a localização) se e só se a resposta à mensagem enviada for positiva.

Strong Stabilization protocol

Em determinadas situações, o *overlay* pode originar ciclos, ciclos esses que não são detectados pelo processo de estabilização descrito anteriormente. Por isso, o EpiChord define um outro protocolo de estabilização que visa corrigir as inconsistências globais do *overlay*. A ideia por de trás deste protocolo é simples, consiste no envio de uma mensagem, e esperar que este seja reenviada de *peer* para *peer*, até voltar ao *peer* que a enviou.

A figura 2.8 mostra um exemplo de um *overlay* EpiChord, no qual os *peers* que o forma originaram ciclos. As setas indicam os sucessores de cada *peer*. Neste exemplo, o *peer* **n** inicializa o processo de estabilização, enviando uma mensagem contendo o seu identificador. Assim que a mensagem chega ao *peer* **m**, este apercebe-se da existência de **n** e corrige a informação sobre o seu sucessor. Este mecanismo, de enviar uma mensagem e esperar que esta passe por todos os *peers* do *overlay*, é segundo os autores do EpiChord, ineficiente e pode demorar muito tempo. Por isso, no EpiChord este protocolo é implementado através do envio de mensagens em paralelo.

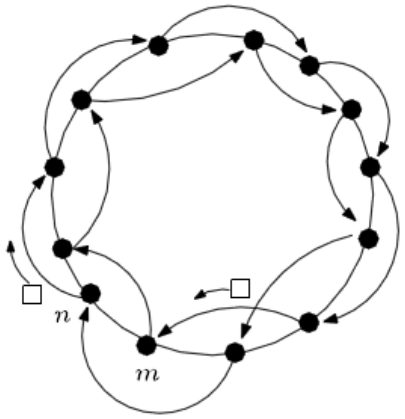

FIGURA 2.8: Exemplo de um *overlay* com ciclos [4]

Capítulo 3

Redes Peer-to-Peer baseadas em SIP

Neste capítulo descreve-se o protocolo de sinalização, Session Initiation Protocol (SIP), e as várias propostas que tem vindo a ser desenvolvidas no âmbito das redes P2PSIP, tendo sido escolhidas aquelas que mais se aproximam do tema deste trabalho.

3.1 *Session Initiation Protocol* (SIP)

O *Session Initiation Protocol* (SIP) é um protocolo de sinalização da camada de aplicação, normalizado pelo IETF. É utilizado para estabelecer, modificar ou terminar sessões entre um ou vários participantes [10]. Actualmente é um dos protocolos mais utilizados para a implementação de serviços de voz sobre IP (VoIP) na internet e também para o estabelecimento de conferências multimédia.

O SIP não especifica quais os tipos de sessões para os quais pode ser utilizado, sendo apenas responsável pela sua gestão. Para tal, o SIP oferece funcionalidades que permitem o registo, autenticação e localização de utilizadores. Permite também que os terminais de uma sessão negoceiem entre si os tipos de dados multimédia a utilizar, assim como permite a alteração de parâmetros de uma sessão já estabelecida enquanto esta decorre.

O protocolo foi desenvolvido utilizando um modelo de pedido e resposta idêntico ao utilizado no protocolo HTTP, onde cada mensagem enviada invoca uma determinada acção do lado do servidor, gerando este pelo menos uma mensagem de resposta.

Como identificador dos utilizadores, o SIP utiliza um endereço cujo formato é idêntico ao formato utilizado nos endereços de e-mail. Um endereço SIP começa por *sip:* seguido um identificador do utilizador (por exemplo um número de telefone) e por fim um identificador do domínio desse utilizador. O endereço *sip:48058@uminho.pt* mostra o formato tradicional de um endereço SIP.
A utilização do nome de um domínio no endereço SIP de um utilizador, facilita a sua localização, pois utilizando o Domain Name System (DNS) é possível obter o endereço de um servidor pertencente ao domínio do utilizador, que possuirá mais informação em como o localizar.

3.1.1 Componentes Principais

É designado por *User Agent*(UA) SIP qualquer dispositivo terminal que execute uma aplicação baseada em SIP, sendo que um UA pode ser um cliente ou um servidor. O cliente, *User Agent Client* (UAC) gera pedidos, para por exemplo estabelecer uma sessão com um outro UA, enquanto que o servidor *User Agent Server* (UAS) recebe e responde a pedidos. O UAC para além de gerar pedidos, é também responsável por processar as mensagens de resposta ao pedido efectuado.

Para além do UA, o protocolo SIP define outros componentes importantes, tais como:

- Servidor Proxy - É um servidor intermediário, que tem como tarefa principal encaminhar as mensagens para o destino, ou para um outro servidor mais próximo do destinatário da mensagem. Para além do encaminhamento, um Proxy pode ser utilizado para aplicar politicas, como por exemplo, verificar se o utilizador pode efectuar a chamada.

- Servidor de Redireccionamento - É um servidor (UAS) que recebe pedidos para a localização de um UA, recorrendo a mensagens de redireccionamento, com URI's alternativos que devem ser contactados para a localização do UA.

- Servidor de Registo - É um servidor responsável por aceitar mensagens de registo, do tipo *REGISTER*, guardando a informação que recebe no serviço de localização do domínio a que pertence.

- Servidor de Localização - O servidor de localização contêm uma lista na qual aos endereços (AoR)[1] são associados zero ou mais endereços de contacto. Este servidor é utilizado pelos servidores de redireccionamento ou proxies para obter informação sobre a localização dos UA.

- Back-to-back User Agent (B2BUA) - Um *back-to-back user agent* (B2BUA) é uma entidade lógica que actua como um cliente (UAC) e como um servidor (UAS) em simultâneo. Ao receber pedidos, este actua como um servidor (UAS) de modo a processar o pedido recebido assim como para gerar a mensagem de resposta. Actua como cliente (UAC) quando necessita de gerar pedidos. Contrariamente a um servidor proxy, um B2BUA participa activamente na troca de mensagens dos diálogos que estabeleceu.

Os componentes descritos anteriormente são componentes lógicos, sendo possível que uma aplicação combine um ou vários deste componentes. É habitual combinar os servidores de localização, de registo e de proxy numa única aplicação SIP.

A figura 3.1 representa um exemplo que ilustra as entidades SIP envolvidas no registo de um utilizador e na sua localização.
Neste exemplo, o utilizador **carol** envia um pedido de registo para o servidor de registo do seu domínio (**chicago.com**), informando-o de que se encontra disponível para contacto no endereço **cube2214a.chicago.com**. O servidor de registo valida o pedido e armazena a informação no servidor de localização. De seguida, o utilizador **bob** decide contactar o utilizador **carol**, para tal, e uma vez que o **bob** não possuí informação sobre o endereço no qual o utilizador **carol** está localizado, conhecendo apenas o seu endereço SIP (carol@chicago.com), necessita de enviar a sua mensagem do tipo **INVITE** para o servidor proxy do domínio do utilizador **carol**. O servidor proxy por sua vez, contacta o servidor de localização do seu domínio, de modo a obter um endereço no qual o destinatário pode ser contactado. Neste caso, recebe como resposta o endereço de contacto que o utilizador **carol** tinha registado anteriormente quando efectuou o seu registo. Uma vez possuindo um endereço no qual o destinatário pode ser contactado, o servidor de proxy, reenvia a mensagem do tipo **INVITE** para o endereço de contacto do destinatário.

[1] Address-of-Record (AoR) é um URI SIP que pode ser visto como um endereço público pelo qual um utilizador pode ser contactado.

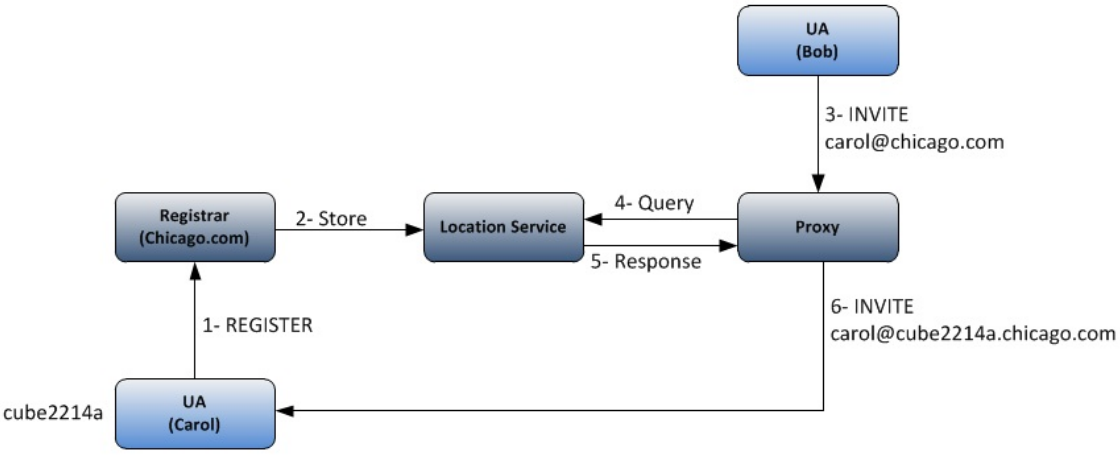

FIGURA 3.1: Exemplo do registo e localização de um utilizador

3.1.2 Estrutura das mensagens

O protocolo SIP especifica um conjunto de mensagens de pedido e de resposta, as quais são idênticas em termos sintácticos às mensagens do protocolo HTTP[11], sendo também elas enviadas como mensagens de texto. Para além das mensagens especificadas no protocolo, o SIP foi desenvolvido de forma a permitir que novos tipos de mensagens possam ser introduzidos, sendo por isso considerado um protocolo expansível.

A estrutura genérica das mensagens SIP está ilustrada na figura 3.2 que contêm um exemplo de uma mensagem SIP do tipo INVITE.
A primeira linha de uma mensagem SIP varia, no caso de a mensagem ser um pedido, a linha contem informação sobre o tipo de pedido. No caso de ser uma mensagem de resposta, a linha contêm entre outros, um valor numérico que representa o tipo de resposta. Depois da linha inicial, segue-se a zona do cabeçalho na qual cada linha contêm informações relativas a um parâmetro. O final do cabeçalho é indicado recorrendo a uma linha em branco, sendo que a seguir pode ser introduzido o corpo da mensagem. O corpo da mensagem é opcional, uma vez que nem todas as mensagens SIP o requerem. Segundo a especificação do protocolo [10], todas as linhas da mensagem SIP, devem terminar com a combinação de caracteres *carriage-return line-feed* (CRLF).

FIGURA 3.2: Formato de uma mensagem SIP

Pedidos

As mensagens SIP referentes a pedidos, diferenciam-se pelo facto de que a sua primeira linha contêm informação relativamente ao pedido a efectuar, estando definidos na especificação do protocolo os seguintes seis tipos de pedidos:

- REGISTER - Este tipo de pedido é utilizado para um utilizador registar informação relativamente à forma como pode ser contactado.
- INVITE, ACK e CANCEL - Estes tipos de pedidos são utilizados para estabelecer sessões entre utilizadores SIP.
- BYE - Utilizado para terminar sessões SIP.
- OPTIONS - Serve para obter informações sobre as funcionalidades disponíveis de um servidor.

Respostas

Uma mensagem de resposta SIP contêm na sua primeira linha um conjunto de parâmetros que representam o estado do pedido efectuado. O primeiro parâmetro é apenas informativo, contendo a versão do protocolo SIP em utilização, seguindo-se um código de estado composto por três dígitos que serve para identificar o estado do pedido efectuado. O último parâmetro é uma string que contêm o texto associado ao código de estado, podendo, em caso do pedido ser rejeitado, conter o motivo da rejeição.

Para o código de estado, a especificação do protocolo define seis tipos de classes, onde cada classe representa um conjunto de respostas do mesmo tipo. O primeiro digito do código representa a classe associada ao código, estando definidas na especificação as seguintes classes[10]:

- Provisório (1xx) - O pedido foi recebido, mas ainda não há uma resposta.
  Exemplo: 100 *Trying*
- Sucesso (2xx) - O pedido enviado foi recebido, processado e aceite.
  Exemplo: 200 OK.
- Redireccionamento (3xx) - O pedido tem de ser redireccionado.
  Exemplo: 302 *Moved Temporarily*
- Erro no Cliente (4xx) - O servidor não consegue processar a mensagem, esta pode ter erros sintácticos ou o servidor não pode satisfazer os requisitos.
  Exemplo: 400 *Bad Request*
- Erro no Servidor (5xx) - O servidor não consegue processar a mensagem, apesar de não ter detectado qualquer erro na mensagem.
  Exemplo: 503 *Service Unavailable*.
- Falha Geral (6xx) - O pedido não pode ser satisfeito em nenhum servidor.
  Exemplo: 600 *Busy Everywhere*.

3.2 A Secure Architecture for P2PSIP-based Communication Systems

Os autores deste artigo [12] estudaram os principais problemas de segurança que existem em redes P2PSIP, tendo proposto uma solução com vista a aumentar o nível de segurança da rede, solucionando alguns dos problemas de segurança mais comuns. A solução utiliza o Chord como algoritmo de DHT, no qual são posicionados alguns nós pré-configurados na rede, denominados de *Chord Secure Proxy* (CSP). Os CSPs são responsáveis por estabelecer ligações seguras entre nós da rede P2PSIP, permitindo a implementação de serviços de segurança como por exemplo, confidencialidade, integridade dos dados e disponibilidade.

3.2.1 Arquitectura Proposta

A arquitectura proposta neste artigo, pode ser dividida em três componentes principais, P2PSIP *peer*, Recursos, *Chord Secure Proxy* (CSP).

- P2PSIP *Peer* - É um elemento que participa na rede, podendo ser um dispositivo móvel, um PC ou qualquer outro dispositivo.
- Recurso - É o valor dos dados armazenado num nó (peer) específico. Os nos (*peers*) e os recursos são identificados através de identificadores com 128/160 bits.
- CSP - É um nó pré-configurado na rede, actuando como um proxy seguro, e de confiança.

Nesta arquitectura, um *peer* que pretenda localizar um recurso ou um outro *peer* no *overlay*, deve enviar uma mensagem P2PSIP para o CSP mais próximo do recurso/peer pretendido. Os autores denominam este primeiro salto entre o *peer* emissor e o CSP, como rede de origem (*source network*). Nesta solução o CSP actua como um proxy entre o *peer* emissor e o destinatário, devendo verificar se o *peer* de destino existe no *overlay*. Caso exista, deverá reencaminhar para este, de um modo seguro a mensagem P2PSIP enviada pelo emissor. Os autores denominam de rede de destino *destination network* a rede formada entre o CSP e os nós responsáveis por localizar o *peer* de destino. Uma vez

encontrado o destinatário, a ligação entre o emissor e o destinatário é feita directamente entre eles.

A figura 3.3 mostra um exemplo de um *overlay* P2PSIP que utiliza a arquitectura descrita no artigo [12].

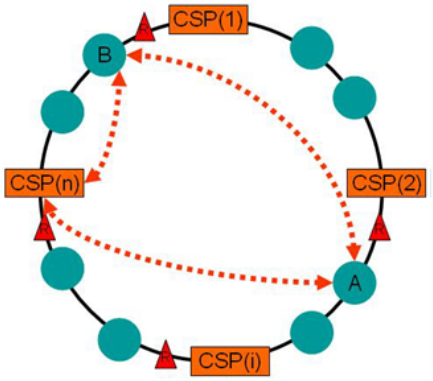

FIGURA 3.3: Exemplo da arquitectura proposta em [12]

O *Chord Secure Proxy* é o elemento mais importante nesta arquitectura, uma vez que é o responsável por estabelecer de forma segura ligações entre nós do *overlay*. Um CSP é composto por quatro componentes fundamentais, *Source inter-working*, Gestor de Políticas, Criptografia, e *Destination inter-working*.

A figura 3.4 mostra os componentes que compõe um Chord Secure Proxy.

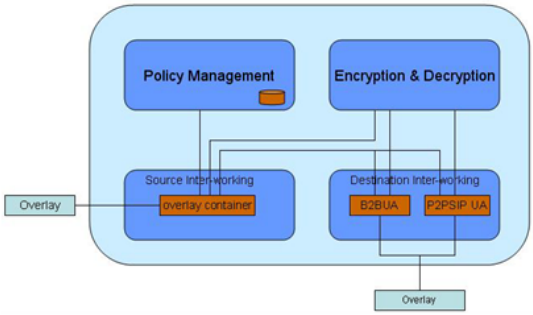

FIGURA 3.4: Arquitectura de um *Chord Secure Proxy* [12]

Source Inter-working

Este componente é responsável por receber as mensagens P2PSIP dos *peers* (remetentes), e consoante os critérios de segurança especificados no pedido, determina qual a estratégia a aplicar para a entrega do pedido ao destinatário.

Gestão de Políticas

Este componente é responsável por decidir os mecanismos de segurança a utilizar, possuindo uma base de dados, na qual estão especificados as políticas existentes e os respectivos mecanismos de segurança a utilizar por cada política. Foi definida uma nova extensão para o cabeçalho das mensagens P2PSIP, de modo a incluir o nível de segurança requerido. A nova extensão ao cabeçalho tem um novo campo *Secure* seguindo-se o respectivo valor, sugerindo os autores que o sistema suporte pelo os seguintes níveis de segurança, nenhum, critico e anonimato.

- Nenhum - Este tipo de política é utilizado quando o utilizador não requer que haja nenhum nível de segurança para a mensagem.
- Critico - Este tipo de política significa que a utilização de mecanismos de segurança é critico para as mensagens, devendo o CSP assegurar a confidencialidade e integridade dos dados, assim como esconder a informação sobre o emissor da mensagem, até que o nó de destino seja encontrado e seja devidamente autenticado. O CSP deverá rejeitar o pedido caso não possa garantir os níveis de segurança necessários neste tipo de politica.
- Anonimato - Este tipo de política requer que o CSP esconda de todos os nós a informação privada relativa ao emissor da mensagem. O CSP devera rejeitar o pedido caso não suporte este tipo de política.

Destination Inter-working

A rede de destino é definida como sendo uma rede lógica que representa as ligações efectuadas a partir de um CSP especifico até ao *peer* que se pretende localizar. Para localizar

um *peer* no *overlay*, um CSP envia uma mensagem do tipo *HelloRequest* para alguns dos seus sucessores que se encontram mais próximos (no sentido contrário aos dos ponteiros do relógio) do destinatário. Cada sucessor, ao receber uma mensagem *HelloRequest*, deve-a reenviar para outro *peer*, utilizando o mecanismo tradicional de localização de recursos do Chord, até que o destinatário seja encontrado.

A mensagem de localização *HelloRequest* ao ser enviada pelo CSP para alguns dos seus sucessores, faz com que o destinatário possa receber a mesma mensagem repetidamente, proveniente de diferentes *peers*. Devendo o destinatário escolher aleatoriamente uma das mensagens recebidas, e enviar para o CSP respectivo uma mensagem do tipo *HelloResponse*.

A utilização de um mecanismo no qual as mensagens de localização são enviadas por um CSP para vários *peers* do *overlay* em simultâneo, gera mais tráfego de dados na rede quando comparado com o mecanismo de localização de recursos tradicional de uma rede de *overlay* baseada no Chord. Contudo, segundo os autores, este mecanismo torna o sistema mais tolerante a falhas, assim como também minimiza o impacto que *peers* maliciosos poderiam ter na rede ao descararem ou adulterarem as mensagens que deveriam reencaminhar.

Os parâmetros que compõe a estrutura das mensagens *HelloRequest* e *HelloResponse* que foram definidas neste artigo são os seguintes:

- Type Of Service (TOS) - Este parâmetro especifica o tipo da mensagem, por exemplo para uma mensagem do tipo *HelloRequest* o valor deste parâmetro poderia ser "8"e "0"para uma mensagem do tipo *HelloResponse*.
- Code - Este parâmetro serve para identificar a ocorrência de erros (valor entre 1 e 15), como por exemplo quando o destinatário de uma mensagem esta inacessível.
- Checksum - Este parâmetro é utilizado para verificar se a mensagem possuí algum erro.
- Call-ID - Parâmetro de 32 bits que serve para identificar uma mensagem.
- CSP Identifier - Identificador P2PSIP com 128/160 bits que identifica o CSP.
- CSP Public IP Address - Endereço IP público do CSP.

- CSP Port - Porta que deverá ser utilizada para estabelecer ligações com o CSP.
- Destination *Peer* Identifier - Identificador P2PSIP de 128/160 bits que identifica o *peer* de destino da mensagem

A mensagem do tipo *HelloResponse*, possui mais dois parâmetros que contem o endereço IP e porta do destinatário:

- Destination Public IP Address - Endereço IP do *peer* de destino da mensagem.
- Destination Port - Porta na qual o destinatário pode ser contactado.

As figuras 3.5(a) e 3.5(b) mostram a estrutura das mensagens *HelloRequest* e *HelloResponse* que foram definidas na solução defendida neste artigo.

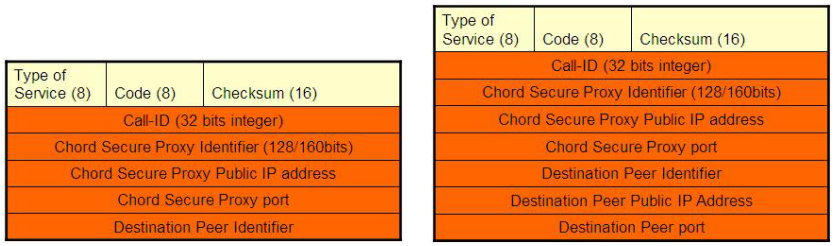

FIGURA 3.5: Estrutura das mensagens *HelloRequest* (a) e *HelloResponse* (b)

Segurança

As ligações tanto na rede de origem (*source network*) como na rede de destino (*destination network*) devem ser ligações seguras, sugerindo os autores deste artigo, a utilização de um mecanismo baseado em PKI (Public Key Infrastructure). Cada elemento do *overlay* deve possuir um par de chaves, uma pública e uma privada. A chave pública, é uma chave que deve ser do conhecimento público, podendo ser distribuída através de um certificado digital, e é utilizada para cifrar dados, ou para validar assinaturas digitais. Em contrapartida, a chave privada nunca deve ser partilhada, devendo esta ser guardada e utilizada apenas pelo seu proprietário.

Utilizando criptografia de chave pública, um nó da rede que deseje enviar uma mensagem cifrada para um outro nó, deve cifrar a mensagem com a chave pública do destinatário da mensagem. Desta forma é possível garantir que apenas o destinatário pode decifrar a mensagem, uma vez que para decifrar uma mensagem cifrada com a sua chave pública, é necessário utilizar a chave privada, a qual apenas o próprio a deve possuir.
Para além de ser utilizada para cifrar/decifrar mensagens, a criptografia de chave pública, permite ainda gerar assinaturas digitais a partir de documentos, ou blocos de dados, que é útil verificar a integridade dos dados. Para se obter uma assinatura digital de um documento, ou de um bloco de dados, um nó deve aplicar uma função de *hash* (como por exemplo o MD5 ou o SHA) sobre os dados, cifrando de seguida o *hash* obtido com a sua chave privada, o resultado desta operação é a assinatura digital do documento. O receptor ao receber um documento assinado digitalmente, pode verificar a sua integridade, para tal, tem de calcular o *hash* do documento recebido, utilizando o mesmo algoritmo de *hashing* utilizado pelo emissor. De seguida, deve decifrar utilizando a chave pública do emissor, o *hash* do documento contido na assinatura digital enviada, e que foi cifrado pelo emissor com a sua chave privada. A integridade dos dados é assegurada se o *hash* calculado no destinatário for igual ao *hash* obtido após ter sido decifrada a assinatura digital.

Encaminhamento de Mensagens (Routing)

Como estratégia de routing das mensagens na rede de *overlay*, os autores deste artigo, sugerem a utilização de um mecanismo semi-recursivo. Neste mecanismo, cada *peer* intermediário reenvia a mensagem para o *peer* mais próximo do destino que conhece. Uma vez entregue, a mensagem de resposta é enviada directamente do destinatário para o emissor. Segundo os autores, esta técnica, reduz substancialmente o número de mensagens que circulam pela rede, quando comparado com outras estratégias de routing, como por exemplo, routing iterativo ou recursivo.

A figura 3.6 mostra um exemplo da utilização de uma estratégia de routing semi-recursivo, no qual o nó **S** pretende localizar o nó **C**, para tal envia uma mensagem para o nó **A**, sendo que este uma vez que não conhece o nó de destino, reenvia a mensagem para o nó mais próximo do destino que conhece, neste caso o nó **B**. Este nó como conhece o destinatário da mensagem, envia a mensagem directamente para o destinatário, sendo

que o destinatário envia depois a mensagem de resposta directamente para o emissor, ou seja para o nó **S**.

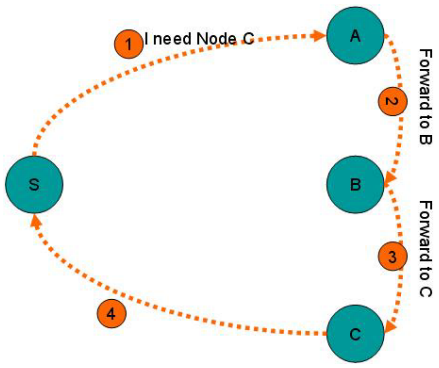

FIGURA 3.6: Estratégia de routing semi-recursivo [12]

3.2.2 Testes e Resultados

A solução proposta neste artigo [12], foi avaliada pelos seus autores tendo sido efectuadas medições para comparar o desempenho da solução proposta, com o desempenho de uma solução P2P puramente baseada no algoritmo DHT Chord.

Número de Saltos

Num *overlay* P2P baseado em Chord, o número de saltos para se localizar um *peer* ou um recurso é em média $\frac{1}{2}logN$. Na solução proposta neste artigo [12], o número de saltos para se localizar um *peer* na rede de destino (*destination network*) é $\frac{1}{2}log(N/S)$, onde N é o número de *peers* no *overlay*, e S é o número de CSP's uniformemente distribuídos no *overlay*. Tendo em conta o salto necessário na rede de origem (*source network)* para a mensagem atingir o CSP, o número médio de saltos total, é de $\frac{1}{2}log(N/S) + 1$.
A figura 3.7 mostra que utilizando a solução proposta, o número de saltos necessários para se localizar um *peer* ou um recurso, é inferior comparativamente com um *overlay* P2P puramente baseado no Chord.

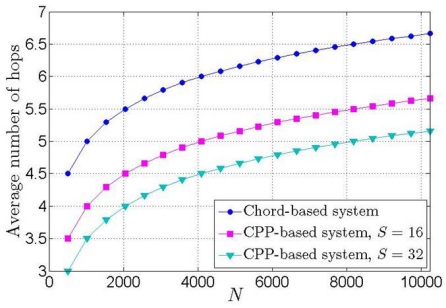

FIGURA 3.7: Comparação do número de saltos [12]

Atraso médio

O atraso das mensagens foi também medido, tendo sido escolhido o *peer* 586 para enviar mensagens para 100 *peers* escolhidos aleatoriamente. Os resultados deste teste, mostram que a solução baseada no Chord, teve um atraso médio de 16ms, enquanto que a solução proposta teve um desempenho inferior, tendo em média um atraso de 62ms. Segundo os autores, isto deve-se em parte, ao facto de o *peer* de destino ter de esperar cerca de 30us para receber as múltiplas mensagens, antes de escolher uma delas e responder. Para além do atraso de 30us, o facto de o CSP enviar a mensagem para vários *peers* em simultâneo, aumenta o tráfego na rede, o que por sua vez pode aumentar o atraso das mensagens.

Níveis de confiança

Os seus autores da solução proposta no artigo [12], decidiram medir o nível de confiança no encaminhamento das mensagens, para tal, definiram T_{medio} como sendo o valor médio de confiança para cada *peer* no *overlay* P2PSIP, podendo o valor variar de 0 a 1. Uma vez que num sistema baseado em Chord, o número de saltos é em média $\frac{log(N)}{2}$, o valor de confiança médio é de $T_{medio}^{log(N)/2}$. Relativamente à solução proposta, tendo em conta que a ligação na rede de origem (*source network*) entre o *peer* e o CSP é considerada segura, o número de saltos médio na rede de destino (*destination network*) é $\frac{1}{2}log(N/S)$, de onde se obtem que o valor de confiança médio é $T_{medio}^{log(N/S)/2}$.

A figura 3.8 mostra que a solução proposta oferece um maior nível de confiança, quando comparado com uma solução puramente baseada no Chord.

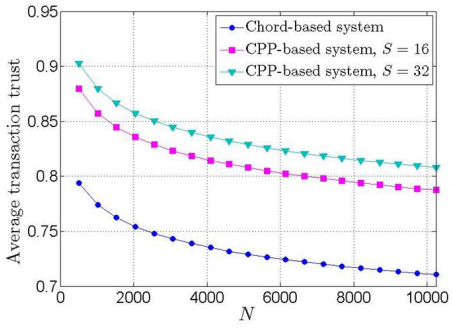

FIGURA 3.8: Níveis de confiança num sistema baseado em CSPs [12]

3.3 *Distributed Session Initiation Protocol* (dSIP)

O *Distributed Session Initiation Protocol* (dSIP) é uma especificação preliminar (*draft*)[13] para um protocolo P2PSIP proposto por alguns autores da *SIPeerior Technologies* e da CISCO. É visto como uma evolução do protocolo SoSIMPLE [14] o qual foi especificado por alguns dos mesmos autores do dSIP. Este protocolo é totalmente baseado em SIP [10], sendo este utilizado para efectuar toda a gestão do *overlay* P2P, como por exemplo para o registo e localização de recursos ou *peers*.

O facto de o SIP ser um protocolo extensível, permitiu a criação de novos cabeçalhos SIP, necessários para o transporte de informação relevante para a gestão do *overlay* P2P. Segundo os autores do dSIP [13], a utilização das mensagens SIP tradicionais, permite a utilização no dSIP, de mecanismos habitualmente utilizados no SIP (STUN [15], TURN [16] e ICE [17]) para ultrapassar os problemas causados por NAT's e firewalls.

3.3.1 Estrutura do *Overlay* P2P

O protocolo dSIP foi desenvolvido de modo a ser modular, podendo ser utilizado com múltiplos algoritmos de DHT, sendo necessário suportar pelo menos o Chord.
Relativamente à estrutura do *overlay* P2P, no dSIP os *peers* são organizados no *overlay* de acordo com o algoritmo DHT em utilização. São atribuídos identificadores únicos (Peer-ID e Resource-ID) aos *peers* e aos recursos armazenados no *overlay*, sendo que ambos os identificadores devem pertencer ao mesmo espaço de endereçamento. O calculo dos identificadores únicos pode ser obtido recorrendo a diversos algoritmos, contudo é necessário

que seja utilizado o mesmo algoritmo por todos os *peers* do *overlay*. Na implementação base, que utiliza o Chord como DHT o algoritmo utilizado é o SHA-1 com 160 bits.
A forma como os identificadores dos *peers* são obtidos pode variar consoante o nível de segurança que é pretendido. Por exemplo, pode ser obtido aplicando o algoritmo SHA-1 sobre a combinação endereço IP e porta do *peer*, ou recorrendo a uma entidade certificadora responsável por emitir os identificadores.
Através do Peer-ID de um *peer* o algoritmo DHT em utilização, determina a localização do *peer* no *overlay*, assim como os identificadores dos recursos pelos quais o *peer* é responsável. A forma como um *peer* é posicionado no *overlay*, e os recursos pelos quais é responsável depende do algoritmo DHT que o *overlay* utiliza.
O *Resource-ID*, identificador utilizado para identificar recursos armazenados no *overlay*, pode ser obtido através da aplicação de uma função de *hash* sobre o nome ou sobre um conjunto de palavras que descrevem o recurso. O recurso é armazenado no *peer* que tiver o *Peer-ID* mais próximo do seu *Resource-ID*.
Devido à constante entrada e saída de *peers* numa rede P2P, a informação relativa aos recursos armazenados vai sendo trocada entre estes de forma a que os recursos estejam sempre acessíveis. São implementados mecanismos de redundância de informação para evitar que haja perda de dados, quando por exemplo um *peer* falha antes de transmitir para outro *peer* a informação relativa aos recursos pelos quais era responsável.

A forma exacta como a localização de recursos é feita varia consoante o algoritmo DHT em utilização. De uma forma genérica, para a localização de um determinado recurso, um *peer* deve consultar a sua tabela de encaminhamento, e enviar a mensagem para o *peer* que possuí o *Peer-ID* mais próximo do *Resource-ID* do recurso pretendido. Devendo esse *peer*, dependendo do mecanismo de encaminhamento em utilização, reenviar a mensagem para o *peer* mais próximo do recurso que conhece, ou enviar uma mensagem de resposta contendo a informação relativa ao *peer* mais próximo do recurso que este conhece.

3.3.2 Mensagens SIP

A utilização de mensagens SIP para a implementação do protocolo P2PSIP foi vista como sendo a melhor solução pelos autores do protocolo dSIP [13]. Por definição uma aplicação

P2PSIP requer a implementação da *stack* SIP, e uma vez que grande parte dos mecanismos que um protocolo P2PSIP necessita, já se encontram especificados no SIP, como por exemplo mecanismos para armazenar, recuperar e consultar a localização de recursos, a utilização do SIP acaba por ser uma escolha natural. Evitando a necessidade de se especificar um novo protocolo de raíz.
O facto de o SIP ser um protocolo por natureza expansível, permitiu que para a implementação do protocolo dSIP, as mensagens tradicionais SIP fossem mantidas. Tendo sido adicionados os seguintes cabeçalhos de modo a permitir a implementação de mensagens para a gestão do *overlay*:

- DHT-PeerID - Cabeçalho obrigatório, utilizado para identificar o emissor de uma mensagem, o *overlay* e os seus parâmetros.
- DHT-Link - Cabeçalho utilizado para um *peer* enviar informação sobre outros *peers* da sua tabela de encaminhamento.

Para além da especificação destes dois novos cabeçalhos SIP, as mensagens do protocolo dSIP caracterizam-se pela presença obrigatória de outros dois cabeçalhos SIP, cabeçalho 'Require' e 'Supported'. Para uma mensagem SIP ser identificada como uma mensagem dSIP, estes dois cabeçalhos devem estar presentes na mensagem e ter o valor 'dht', indicando desta forma tratar-se de uma mensagem dSIP.

Os autores do dSIP, classificam os tipos de mensagens SIP utilizados em duas classes. A primeira classe de mensagens é utilizada para a manutenção da DHT, como por exemplo mensagens enviadas quando um *peer* entra ou sai do *overlay*, ou para transferir informação entre *peers*. A segunda classe de mensagens, é utilizada para implementar funções que são típicas no SIP, como o registo de utilizadores, ou o estabelecimento de uma sessão entre utilizadores.

A figura 3.9 ilustra dois exemplos de mensagens SIP utilizadas pelo protocolo dSIP. Em ambos os exemplos é possível verificar que a estrutura e a sintaxe das mensagens não foram alteradas, sendo visíveis os novos cabeçalhos obrigatórios introduzidos pelo dSIP.

```
REGISTER sip:10.7.8.129 SIP/2.0
To: <sip:peer@10.4.1.2;peer-ID=463ac413c4>
From: <sip:peer@10.4.1.2;peer-ID=463ac413c4>
Contact: <sip:peer@10.4.1.2;peer-ID=463ac413c4>
Expires: 600
DHT-PeerID: <sip:peer@10.4.1.2;peer-ID=463ac413c4>;algorithm=sha1;
            dht=dhtalg1.0;overlay=chat;expires=600
Require: dht
Supported: dht
```

(a)

```
SIP/2.0 302 Moved Temporarily
To: <sip:peer@10.4.1.2;peerId=463ac413c4>
From: <sip:peer@10.4.1.2;peerId=463ac413c4>
Contact: <sip:peer@10.3.1.7;peerId=47e46fa2cd>
Expires: 600
DHT-PeerID: <sip:@10.7.8.129;peerId=084d299ff2>;algorithm=sha1;
            dht-param=dhtalg1.0;overlay=chat;expires=600
Require: dht
Supported: dht
```

(b)

FIGURA 3.9: Exemplo de mensagens SIP do protocolo dSIP

Encaminhamento de Mensagens

Relativamente ao encaminhamento de mensagens, quando um *peer* deseja localizar um outro *peer* ou um recurso no *overlay*, este consulta a sua tabela de encaminhamento, e escolhe na sua tabela, o *peer* com o identificador *Peer-ID* mais próximo (podendo ser utilizadas outras métricas, dependendo do algoritmo DHT em uso) do identificador do *peer*/recurso que pretende localizar. É enviado para este *peer* uma mensagem SIP, devendo este *peer*, no caso de não ser o *peer* pretendido, efectuar o encaminhamento da mensagem para outro *peer* de modo a que o *peer* ou recurso pretendido seja localizado. O mecanismo utilizado no *overlay* para o encaminhamento de mensagens não é definido pelo protocolo, podendo ser utilizados qualquer um dos seguintes mecanismos:

- Iterativo - Se o *peer* que recebe a mensagem não for o *peer* pretendido, este deve responder ao emissor da mensagem, com uma mensagem do tipo *302 Redirect*, na qual indica o *peer* que conhece que possuí o identificador mais próximo do identificador do *peer*/recurso pretendido. Cabendo ao emissor, enviar uma nova mensagem para o *peer* indicado na mensagem do tipo *302 Redirect* recebida, sendo que este processo repete-se até que o *peer*/recurso pretendido seja encontrado.

- Recursivo - Neste tipo de mecanismo, se o *peer* que recebe a mensagem não for o *peer* pretendido, este deve consultar a sua tabela de encaminhamento e escolher

o *peer* que possuí o identificador mais próximo do identificador do *peer*/recurso pretendido, reenviando a mensagem para este *peer*. O processo é repetido até que o *peer*/recurso pretendido seja encontrado, sendo que a mensagem de resposta, deve seguir o mesmo caminho seguido pela mensagem de localização enviada.

- Semi-Recursivo - Este mecanismo, é idêntico ao mecanismo recursivo, sendo a única diferença a forma como a mensagem de resposta é enviada. Neste mecanismo de encaminhamento a mensagem de resposta é enviada directamente para o *peer* que iniciou o processo de localização.

3.4 Peer-to-Peer Protocol (P2PP)

O *Peer-to-Peer Protocol* (P2PP) é uma outra especificação preliminar do IETF (*Internet Draft*) [18] para um protocolo P2PSIP, tendo sido desenvolvido com o objectivo de possibilitar a criação de um *overlay* P2P com capacidade para armazenar e localizar recursos, utilizando algoritmos P2P com ou sem estrutura. Os nós que compõe o *overlay*, neste protocolo podem ser denominados de duas formas, *peers* ou clientes. Um *peer* é um nó que participa no *overlay* disponibilizando os seus recursos com o *overlay*, para por exemplo, armazenar informação, ou encaminhar mensagens no *overlay*. Um cliente, por sua vez é um nó que não participa no *overlay*, e que por isso não é utilizado para armazenar informação nem para o encaminhamento de mensagens do *overlay*, sendo que um cliente pode aceder aos serviços disponibilizados pelo *overlay*, comunicando directamente com um ou vários *peers* pertencentes ao *overlay*.
Este protocolo não especifica quais os nós que devem actuar como *peers* ou como clientes, cabendo essa decisão ao operador do *overlay*.

Neste protocolo, tanto os *peers* como os clientes são identificados através de um identificador único, de tamanho fixo, devendo estes pertencer ao mesmo espaço de endereçamento. Os identificadores são obtidos através de um servidor, denominado por E&A *(Enrollment and Authentication)*, responsável por gerir os registos e a autenticação dos *peers*, sendo que um *peer* do *overlay* pode actuar como E&A. Segundo os autores [18] o recurso a servidores, para por exemplo, a autenticação dos utilizadores, ou para obter endereços de *peers* pertencentes ao *overlay* (*bootstrap servers*) pode levar a que a natureza distribuída

do *overlay* seja posta em causa. Contudo eles afirmam que apesar de a utilização deste tipo de servidores não ser obrigatória, é benéfica, pois evita alguns problemas a nível de segurança.

3.4.1 Arquitectura do Protocolo

A arquitectura do protocolo P2PP divide-se em três camadas, camada de aplicação, camada de gestão do *overlay* e camada de transporte.
A camada de aplicação (*Application Layer*), fornece às aplicações um conjunto de *Application Programming Interfaces* (APIs) que estas podem utilizar para aceder aos serviços disponibilizados pelo *overlay*.
A camada de gestão do *overlay* (*Overlay Layer*) é a camada principal, pois possuí os mecanismos para o encaminhamento de mensagens, manutenção do *overlay*, armazenamento, replicação e NAT transversal. Relativamente ao encaminhamento de mensagens no *overlay*, existem três formas diferentes de encaminhar as mensagens no *overlay*, de forma recursiva, iterativa ou em paralelo. A manutenção do *overlay*, consiste em tentar garantir o correcto funcionamento do mecanismo de encaminhamento de mensagens, assim como garantir conectividade dos *peers* quando a rede esta sobrecarregada. O mecanismo de replicação é responsável por garantir a disponibilidade dos recursos armazenados no *overlay*.
A camada de transporte é responsável pela transmissão das mensagens, podendo estas ser enviadas através de qualquer protocolo de transporte, sendo contudo recomendado a utilização de protocolos de transporte fiáveis como por exemplo, o TCP ou TLS. Contudo, se o protocolo utilizado não for fiável, como por exemplo o UDP, ou DTLS, esta camada fornece um mecanismo baseado em *Acknowledges* (ACKs) de modo a tornar o transporte das mensagens fiável.

A figura 3.10 mostra a arquitectura em camadas do protocolo P2PP.

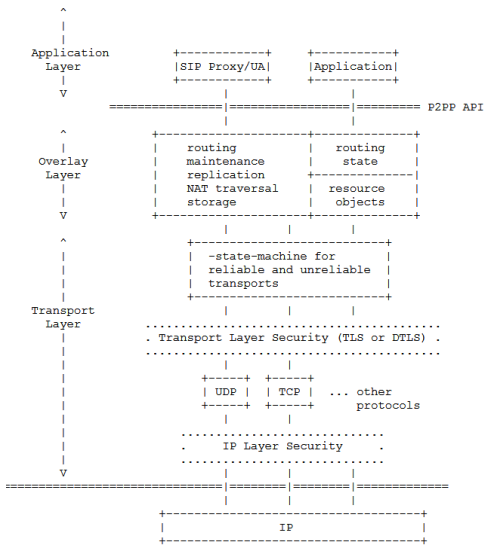

FIGURA 3.10: Arquitectura de um *peer* P2PP [18]

3.4.2 Formato das Mensagens

As mensagens do protocolo P2PP são compostas por um cabeçalho fixo, comum a todas as mensagens, seguindo-se por uma sequência de objectos do tipo *type-length-value* (TLV). Um objecto TLV é composto por um identificador do objecto, um parâmetro que especifica o tamanho do objecto, e por fim, um parâmetro com o valor do objecto. Um exemplo de um objecto TLV é o objecto *Peer-Info*, que contêm informações relativas a um determinado *peer*.

A figura 3.11 mostra a estrutura do cabeçalho das mensagens P2PP. Descrição dos

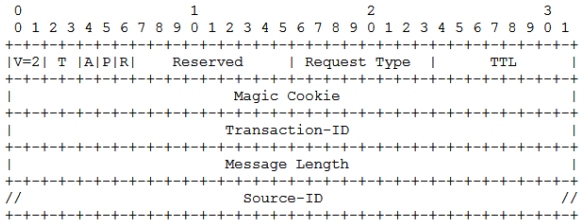

FIGURA 3.11: Formato do cabeçalho das mensagens P2PP [18]

parâmetros do cabeçalho:

- Versão - Parâmetro de 2 bits que especifica a versão do protocolo utilizada

- T - Parâmetro de 2 bits, que indica o tipo de mensagem, podendo esta ser:
  - 00 - Um pedido
  - 01 - Uma indicação, por exemplo, para informar um *peer* de alterações na tabela de encaminhamento.
  - 10 - Uma resposta
  - 11 - Uma mensagem de resposta, com um ACK
- A - Este parâmetro é utilizado se o protocolo de transporte não for fiável, se o parâmetro tiver o valor "1"indica que a mensagem é uma mensagem de confirmação (ACK).
- P - Se este parâmetro tiver o valor 1, indica que a mensagem foi enviada por um *peer*, caso contrário foi enviada por um cliente.
- R - Parâmetro que indica o método de encaminhamento utilizado, se tiver o valor "1"indica que a mensagem foi enviada um método recursivo, caso contrário foi enviada de forma iterativa. Este parâmetro não é utilizado em mensagens de resposta ou de indicação.
- Request Type - Parâmetro que identifica o tipo de pedido, ou o tipo de indicação.
- TTL - Parâmetro de 8 bits que indica o número máximo de *peers* pelos quais a mensagem pode passar.
- Magic Cookie - Parâmetro de 32 bits com um valor fixo (0x596ABF0D), que serve para diferenciar as mensagens do protocolo P2PP de mensagens de outros protocolos como o STUN.
- Transaction-ID - Parâmetro de 32 bits que serve para identificar inequivocamente a mensagem.
- Message Length - Parâmetro de 32 bits que indica o tamanho do corpo da mensagem.
- Source-ID - Parâmetro que contêm o Peer-ID do *peer* ou cliente que enviou a mensagem.

3.5 REsource LOcation And Discovery (RELOAD)

O REsource LOcation And Discovery (RELOAD) é a mais recente proposta do IETF [19] (*Internet Draft*) para a criação de um protocolo P2PSIP. O grupo de trabalho responsável por esta especificação é composto por alguns dos autores de propostas P2PSIP anteriores, como o dSIP [13], ASP [20] e P2PP [18]. Contrariamente a outras propostas, como o dSIP, este protocolo não utiliza mensagens SIP na gestão do *overlay*, utilizando um novo protocolo. O protocolo utilizado na gestão do *overlay*, é um protocolo binário, desenvolvido para ser mais leve, permitindo segundo os seus autores [19] melhorar o desempenho global do protocolo, uma vez que o tamanho das mensagens é reduzido quando comparado com mensagens de texto como as SIP, o que reduz o tráfego no *overlay*.

3.5.1 Arquitectura

A arquitectura do protocolo RELOAD divide-se em três camadas distintas, camada de aplicação, camada *peer-to-peer* ou RELOAD, e a camada de transporte.
Apesar de o RELOAD ter sido desenvolvido com o objectivo de permitir a utilização de aplicações SIP num contexto P2P, a camada de aplicação permite o uso de diferentes protocolos aplicacionais, como por exemplo o SIP ou o XMPP[2].

Para além de o RELOAD permitir a utilização de diferentes protocolos ao nível da aplicação, a camada *peer-to-peer*, responsável pela criação e gestão do *overlay*, permite que o *overlay* possa utilizar qualquer algoritmo DHT. Contudo, como em outras propostas P2PSIP, a implementação do algoritmo Chord é obrigatória, pois este é o algoritmo DHT utilizado por defeito. Esta camada é composta pelos seguintes componentes:

- *Message Transport* - Responsável pelo envio e recepção de mensagens.
- *Storage* - Responsável pelo armazenamento de informação do *overlay*
- *Topology Plugin* - Responsável por implementar o algoritmo DHT a utilizar no *overlay*.

[2]O Extensible Messaging and Presence Protocol(XMPP) é um protocolo para comunicações em tempo real, utilizado em diversas aplicações na internet.

- *Forwarding and Link Management* - 'Armazena e implementa a tabela de encaminhamento, fornecendo serviços de encaminhamento de pacotes entre os nós.'

A figura 3.12 mostra a arquitectura por camadas adoptada pelo protocolo RELOAD. Analisando a figura, é possível verificar que uma aplicação VoIP por exemplo, que utilize SIP, pode utilizar o RELOAD para localizar outros *peers* no *overlay*, podendo também (opcionalmente) utilizar o *overlay* para estabelecer ligações entre os *peers* que se encontram protegidos por firewalls ou NATs, beneficiando dos mecanismos que o RELOAD implementa para ultrapassar esses problemas, nomeadamente o ICE [17], STUN [15] e o TURN [16].

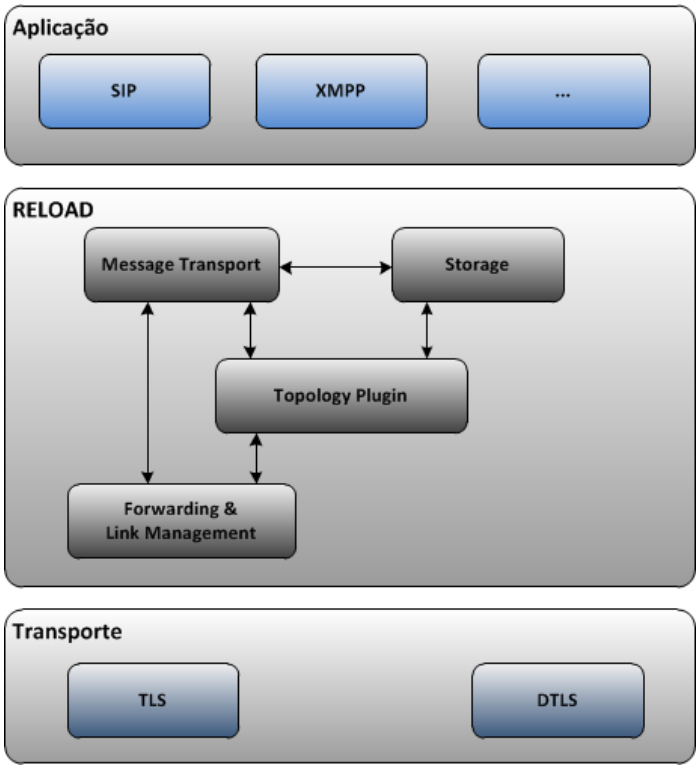

Figura 3.12: Arquitectura do protocolo RELOAD

Capítulo 4

Uma proposta de arquitectura P2PSIP hierárquica

A implementação de um protocolo *peer-to-peer* totalmente baseado em SIP (P2PSIP) era um dos objectivos principais deste trabalho. Das várias propostas para protocolos P2PSIP descritas no capitulo 3, foi escolhido o dSIP [13]. A escolha do dSIP deve-se ao facto de esta ser uma solução totalmente baseada em SIP, sendo por isso, das propostas apresentadas, aquela que melhor se adequa aos objectivos deste trabalho.

O facto de o dSIP ser totalmente baseado em SIP é uma vantagem pois o SIP é um protocolo simples, normalizado, cujo funcionamento e desempenho é já bastante conhecido, sendo por isso um protocolo maduro. Para além disso, é um protocolo suportado por um grande número de dispositivos, o que pode permitir a reutilização da *stacks* SIP implementadas, minimizando o número de protocolos que um *peer* necessita de suportar. Um outro aspecto positivo da utilização SIP, é o facto de que habitualmente o seu tráfego não é bloqueado nas *firewalls*. Por estes motivos, a utilização do protocolo SIP, torna mais fácil a implementação de novos serviços assim como pode permitir a interoperabilidade entre diferentes serviços.

Relativamente ao *overlay peer-to-peer*, os *peers* que o formam posicionam-se e estabelecem ligações com outros *peers* de acordo com o algoritmo DHT em utilização no *overlay*.

Uma vez formado o *overlay*, este é utilizado para armazenar e localizar recursos, oferecendo de uma forma distribuída, os serviços que habitualmente um Servidor de Registo e Localização oferece numa arquitectura SIP tradicional.

Os recursos armazenados pelo *overlay* são compostos por um par chave/valor, onde a chave é o identificador do recurso. No contexto deste trabalho, a chave de um recurso é o endereço SIP do utilizador, e o valor é o endereço IP e porta de contacto do utilizador. O identificador do recurso (o *hash* da chave) é utilizado pelo algoritmo DHT do *overlay* para definir a localização na qual o recurso deve ser armazenado no *overlay*.

4.1 Hierarquia com dois níveis

Num *overlay* P2P tradicional todos os *peers* possuem a mesma importância, devendo participar activamente nas tarefas de encaminhamento, armazenamento e localização de recursos disponibilizadas pelo *overlay*. Contudo, há casos em que atribuir a todos os *peers* a mesma importância pode não ser o mais desejável, quer por questões de segurança, quer por outros motivos como por exemplo o desempenho do *overlay*.

Neste trabalho, abordamos o segundo ponto, no qual pretendemos estudar qual o impacto que tem no desempenho do *overlay* a existência de *peers* com menores capacidades. Por *peers* com menores capacidades, designamos aqueles que possuem algumas das seguintes características:

- Ligação à rede de menor qualidade (tráfego limitado, menor largura de banda, ou ligação com grande probabilidade de perda de pacotes)
- Pouco tempo de permanência no *overlay*, podendo gerar muito tráfego de manutenção do *overlay* (transferência de recursos, actualizações das tabelas..)
- Recursos limitados em termos de hardware (CPU, memoria, bateria..)

Os *smartphones* são um bom exemplo de um dispositivo móvel capaz de participar num *overlay* P2P mas cuja participação pode ter algum impacto tanto no *overlay* como no próprio dispositivo, devido às possiveis limitações a nível de conectividade e também às

características do dispositivo, nomeadamente a bateria.

A existência no *overlay*, de *peers* com algumas das características enumeradas como por exemplo, com pouca largura de banda, ou alta probabilidade de perda de pacotes, pode influenciar o desempenho do *overlay*, afectando o tempo para a localização de recursos.

Para minimizar estes problemas, implementamos um *overlay* P2P de dois níveis, com um funcionamento idêntico aos *overlays* baseados em *super-peers* descritos no capitulo 2.1.
O nível mais baixo da hierarquia, é composto por nós com menores capacidades, designados por clientes. Um cliente não forma nem participa activamente em qualquer *overlay* P2P, podendo contudo, utilizar os serviços disponibilizados pelo *overlay*. Para que possa aceder aos serviços disponibilizados pelo *overlay*, um cliente tem de estabelecer uma ligação com pelo menos um *peer* do *overlay*, enviando para este os seus pedidos para armazenar ou localizar recursos.
Desta forma, um cliente actua sempre para seu próprio beneficio, não recebendo mensagens que não são para si, nem tem a responsabilidade de armazenar dados do *overlay*, utilizando um *peer* pertencente ao *overlay* como intermediário para aceder aos serviços que este disponibiliza.

O nível superior da hierarquia, é composto pelos nós(*peers*) que formam realmente o *overlay* DHT. Estes actuam como *peers* normais, armazenando recursos e encaminhando mensagens entre si. A única alteração que é necessário efectuar nos *peers*, é adicionar suporte para clientes. Isto é, permitir que os *peers* possam receber do exterior do *overlay*, mensagens para localizar ou armazenar recursos no *overlay*. Sempre que é recebido de um cliente um pedido para, por exemplo armazenar um recurso no *overlay*, este armazena o recurso no *overlay*, como se este fosse seu, não havendo necessidade de efectuar alterações nos algoritmos DHT. A figura 4.1 mostra um exemplo da hierarquia descrita. É possível verificar na figura, que o *overlay* é composto exclusivamente por *peers*, ficando os clientes no nível inferior, acedendo aos serviços do *overlay* através de um ou vários *peers*.

Na especificação do dSIP, os seus autores, definem um conceito de cliente, no qual, um cliente utiliza os serviços do *overlay* através de *peers* especiais, denominados por *adapter-peers*. Este conceito de cliente, pressupõe, que o cliente é um UA SIP normal, que

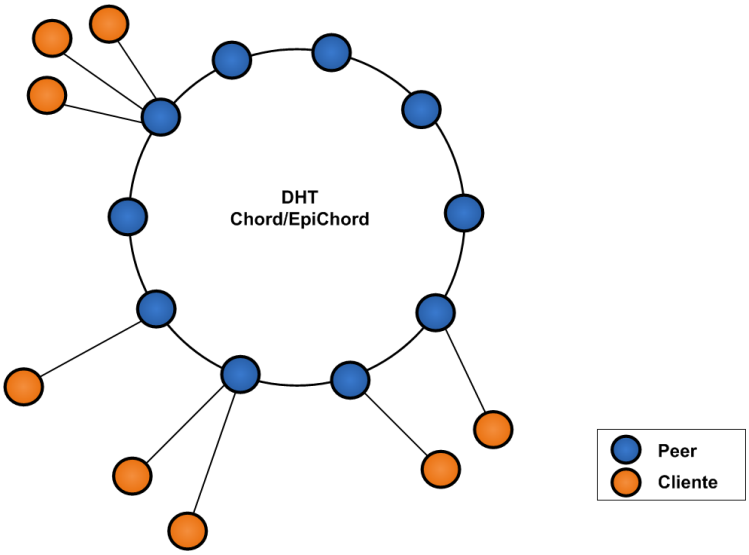

FIGURA 4.1: P2PSIP - Hierarquia de dois níveis

não tem conhecimento do *overlay*, nem implementa um protocolo P2P que lhe permita participar no *overlay*, efectuando uma ligação com um *adapter-peer* do *overlay*, que actua em beneficio do cliente. O papel de um *adapter-peer* neste conceito, é o de actuar como um *gateway* entre o cliente e o *overlay*.

Apesar de ambos os conceitos de cliente, serem idênticos em alguns aspectos, os seus objectivos são diferentes. Nesta dissertação, um cliente é um UA SIP, que se destinge por ter capacidade para participar no *overlay* P2P, mas que, por algum motivo (por exemplo limitações de conectividade) não participa, utilizando por isso um *peer* para aceder aos serviços do *overlay*.

Uma vez que possuem objectivos diferentes, ambos os conceitos podem ser utilizados em simultâneo. Dessa forma é possível que UA SIP normais, sem conhecimento do *overlay*, possam usufruir dos seus serviços, e que um UA P2PSIP que até participava no *overlay*, possa ser despromovido a cliente, de forma a não prejudicar o desempenho do mesmo.

4.2 *Overlay* peer-to-peer

A especificação preliminar do protocolo dSIP[13] não impõe restrições relativamente aos algoritmos DHT que podem ser utilizados pelo *overlay*. Contudo, define que pelo menos

o algoritmo Chord deve ser implementado.
Neste trabalho, para além da implementação obrigatória do algoritmo Chord, decidiu-se implementar o EpiChord, que é uma variante do algoritmo Chord, e que promete melhorar o desempenho geral do *overlay*.

Com a implementação destes dois algoritmos DHT, pretende-se analisar o desempenho de ambos num cenário real, de modo a verificar se os resultados obtidos estão de acordo com os resultados obtidos através de simulação pelos autores do EpiChord.
Para além da análise ao desempenho dos algoritmos, pretendemos também analisar qual o impacto da utilização de um *overlay* com ou sem hierarquia, analisando alguns parâmetros importantes na localização de recursos, como o número de saltos e o tempo médio da localização.

Na implementação desenvolvida neste trabalho, ambos os algoritmos DHT utilizam como algoritmo de *hashing* o SHA-1 de 160 bits, utilizado para gerar o *hash* dos identificadores dos *peers* e dos recursos. O uso de identificadores compostos por 160 bits, significa que o anel formado pelos **peers** do *overlay* tem um espaço de endereçamento 2^{160} identificadores.

Um outro aspecto importante na implementação dos algoritmos DHT da aplicação é a estratégia de encaminhamento a utilizar. Segundo os autores do EpiChord, deve ser utilizada uma estratégia de encaminhamento iterativo, pois uma vez que são enviadas mensagens em paralelo, é necessário prevenir que um *peer* receba várias vezes a mesma mensagem (proveniente de diferentes *peers*). Por isso, com uma estratégia de encaminhamento iterativo, o *peer* que envia as mensagens consegue garantir que uma mensagem não é enviada duas vezes para o mesmo *peer*, e também verificar se as novas mensagens que envia são enviadas para *peers* mais próximos do destino.

Visto que o algoritmo EpiChord foi desenvolvido para funcionar com uma estratégia de encaminhamento iterativo, decidimos que esta deveria ser a estratégia a utilizar na implementação de ambos os algoritmos, Chord e EpiChord. Desta forma, as comparações aos resultados obtidos de ambos os algoritmos DHT tornam-se mais justas.

A implementação do algoritmo Chord neste trabalho foi baseada na especificação preliminar [21] dos mesmos autores de dSIP, na qual se descreve a forma como o algoritmo Chord deve ser implementado num *overlay* P2PSIP dSIP.

Relativamente à implementação do EpiChord, esta seguiu também a especificação preliminar [21] que descreve a forma como o Chord deve ser implementado no dSIP. Um dos pontos principais da implementação do EpiChord, é a gestão da *cache*, pois esta é completamente diferente da gestão da *finger table* do Chord. No EpiChord a *cache* é preenchida com informação proveniente das mensagens recebidas. É importante detectar e remover entradas mortas, que falharam um determinado número de vezes ou cujo *lifetime* exceda um determinado valor pré-definido. A gestão da *cache* é bastante importante para o bom funcionamento do algoritmo, pois a localização de recursos baseia-se na informação contida na *cache*. Para além disso, o facto de não haver um número máximo de entradas na *cache* torna importante a detecção e remoção de entradas expiradas, de modo a limitar o número de entradas na *cache* a cada instante.

Um outro aspecto a ter em conta na implementação deste algoritmo, é a forma como é feita a introdução e localização de recursos no *overlay*. Enquanto que no Chord se recorre a um mecanismo simples de troca de mensagens para inserir ou localizar recursos, no EpiChord o processo é um pouco mais complexo,utilizado-se um mecanismo baseado no envio de mensagens em paralelo.

4.3 Mensagens SIP utilizadas

As mensagens utilizadas pelo protocolo dSIP são mensagens SIP tradicionais com a adição de dois novos tipos de cabeçalhos (DHTPeerID e DHT-Link), necessários para o transporte de informação relevante para a gestão do *overlay peer-to-peer*.

Visto que neste trabalho se pretende que exista a possibilidade de haver clientes e *peers*, numa hierarquia de dois níveis, e como o protocolo dSIP especifica apenas as mensagens que os *peers* devem trocar entre si, não prevendo o cenário da existência deste tipo de clientes. Foi necessário especificar os tipos de mensagens que os clientes devem utilizar para comunicar com os *peers*. Para simplificar este processo, decidiu-se criar um novo

cabeçalho SIP denominado por 'ClientID', e que é baseado no cabeçalho 'DHT-PeerID'. O novo cabeçalho é utilizado apenas pelos clientes, e tem como objectivo permitir que um *peer* possa identificar a origem de uma mensagem. A existência de um cabeçalho 'ClientID' ou 'DHT-PeerID' numa mensagem, permite a um *peer* identificar facilmente se a mensagem é oriunda de um cliente ou de um *peer*.
Desta forma, clientes e *peers* utilizam ambos os tipos de mensagens definidos pelo protocolo dSIP, onde a única diferença nas mensagens enviadas por clientes ou *peers* é a utilização do novo cabeçalho 'ClientID' ou do cabeçalho 'DHT-PeerID'.

4.3.1 Cabeçalhos das mensagens

Relativamente aos cabeçalhos das mensagens SIP definidos pelo dSIP, a implementação dos algoritmos Chord e EpiChord não introduz qualquer alteração nestes, definindo apenas a forma como o cabeçalho *DHT-Link* deve ser constituído.
Na implementação Chord e EpiChord, o cabeçalho *DHT-Link* é utilizado para um *peer* enviar para outros informação relativa ao seu sucessor, antecessor ou elementos da tabela de encaminhamento(*finger table* no Chord, e *cache* no EpiChord).
Segundo [21], na implementação Chord, o parâmetro *link* do cabeçalho deve ter um dos seguintes valores:

- P[N] - Indica que o cabeçalho contêm informação relativa ao antecessor N do *peer*.
- S[N] - Indica que o cabeçalho contêm informação relativa ao sucessor N do *peer*.
- F[N] - Indica que o cabeçalho contêm informação relativa à entrada N da *finger table* do *peer*.

O valor de **N** deve ser um valor positivo, e para *links* do tipo P ou S indica a sua a profundidade devendo este valor ser igual ou superior a 1. Por exemplo, se o parâmetro *link* tiver o valor P1, significa que o cabeçalho contem informação sobre o antecessor imediato do *peer*, já no caso de o valor ser por exemplo S5, significa que a informação contida no cabeçalho é relativa ao quinto sucessor do *peer*. A razão pela qual o valor de N deve ser superior a 1, deve-se ao facto de que se o valor for por exemplo S0 ou P0, o

cabeçalho conteria informação sobre o próprio *peer*, o que não faz sentido.
Se o parâmetro *link* for do tipo F, isto é, para cabeçalhos que possuem informação sobre um *peer* da *finger table*, o valor de N indica o índice do *peer* na *finger table*.

A figura 4.2 mostra um exemplo de um cabeçalho DHT-Link utilizado na implementação Chord do dSIP. Neste exemplo, o cabeçalho possuí informação sobre o sucessor imediato (S1) do *peer* que envia a mensagem. Os atributos relevantes do *peer*, tais como o seu identificador (*peer-ID*) e o endereço IP são visíveis neste cabeçalho.

```
DHT-Link: <sip:peer@192.0.2.1;peer-ID=671a65bf22>;link=S1;expires=600
```

FIGURA 4.2: Chord - Exemplo de um cabeçalho DHT-Link do protocolo dSIP

Diferenças na implementação EpiChord

O conteúdo do cabeçalho DHT-Link na implementação do algoritmo EpiChord, é ligeiramente diferente nomeadamente nos parâmetros *link* e *expires*.
Visto que o EpiChord não utiliza uma *finger table* mas sim uma *cache* como tabela de encaminhamento, o valor **F**[**N**] que o parâmetro *link* pode ter, deixa de fazer sentido. Por isso, na implementação do EpiChord, este deixa de poder ter o valor **F**[**N**] passando a poder ter um novo valor **C**(sem qualquer índice), que indica que o cabeçalho contêm informação relativa a um *link* da *cache*.

Uma outra diferença na implementação EpiChord, tem a ver com o significado do parâmetro *expires* do cabeçalho DHT-Link. O EpiChord não utiliza o parâmetro *expires* nas entradas da sua *cache*, utilizando antes o parâmetro *lifetime*, que tem um significado diferente. Para evitar a criação de um novo cabeçalho, no qual a única diferença seria que o parâmetro *expires* passaria a chamar-se *lifetime*, decidiu-se utilizar o mesmo cabeçalho da implementação Chord, onde o parâmetro *expires* contêm o valor do parâmetro *lifetime* do *link* na *cache* do *peer*.

4.3.2 Tipos de mensagens P2PSIP

Em todos os pedidos P2PSIP efectuados, são utilizadas mensagens SIP do tipo *REGISTER*, e as respectivas respostas, tem habitualmente um dos seguintes três códigos: *302 (Redirect)* quando a mensagem deve ser redirecionada para outro *peer*, *404 (Not Found)* quando o *peer* ou recurso não foi encontrado e *200 (OK)* quando o pedido foi efectuado com sucesso.

Apesar de todas as mensagens P2PSIP serem originadas a partir de uma mensagem SIP do tipo *REGISTER*, existem vários tipos de mensagens P2PSIP, onde cada tipo de mensagem se diferencia pelos cabeçalhos que a compõem.

Os tipos de mensagens, entre pedidos e respostas, utilizados para a comunicação entre *peers*, e também entre *peers* e clientes são os seguintes:

- Peer Register
- Peer Query
- Peer Leave
- Resource Register
- Resource Query
- Resource Remove
- Client Register
- Client Leave
- Client Resource Register
- Client Resource Remove
- Client Resource Query

Apesar de todos esses tipos de mensagens, o formato de algumas dessas mensagens são idênticos, existindo por vezes um parâmetro que define se uma mensagem é de um tipo

ou de outro. Um exemplo são as mensagem de registo e de remoção de recursos. Este tipo de mensagens são idênticos, a única diferença reside no valor do parâmetro *expires*. Se este tiver o valor 0, significa que é uma remoção, caso contrário é um registo.

Mensagens para o registo ou remoção de *Peers*/Recursos

As mensagens utilizadas para o registo ou remoção de *peers* ou recursos no *overlay*, são mensagens P2PSIP, onde os cabeçalhos *To* e *From* são iguais. Estes cabeçalhos possuem o endereço *sip:peer@IP_PEER:PORTA* no registo de *peers*, ou o recurso a armazenar, exemplo *sip:raul@uminho.pt*. Estes cabeçalhos são também compostos pelo parâmetro *peer-ID* ou *resource-ID*, onde o valor do parâmetro é o identificador do *peer* ou recurso a registar. Neste tipo de mensagens, é também necessário incluir um cabeçalho *Contact*, contendo informação de contacto do *peer* ou recurso a registar. O cabeçalho *Expires* deste tipo de mensagem, define se a mensagem é para inserir ou remover um *peer*/recurso. Se o valor deste cabeçalho for 0, significa, no caso de ser relativo a um *peer*, que este vai sair do *overlay*. No caso de um recurso, significa que o recurso deve ser removido do *overlay*.

As figuras 4.4 e 4.3[1] mostram dois exemplos de mensagens P2PSIP para o registo de um *peer* e um recurso no *overlay*.

```
REGISTER sip:192.168.1.89:8000 SIP/2.0
Call-ID: f697609475764dee776ed633e7d67e38@192.168.1.89
CSeq: 1 REGISTER
From: <sip:raul@uminho.pt;resource-ID=ae1b3...e02f8>;tag=56462483
To: <sip:raul@uminho.pt;resource-ID=ae1b3...e02f8>;tag=150bc89e
Via: SIP/2.0/UDP 192.168.1.89:5060;branch=z9hG4bK-383531-ea901c...
Max-Forwards: 70
Contact: <sip:raul@192.168.1.89:5060>
DHT-PeerID: <sip:peer@192.168.1.89:5060;peer-ID=8f066...8fef9>;algorithm=sha1;dht=Chord1.0;overlay=braca;expires=1
Require: dht
Supported: dht
Expires: 999
Content-Length: 0
```

FIGURA 4.3: Exemplo de uma mensagem P2PSIP para o registo de um recurso

Na figura 4.4 é possível verificar que a mensagem SIP é do tipo *Register* e que os cabeçalhos *From* e *To* possuem informação relativa ao *peer* que pretende entrar no *overlay*.

[1]De forma a reduzir o tamanho da imagem, nesta e noutras imagens idênticas, os identificadores dos *peers* e recursos foram reduzidos.

```
REGISTER sip:192.168.1.89:10000 SIP/2.0
Call-ID: abb897c1393d2a80de6ce048e56d32d1@192.168.1.89
CSeq: 1 REGISTER
From: <sip:peer@192.168.1.89:5060;peer-ID=8f066...8fef9>;tag=76a97f77
To: <sip:peer@192.168.1.89:5060;peer-ID=8f066...8fef9>;tag=e131d95f
Via: SIP/2.0/UDP 192.168.1.89:5060;branch=z9hG4bK-343533-a42f9...
Max-Forwards: 70
Contact: <sip:peer@192.168.1.89:5060;peer-ID=8f066...8fef9>
DHT-PeerID: <sip:peer@192.168.1.89:5060;peer-ID=8f066...8fef9>;algorithm=sha1;dht=Chord1.0;overlay=braca;expires=1
Require: dht
Supported: dht
Expires: 1
Content-Length: 0
```

FIGURA 4.4: Exemplo de uma mensagem P2PSIP para o registo de um *peer*

Mensagens para localização de *Peers*/Recursos

As mensagens de localização de *peers* ou recursos, caracterizam-se pelo facto de o cabeçalho do destinatário, *To* ter o seguinte endereço SIP *sip:peer@0.0.0.0*. Este cabeçalho é seguido do parâmetro peer-ID (na localização de *peers*) ou resource-ID (na localização de recursos). Como valor, este parâmetro tem o identificador do *peer* ou recurso a localizar.

A figura 4.5 contêm um exemplo de uma mensagem P2PSIP para a localização de um *peer* no *overlay*.

```
REGISTER sip:192.168.1.89:8000 SIP/2.0
Call-ID: f5eba0513a516430b43b70457fb1c389@192.168.1.89
CSeq: 1 REGISTER
From: <sip:peer@192.168.1.89:5060;peer-ID=8f066...8fef9>;tag=b398c146
To: <sip:peer@0.0.0.0;peer-ID=b8491...5588b>;tag=20ba984e
Via: SIP/2.0/UDP 192.168.1.89:5060;branch=z9hG4bK-323932-5aadb...
Max-Forwards: 70
DHT-PeerID: <sip:peer@192.168.1.89:5060;peer-ID=8f066...8fef9>;algorithm=sha1;dht=Chord1.0;overlay=braca
Require: dht
Supported: dht
Content-Length: 0
```

FIGURA 4.5: Exemplo de uma mensagem P2PSIP para a localização de um *peer*

Capítulo 5

Implementação

Neste capítulo descreve-se a arquitectura e implementações desenvolvidas, onde o JAVA foi a linguagem de programação escolhida. Os algoritmos DHT implementados foram o Chord [3] e o EpiChord [4], tendo ambos sido implementados de forma a poderem funcionar num *overlay* com um sem hierarquia.

5.1 Arquitectura

A arquitectura das aplicações desenvolvidas foi implementada de modo a que a mesma arquitectura pudesse ser utilizada pelas várias aplicações, permitindo ser adaptada a qualquer tipo de algoritmo DHT que se pretenda implementar. Para tal, o desenho da arquitectura pode ser dividido em várias camadas, posicionadas hierarquicamente. A figura 5.1 ilustra as camadas envolvidas na arquitectura das aplicações.

Em termos de desenvolvimento, neste trabalho foram desenvolvidas as camadas P2PSIP e a camada responsável pelo algoritmo DHT a utilizar. As camadas SIP e transporte foram implementadas recorrendo à biblioteca JAIN-SIP[22]. Esta biblioteca oferece um conjunto de API's que permitem facilmente criar uma aplicação capaz de enviar e receber mensagens SIP.

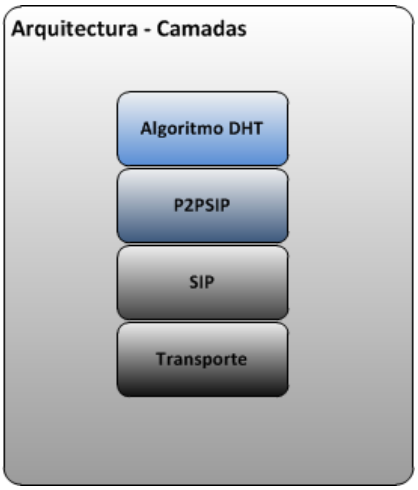

FIGURA 5.1: Arquitectura das aplicações - Camadas

5.1.1 Camada P2PSIP

Esta camada é responsável por efectuar a ligação entre as camadas superiores da aplicação e a *stack* SIP a utilizar. Deste modo, a utilização de diferentes bibliotecas SIP, ou, até mesmo bibliotecas para suportar outros protocolos, depende apenas de modificações ao nível desta camada. A separação em diferentes camadas permitiu, que a implementação das camadas superiores, pudesse ser feita independentemente da biblioteca SIP a utilizar.

Para criar um certo nível de abstracção com as camadas superiores, esta camada recorre a um conjunto de classes e interfaces, das quais se destacam, as classes *SipLayer*, *ConvertToSIP*, *ConvertToP2PSIP* e as interfaces *IP2PSIPResponseListener*, *IP2PSIPRequestListener* e *IMessagesStatsListener*.

A figura 5.2 mostra as entidades principais que compõe esta camada.

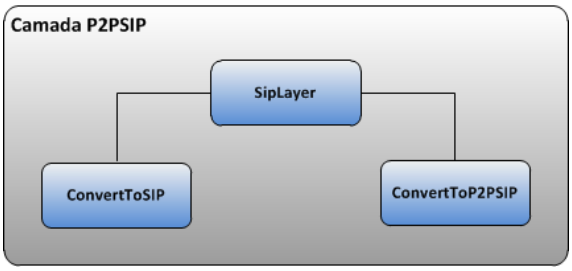

FIGURA 5.2: Entidades principais da camada P2PSIP

A classe *SipLayer* é a classe principal desta camada pois é responsável por comunicar directamente com a *stack* SIP, para o envio e recepção de mensagens. Ao ser inicializada a classe SipLayer, é necessário passar uma referência para um objecto que implemente a interface *IP2PSIPRequestListener*. Essa referência é necessária para que a camada P2PSIP possa notificar a camada DHT da chegada de novas mensagens.

Envio de Mensagens

Para o envio de mensagens P2PSIP, a classe SipLayer disponibiliza os seguintes métodos:

- *public void enviarPedidoP2PSIP(P2PSIPMessage msg, IP2PSIPResponseListener callback)* - Este método é utilizado para o envio de um pedido P2PSIP. Como parâmetros recebe a mensagem P2PSIP a enviar, assim como uma referência para o objecto que deverá ser notificado quando for recebida uma resposta para a mensagem enviada.
- *public void enviarRespostaP2PSIP(P2PSIPMessage msg)* - Este método é utilizado para o envio de uma mensagem de resposta a um pedido P2PSIP recebido. Como parâmetros tem apenas a mensagem P2PSIP a enviar como resposta ao pedido.

A interface *IP2PSIPResponseListener* é utilizada para que após o envio de um pedido P2PSIP, a sua resposta possa ser entregue a um objecto na camada superior. Esta interface, define os seguintes métodos abstractos:

- *public void onTransFailureResponse(P2PSIPMessage msg)* - Método executado, quando o código de resposta da mensagem SIP tem o valor superior a 299. O único parâmetro deste método é um parâmetro do tipo *P2PSIPMessage*, que contêm a resposta.
- *public void onTransProvisionalResponse(P2PSIPMessage msg)* - Método executado, quando o código de resposta da mensagem SIP tem um valor entre 100 e 199. O único parâmetro deste método é um parâmetro do tipo *P2PSIPMessage*, que contêm a resposta.
- *public void onTransSuccessResponse(P2PSIPMessage msg)* - Método executado, quando o código de resposta da mensagem SIP é um código de sucesso, isto é, tem um valor

compreendido entre 200 e 299. O único parâmetro deste método é um parâmetro do tipo *P2PSIPMessage*, que contêm a resposta.

- *public void onTransTimeout()* - Método executado quando o envio de uma mensagem P2PSIP resulta em *timeout*.

A utilização de uma interface para notificar a camada superior da chegada de uma resposta, permite criar uma maior abstracção entre as camadas. Desta forma, no envio de um pedido P2PSIP, a camada P2PSIP requer apenas uma referência para um objecto, que pode ser de qualquer tipo, necessitando apenas de implementar a interface especificada. A camada P2PSIP é apenas responsável por notificar o objecto em questão, da ocorrência de um desses eventos. A implementação dos métodos definidos por esta interface, depende exclusivamente da camada superior.

Os métodos disponíveis para o envio de mensagens P2PSIP recorrem à classe *ConvertToSIP* para efectuar a conversão do objecto *P2PSIPMessage* (que contêm a informação da mensagem P2PSIP a enviar) num objecto SIP reconhecido pela camada SIP em utilização.

Recepção de Mensagens

A recepção de mensagens SIP é feita de diferentes formas, consoante a mensagem recebida for um pedido, ou uma resposta a um pedido enviado anteriormente. Em ambos os casos, é verificado se a mensagem SIP recebida, é uma mensagem P2PSIP. Esta verificação é feita através da análise dos cabeçalhos da mensagem recebida. Após se verificar que a mensagem recebida é uma mensagem P2PSIP, a mensagem SIP é convertida numa mensagem P2PSIP equivalente, recorrendo à classe *ConvertToP2PSIP*.

No caso de a mensagem recebida ser um pedido, esta é entregue na camada DHT, através da interface *IP2PSIPRequestListener* de modo a ser processada. Caso seja uma resposta a um pedido enviado anteriormente, o código da resposta da mensagem SIP é analisado e consoante o seu valor, a notificação enviada para a camada DHT difere. A notificação da camada DHT é feita através da interface *IP2PSIPResponseListener* do objecto de *callback* referenciado no envio da mensagem que originou a resposta recebida.

5.1.2 Camada DHT

Esta camada, é responsável pela implementação do algoritmo P2P utilizado para criar e manter um *overlay* P2P. A base desta camada é formada por um conjunto de classes e interfaces genéricas, que podem ser extendidas por cada implementação de um algoritmo P2P que se pretenda utilizar. Das classes base, a classe *Peer* e *Cliente* são das mais importantes. A classe Peer, representa um *peer* genérico, implementado e definindo um conjunto de métodos (alguns deles abstractos) comuns a qualquer *peer*, como por exemplo os métodos *put* e *get*.
A classe *Cliente* implementa as funcionalidades que um cliente de um *overlay* P2PSIP deve possuir. Para além destas duas classes, existem outras, como por exemplo, classes para representar e gerir os recursos que cada *peer* possui.

Neste trabalho, para a implementação dos algoritmos P2P Chord e EpiChord foram criadas duas novas classes *ChordPeer* e *EpiChordPeer*. Estas duas classes, são baseadas na classe base *Peer*, implementando, entre outros, os métodos que a classe base define como abstractos, com o funcionamento desejado para cada um dos algoritmos.

A figura 5.3 mostra algumas das classes que formam a base desta camada.

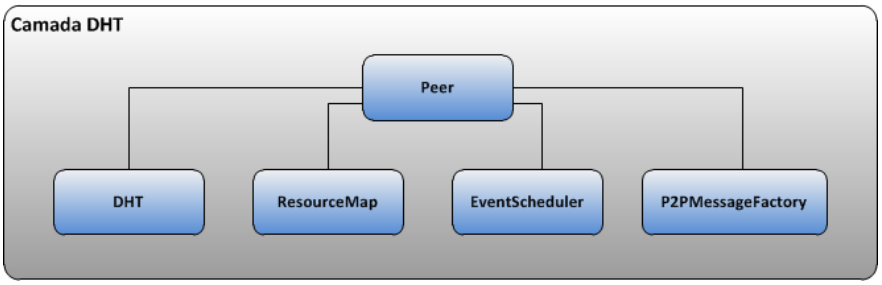

FIGURA 5.3: Entidades principais da camada DHT

A comunicação entre *peers* do *overlay*, é feita através dos serviços disponibilizados pela camada P2PSIP para o envio e recepção de mensagem P2PSIP. Por isso, e uma vez que independentemente do algoritmo P2P, um *peer* e/ou cliente necessita de ser notificado da chegada uma mensagem P2PSIP, as classes *Peer* e *Cliente*, implementam a interface *IP2PSIPRequestListener*.

Informação sobre o algoritmo - *DHT*

A classe DHT desta camada, é uma classe genérica, com métodos abstractos e que deve ser extendida na implementação de algoritmos DHT. O objectivo é permitir que todos os algoritmos DHT utilizem classes derivadas desta para representar informação relativa ao algoritmo DHT. A informação contida nesta classe identifica o nome do algoritmo DHT, por exemplo *Chord*, assim como o nome do algoritmo utilizado para calcular o *hash* dos identificadores.

Para além dessa informação relativa ao algoritmo, esta classe define um método abstracto para o calculo de distâncias entre dois identificadores. Este método é abstracto, pois a forma como é calculada a distância entre identificadores varia consoante o algoritmo DHT em utilização. Por isso, todos os algoritmos DHT implementados, devem implementar uma classe derivada desta, na qual este método é implementado de acordo com o algoritmo.

Temporizador de eventos - *EventScheduler*

O temporizador de eventos, é utilizado em diversos cenários, sempre que é necessário que uma tarefa seja executada num determinado instante de tempo.

A classe *EventScheduler* contém a implementação do temporizador e é responsável pela gestão dos eventos, oferecendo métodos para adicionar ou remover eventos. O armazenamento da informação relativa aos eventos é feito recorrendo a uma *HashMap*, onde a chave de cada registo é o nome do evento, e o valor associado é uma instância da classe *Event*. Para além da *HashMap* que contêm os eventos, a classe *EventScheduler* possui ainda um temporizador responsável por detectar o instante de tempo em que cada evento deve ocorrer. Como temporizador foi utilizado o *ScheduledExecutorSipTimer* da biblioteca Jain-SIP. Para adicionar novos eventos a este temporizador é necessário uma instância de *Event*, que possui informações sobre o evento, assim como o instante de tempo no qual se pretende que o evento adicionado ocorra.

A classe *Event* é uma classe que deriva de *SIPStackTimerTask* da biblioteca Jain-SIP, e contêm informação sobre o evento, como o nome, o instante de tempo em que o evento

deve ocorrer, e também uma referência para um objecto que implementa a interface *IEvent*. Assim que o temporizador detecta que chegou o momento de disparar um determinado evento, é executado o método *runTask* desse evento. Esse método uma vez executado, notifica o objecto interessado no evento, por exemplo um Peer, da ocorrência do evento através do método *onEventTimeout* da interface *IEvent* que o objecto interessado deve implementar. A notificação passada ao objecto, possuí apenas o nome do evento que ocorreu, a partir do nome do evento, o objecto deverá ser capaz de identificar o significado do evento, e iniciar, se necessário uma determinada tarefa.

Gestão de recursos - ResourceMap

A gestão de recursos é algo comum a qualquer *peer* pertencente a um *overlay* DHT, independentemente do algoritmo em utilização. Por isso, na camada DHT foram implementadas as classes *Resource* e *ResourceMap* para a gestão de recursos. A classe *Resource* é utilizada para guardar informação relativa a um recurso, nomeadamente: o identificador, nome, valor associado ao recurso e também o tempo ao fim do qual o recurso expira. Para gerir os recursos armazenados num *peer*, é utilizada a classe *ResourceMap*. Esta recorre a uma *HashMap* para efectuar o armazenamento dos recursos. A chave de cada registo inserido na *HashMap* é o identificador do recurso e o valor associado à chave é uma instância da classe *Resource* que contêm informação sobre o recurso.

Para além da gestão de recursos em termos de inserções e remoções, a classe *ResourceMap* necessita de detectar quando um recurso expira, de forma a que este seja removido. Para isso, é utilizada a classe *EventScheduler* para criar um temporizador para gerir o prazo de validade de cada recurso. Cada vez que é inserido ou actualizado um recurso, é adicionado ou actualizado um evento ao temporizador de modo a que esse evento ocorra no instante de tempo em que o recurso expira. Assim quando um determinado recurso expira, o evento associado ao recurso é disparado e nesse instante o recurso é removido da lista de recursos armazenados.

O gestor de recursos possui ainda, uma referência para uma instância de um objecto que implementa a interface *ResourceRegistrationListener*, referência essa que permite que

esse objecto, (o *Peer*, pois implementa esta interface) seja notificado sempre que um recurso é inserido, ou é removido por ter expirado.

Criação de mensagens P2PSIP - *P2PMessageFactory*

A criação de mensagens P2PSIP, é uma tarefa comum a várias funcionalidades de um *peer*/cliente, independentemente do algoritmo P2P. Por isso, e para evitar a repetição de código, foi criada a classe *P2PMessageFactory*. Essa classe possui um conjunto de métodos que permitem a criação de diferentes tipos de mensagem P2PSIP. Assim, a criação dos tipos de mensagens mais comuns, como *queries* para localizar *peers* ou recursos ou o registo de *peers* e recursos é feita nesta classe, evitando a repetição de código.

A especificação do dSIP[13] define os tipos de mensagens P2PSIP utilizados neste trabalho. Para suportar a comunicação entre *peers* e clientes, foi necessário criar novos tipos de mensagens. A única diferença entre os novos tipos de mensagens, e os tipos equivalentes para comunicação entre *peers*, é a substituição do cabeçalho *DHT-PeerID* pelo novo cabeçalho *ClientID*.

Gestão de clientes

Neste trabalho, para além da criação e gestão de um *overlay* P2P, um *peer* possui a capacidade de comunicar com entidades externas ao *overlay*. Essas entidades, denominadas por Clientes, utilizam os *peers* do *overlay* para aceder aos serviços que o *overlay* disponibiliza, neste caso, a inserção e localização de recursos. Para isso, um *peer* necessita de processar pedidos de clientes, e actuar em beneficio do cliente.

Existem três tipos de pedidos que um *peer* pode receber vindo de um cliente: pedido de registos do cliente, inserir ou remover um recurso, e localizar recursos. Cada um desses pedidos é processado nos métodos *processClienteRegistration*, *processClienteResourceRegistration*, *processClientResourceQuery*, definidos de forma abstracta na classe *Peer* da camada DHT.

Assim que é recebido um pedido de um cliente, um *peer* actua de acordo com o seu algoritmo DHT (Chord ou EpiChord), tratando de enviar todas as mensagens necessárias

para satisfazer o pedido. O cliente só recebe uma mensagem de resposta ao seu pedido, quando este for concluído, isto é, se um cliente pretende inserir um recurso no *overlay*, todas as mensagens intermédias que os *peers* podem ter de trocar não são do conhecimento do cliente. Assim que o *peer* concluí o pedido do cliente, informa-o do resultado da operação.

Para além do processamento de pedidos, um *peer*, quando pretende abandonar o *overlay*, deve avisar os clientes aos quais está ligado, de que vai sair do *overlay*, enviando-lhes uma lista com contactos de outros *peers* pertencentes ao *overlay*. O envio dessa lista serve para que o cliente passe a conhecer novos *peers*, com os quais pode estabelecer uma ligação para usufruir dos serviços do *overlay*.

5.2 Chord

O funcionamento do algoritmo Chord implementado neste trabalho é baseado na especificação descrita em [21]. Nesta especificação é descrita a forma como o algoritmo deve ser implementado recorrendo ao protocolo dSIP.

A implementação Chord foi baseada na implementação de Cirani et al.[23], contudo o código sofreu bastantes modificações. Foi aproveitada a estrutura das classes, ou seja, a forma como as classes associadas à implementação de um algoritmo DHT se interligam. Relativamente ao código, apenas algumas partes foram utilizadas, para por exemplo a criação de mensagens P2PSIP. Mesmo as partes de código que foram reutilizadas sofreram bastantes modificações, ate porque a *stack* SIP, que na implementação de Cirani et al. era a MJSIP, foi, nesta implementação alterada para a Jain-SIP.

Esta implementação, utiliza, entre outras, as classes base *Peer*, *ResourceMap* e *EventScheduler* disponibilizadas pela camada DHT. Algumas destas classes, como por exemplo a *ResourceMap* oferecem de base todas as funcionalidades necessárias, pelo que não houve necessidade de as extender. Já a classe *Peer*, que por defeito, implementa e define alguns métodos genéricos, teve de ser extendida, originando uma nova classe *ChordPeer*.

A classe ChordPeer é a classe principal da implementação Chord, todo o processamento de pedidos P2PSIP que um *peer* Chord receba, é processado de acordo com a especificação

do algoritmo nesta classe. O processamento deste tipo de mensagens, inclui, entre outros, pedidos de admissão de novos *peers* e também pedidos para a inserção de recursos.

Para além do processamento de mensagens, a implementação do Chord necessita de componentes para a gestão de recursos, gestão da *finger table* e execução de determinadas tarefas periodicamente. Destes componentes, dois deles foram implementados recorrendo às classe base da camada DHT *ResourceMap* e *EventScheduler*. A gestão da tabela de encaminhamento, uma vez que depende do algoritmo, necessita de uma implementação especifica, para isso, foi criada a classe *ChordRoutingTable*.

As funcionalidades que o *overlay* oferece, tais como armazenamento ou localização de recursos podem ser mapeadas em acções que um *peer* deve desempenhar. Uma acção neste contexto, é um pedido que um *peer* efectua a um outro *peer* pertencente ao *overlay*. Existem diversos tipos de acções que um *peer* pode executar, umas associadas aos serviços disponibilizados pelo *overlay* e outras a tarefas de manutenção que são necessárias para manter o bom funcionamento de todo o *overlay*.

Nesta implementação, cada acção que um *peer* pode executar encontra-se implementada na sua própria classe. Por cada nova acção que um *peer* execute, é criada uma nova instância da classe associada a essa acção. É depois nessas classes que as mensagens com os pedidos P2PSIP são criadas e enviadas. O envio dos pedidos é feito directamente através da classe *SipLayer* da camada P2PSIP. A resposta ao pedido enviado é recebida pela classe *SipLayer*, e enviada directamente para a instância do objecto que iniciou a acção (enviou o pedido) através da interface *IP2PSIPResponseListener* que estas classes implementam. Desta forma, toda a lógica relativa aos vários tipos de acções, encontra-se na classe associada a essa acção, sendo mais fácil efectuar alterações, assim como gerir e detectar possíveis erros.

Apesar de as várias acções estarem implementadas cada uma na sua classe, a classe *ChordPeer* continua a desempenhar um papel importante, pois é ela a responsável por inicializar cada uma das acções disponíveis.

A implementação Chord permite que alguns dos seus parâmetros sejam configurados. A configuração desses parâmetros é feita através da classe principal *ChordPeer*. Esta classe permite a configuração dos seguintes parâmetros do sistema:

- Número de entradas na *finger table*
- Número de réplicas dos recursos
- O período do processo de estabilização do algoritmo.
- O período para a verificação das entradas da *finger table*

5.2.1 Criação e manutenção do *Overlay*

Para criar um *overlay*, um *peer* não necessita de efectuar nenhum processo especifico, basta iniciar os seus apontadores para os seus vizinhos(sucessores e antecessores) assim como a *finger table*. Uma vez criado, e estando inicialmente sozinho no *overlay*, o *peer* deve definir-se a si próprio como seu sucessor. Já o apontador para o antecessor deve ter o valor *NULL*, pois segundo a especificação [21] o antecessor de um *peer* nunca pode ser o próprio *peer*. A *finger table* deve ser inicializada contendo em todas as suas entradas informação relativa ao próprio *peer*.

Estando o *overlay* construído, e populado, é necessário efectuar periodicamente tarefas de manutenção. Isto porque a estabilidade do *overlay* pode ser afectada pela entrada e saída abrupta de *peers*. As tarefas de manutenção que um *peer* deve efectuar servem para manter actualizadas as informações relativas aos seus vizinhos directos (sucessor e antecessor) e as entradas da sua *finger table*.

A estabilização periódica de um *peer*, consiste na verificação do estado do seu sucessor, e se este ainda o considera seu antecessor. Para isso é enviada uma mensagem do tipo *Peer Query* onde o Peer-ID a localizar é o identificador do sucessor. Na mensagem de resposta é verificado se o antecessor do seu sucessor é o *peer* que enviou a mensagem de localização. Se não for, significa que um novo *peer* se ligou entre eles. Neste caso o *peer* define como seu novo sucessor, o antecessor retornado na mensagem de resposta, e envia para este uma mensagem do tipo *Peer Register*.

A figura 5.4 mostra um exemplo no qual é visível um cenário em que um *peer* possui momentaneamente informação desactualizada sobre o seu sucessor.

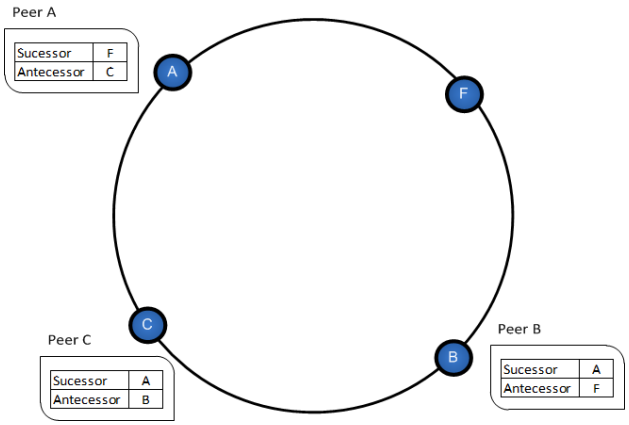

FIGURA 5.4: Exemplo no qual o *peer B* possuí informação de encaminhamento desactualizada

No exemplo apresentado na figura 5.4 o *overlay* é composto por apenas quatro *peers*, dos quais o A, B e F foram os primeiros a entrar. Posteriormente, um novo *peer*, C, decide juntar-se ao *overlay*. Aplicando a função de *hash* utilizada pelo *overlay* ao endereço ip e porta do novo *peer*, o seu identificador pertence ao espaço de endereçamento pelo qual é responsável o *peer* A. Após efectuar o processo de admissão ao *overlay* com o *peer* A, o novo *peer* é finalmente adicionado ao *overlay*. Na mensagem de confirmação enviada por A para C, este envia a informação relativa ao seu sucessor (F) e antecessor (B). Com esta informação, o novo *peer* que já sabia que o seu sucessor é o *peer* que o admitiu, ou seja A, passa também a saber que o seu antecessor é o *peer* B. Contudo, o *peer* B não se apercebe da chegada do novo *peer*, pelo que para ele o seu sucessor continua a ser o *peer* A. Assim que o *peer* B inicializa o processo de estabilização periódico, envia uma mensagem de localização para o *peer* que pensa ser o seu sucessor, ou seja A. O *peer* A recebe a mensagem de B, e verifica que o *peer* que B procura é ele próprio, respondendo com uma mensagem do tipo 200 (OK), incluindo na mensagem informação sobre o seu sucessor (F) e o seu antecessor (C). Assim que o *peer* B recebe a mensagem de resposta, verifica que o *peer* A tem como seu antecessor um outro *peer* que não B, significando isto que um novo *peer* se ligou entre A e B. Ao detectar essa situação, o *peer* B verifica que o *peer* C é o seu novo sucessor, iniciando de seguida o processo de registo junto do novo sucessor. Este processo de registo é igual ao processo que um *peer* efectua quando pretende juntar-se ao *overlay*, enviando uma mensagem do tipo *Peer Register*. Após ser efectuado o registo junto do novo sucessor, o *peer* B actualiza a informação relativa ao

seu sucessor, passando nesse instante a informação a estar correcta.

Na manutenção periódica do *overlay*, é também necessário verificar as entradas da *finger table*, pois devido à entrada e saída constante de *peers* no *overlay*, a informação contida na *finger table* rapidamente fica desactualizada. Isto é, alguns dos *peers* podem já ter abandonado o *overlay* ou já não serem eles os responsáveis pelo espaço de endereçamento associado à entrada da *finger table* na qual eles se encontram. Por isso, é necessário verificar periodicamente as entradas da *finger table*.

A actualização da *finger table*, consiste no envio de mensagens de localização, do tipo *Peer Query* para localizar o *peer* responsável pelo espaço de endereçamento associado a cada entrada da *finger table*. Assim que é localizado o *peer* responsável pelo identificador de uma entrada, a entrada da *finger table* associada a esse identificador passa a estar associada a esse *peer*.

Os autores de [21] recomendam que o período para estas verificações seja um valor entre 60 e 360 segundos. Neste trabalho, foi definido um período de 70 segundos para a estabilização do algoritmo (verificação do sucessor e também do antecessor) e de 100 segundos para a verificação das entradas da *finger table*. Estes valores foram escolhidos tendo em conta que para o bom funcionamento do algoritmo é mais importante manter mais actualizado os apontadores para o sucessor e antecessor do que a *finger table*.

Na implementação deste algoritmo, os processos de manutenção foram implementados nas seguintes classes:

- *Stabilize* - Classe utilizada para verificar o estado do sucessor, verificando se este se mantêm o seu sucessor
- *CheckPredecessor* - Classe utilizada para verificar o estado do antecessor, verificando se este se mantêm o seu antecessor
- *FixFinger* - Classe utilizada para obter o *peer* responsável por uma determinada entrada da *finger table*
- *KeepAlive* - Classe utilizada para que o *peer* periodicamente se registe novamente no seu sucessor, actualizando o parâmetro *expires* associado ao seu registo

- *ResourceTransfer* - Classe utilizada sempre que é necessário efectuar a transferência de recursos entre *peers*.

Cada uma dessas classes é responsável pelo envio dos pedidos P2PSIP associados ao tipo de tarefa a executar, assim como efectuar o processamento das respostas recebidas. A classe *ChordPeer* é a responsável por inicializar e executar estas tarefas. Utiliza um temporizador de eventos, no qual regista as tarefas para as quais deseja ser notificada quando o período de tempo associado a cada tarefa expirar. Assim que o temporizador, notifica a classe *ChordPeer* (através do método *onEventTimeout* da interface *IEvent*) esta executa um determinado método de acordo com a tarefa a executar.

5.2.2 Gestão da tabela de encaminhamento

A tabela de encaminhamento de um *peer* Chord é composta pela *finger table* e por apontadores, para o sucessor e antecessor do *peer*.

Neste trabalho, a implementação da tabela de encaminhamento de um *peer* Chord, foi efectuada na classe *ChordRoutingTable*. Esta classe é bastante simples. É utilizada apenas para armazenamento de informação relativa aos vizinhos(sucessor e antecessor) e às entradas da *finger table*, oferecendo um conjunto de métodos para gerir essa informação. O número de entradas na *finger table* é um parâmetro configurável do sistema, podendo o seu valor ser alterado através da classe *ChordPeer*.

Na classe *ChordRoutingTable* a informação sobre os *peers*, seja eles vizinhos, ou elementos da *finger table*, é efectuada com recurso a instâncias de objectos da classe *Contact*.

A classe *Contact* é utilizada para guardar informação sobre um determinado *peer*, guardando: uma string com o identificador do *peer*, o endereço ip, a porta, e um parâmetro que define a validade do contacto. O valor deste último parâmetro contém o número de segundos que o contacto tem de validade, se esse valor chegar a 0, o contacto deve ser renovado. O algoritmo Chord, implementa mecanismos de estabilização, que asseguram que os *peers* periodicamente actualizam a informação relativa aos seus vizinhos e à sua *finger table*. Sempre que estes mecanismos são executados, a validade dos contactos vai sendo actualizada consoante as mensagens que vão sendo recebidas.

Todos os métodos oferecidos por esta classe, são controlados e manuseados na classe principal *ChordPeer*. É também nesta classe que é feita a gestão da validade dos contactos. Isto é, sempre que é actualizado um contacto se este pertencer à *finger table*, é criado no temporizador de eventos da classe um novo evento que deverá disparar uns segundos antes do contacto expirar. Assim que o evento é disparado, a classe *ChordPeer* inicializa o processo de actualização do contacto prestes a expirar. O valor utilizado no temporizados para que o evento dispare, corresponde a *2/3* da validade do contacto, isto é, se um contacto tiver uma validade de 30 segundos, o evento irá disparar quando restarem apenas 10 segundos para o contacto expirar.

5.2.3 Encaminhamento de mensagens

O algoritmo Chord implementado, utiliza um mecanismo de encaminhamento iterativo. Utilizando encaminhamento iterativo, quando um *peer* recebe uma mensagem para a qual não é o responsável pelo identificador pretendido, deve responder com uma mensagem de redirecionamento, do tipo 302 (Redirect). Indicando nessa mensagem o *peer* que conhece que poderá ser o responsável pelo espaço de endereçamento ao qual pertence o identificador pretendido. O originador da mensagem, ao receber a mensagem de resposta envia uma nova mensagem, desta vez destinada ao *peer* contido na mensagem de redirecionamento recebida. Este processo é repetido até que o *peer* responsável pelo identificador seja localizado.

A figura 5.6 apresenta um diagrama de sequência no qual é visível a troca de mensagens envolvidas na admissão de um novo *peer*. Na admissão desse novo *peer*, o *peer* foi redirecionado duas vezes, até encontrar o *peer* responsável pela sua admissão.

5.2.4 Algoritmo de localização

A localização de recursos e *peers*, é uma das principais funcionalidades de uma rede *peer-to-peer*. Para além da localização de recursos, em determinadas situações é também necessário localizar *peers*. A localização de *peers* utiliza mensagens do tipo *Peer Query* e tem como objectivo saber se um *peer* com determinado identificador está presente no

overlay, ou no caso de não estar, qual o *peer* responsável pelo identificador. O processo de construir e actualizar a *finger table*, recorre a este tipo de localização para preencher as entradas da tabela.

Num *overlay* Chord, cada *peer* é responsável por armazenar a informação relativa aos recursos cujos identificador estejam compreendidos entre o seu identificador e o identificador do *peer* que o antecede no *overlay*.

O fluxograma apresentado na figura 5.5 representa o algoritmo utilizado para determinar (com base na tabela de encaminhamento), qual o *peer* que deverá ser responsável por um identificador.

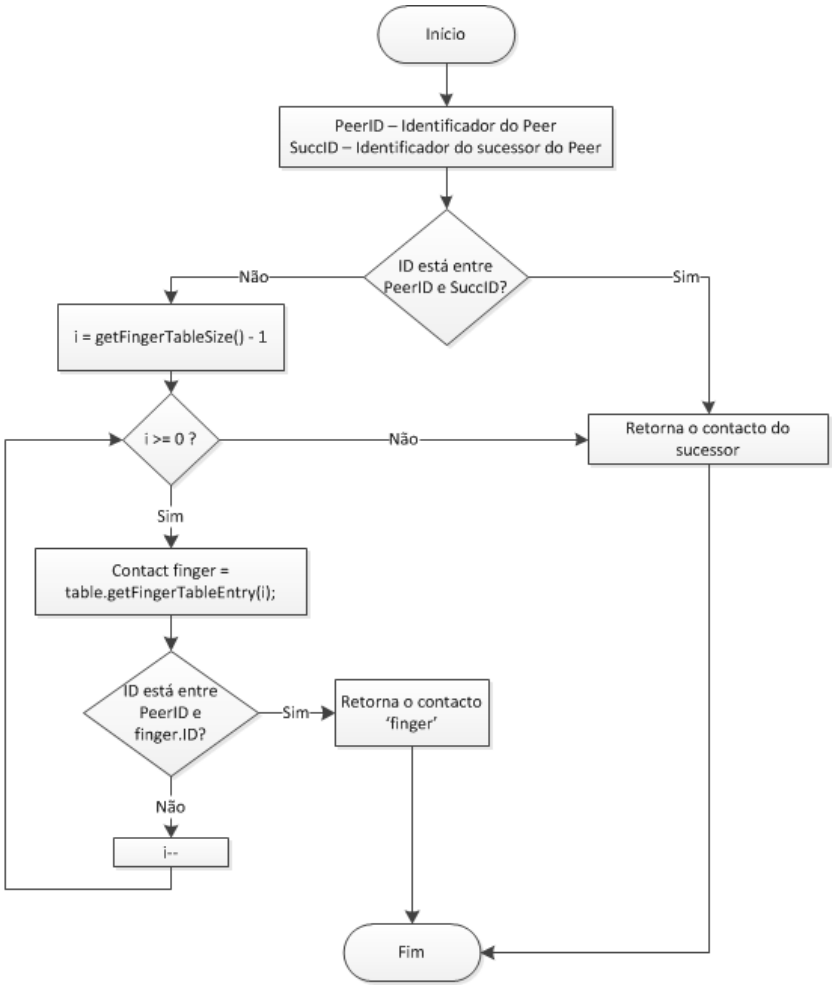

FIGURA 5.5: Fluxograma - Obtenção do *peer* responsável por um identificador

O algoritmo representado pelo fluxograma da figura 5.5 foi implementado na classe *ChordPeer*, e o seu funcionamento encontra-se dividido em vários métodos. O algoritmo implementado é utilizado em todas as situações em que um *peer* sabe que o identificador não é da sua responsabilidade, e que necessita de obter através da sua tabela de encaminhamento (sucessor, antecessor e *finger table*), o *peer* que deverá ser responsável. Por exemplo, quando uma mensagem de localização de um recurso ou *peer* é recebida, se o *peer* que a recebe não for o responsável, executa este algoritmo de modo a obter o contacto do *peer* que poderá ser o responsável. Neste exemplo, o contacto do *peer* seria enviado numa mensagem de resposta com o código 302 (Redirect).

Devido ao facto de cada *peer* possuir informação sobre apenas alguns *peers* do *overlay*, o *peer* obtido através deste algoritmo, é aquele, que dos *peers* conhecidos, se encontra mais proximo e aparenta ser o responsável pelo identificador. Contudo, não há garantias de que de facto o seja, pelo que só após ser contactado é possível determinar se este é ou não responsável. No caso de não o ser, o processo é repetido, e a cada nova iteração o destinatário pretendido vai estando mais próximo, até que por fim é encontrado.

5.2.5 Admissão de *peers*

A admissão de novos *peers* ao *overlay* é feita pelo *peer* responsável pelo espaço de endereçamento ao qual o identificador do novo *peer* pertence.

Quando um *peer* recebe uma mensagem do tipo *Peer Register* deve verificar se é o responsável pela admissão do novo *peer*. Se for envia uma mensagem de resposta do tipo 200 (OK), confirmando a sua admissão no *overlay*. Na mensagem de resposta é incluída informação sobre o sucessor e antecessor do *peer*. Essa informação é utilizada pelo novo *peer* para saber quem são os seus vizinhos, nomeadamente o seu antecessor, pois o sucessor é sempre o *peer* que o admitiu no *overlay*.
No caso de o *peer* não ser o responsável pela admissão, envia uma mensagem de reencaminhamento, contendo o contacto do *peer* que este deverá contactar para se juntar ao *overlay*. Uma vez recebida a mensagem de redirecionamento, o novo *peer* recomeça o processo, enviando nova mensagem do tipo *Peer Register* para o novo contacto.

A admissão do novo *peer*, pode significar que alguns dos recursos armazenados pelo *peer* que o admitiu, passam a ser da responsabilidade do novo *peer*. Nesse caso, é iniciado o processo de transferência de recursos.

As mensagens do tipo *Peer Register* nem sempre são utilizadas por *peers* não pertencentes ao *overlay*. *Peers* pertencentes ao *overlay*, enviam este tipo de mensagens sempre que detectam um novo sucessor. Nestes casos, os *peers* enviam uma mensagem do tipo *Peer Register* para o novo sucessor, efectuando o mesmo processo que um novo *peer* efectua quando pretende juntar-se ao *overlay*.

A figura 5.6 apresenta um diagrama de sequência no qual é visível a troca de mensagens envolventes na admissão de um novo *peer*. Na figura, o novo *peer*, *A*, envia inicialmente o seu pedido de admissão para o *peer B*. Este, como não é o responsável pela sua admissão, responde com uma mensagem de redirecionamento, contendo o contacto do *peer C*, que considera ser (dos *peers* que conhece) o *peer* responsável pela admissão. O *peer A* repete este processo até que finalmente encontra o *peer* responsável pela sua admissão (D), respondendo este afirmativamente ao pedido de registo.

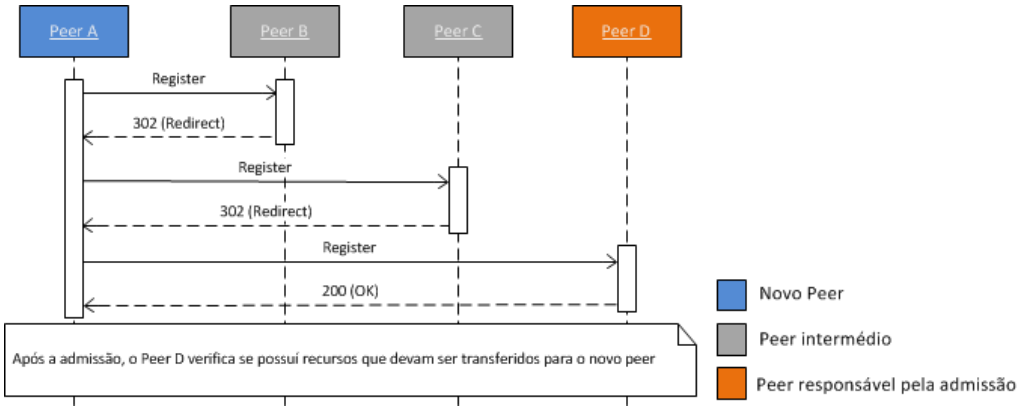

FIGURA 5.6: Exemplo - Admissão de um novo *peer*

Na implementação Chord, o processamento de pedidos P2PSIP para registo de *peers* é todo ele feito na classe *ChordPeer*. A implementação da lógica envolvida na criação e envio de pedidos deste tipo é feita na classe *Join*. É também nessa classe que é feito o processamento das respostas recebidas aos pedidos de registo efectuados.

A classe *Join* implementa a interface *IP2PSIPResponseListener* o que lhe permite ser notificada directamente pela camada P2PSIP da chegada de respostas aos pedidos

efectuados. Assim que o processo de registo é concluído, com ou sem sucesso, a classe *ChordPeer* é notificada através da interface *JoinListener* do resultado da operação. O processo de registo pode ser concluído sem sucesso, se por exemplo o *peer* com o qual se tentou efectuar o registo não responder e o *peer* não possuir informação sobre outros *peers* pertencentes ao *overlay*.

5.2.6 Inserção/remoção e localização de recursos

Para inserir ou remover um recurso do *overlay*, um *peer* deve criar uma mensagem do tipo *Resource Register*, na qual especifica qual o recurso que pretende registar, e a validade do mesmo. A validade do recurso, é o número em segundos, que o registo do recurso deve ser mantido pelo *peer* que o armazenar. Se numa mensagem de registo o valor deste parâmetro for igual a 0, significa que o *peer* pretende remover o recurso do *overlay*.

A criação das mensagem para inserir e remover recursos do *overlay*, assim como a gestão destes processos está implementada nas classes *Put* e *Remove*. Por cada recurso que se pretenda inserir ou remover, é criada uma nova instância de uma destas classes. O processo de inserção ou remoção do recurso no *overlay* é iniciado assim que o método *execute()* da respectiva classe é executado.

Uma vez iniciado o processo, é verificado se o próprio *peer* é o responsável por armazenar o recurso, se for, o recurso é guardado ou removido do gestor de recursos do *peer*, terminando ai o processo de inserção ou remoção do recurso. Caso contrário é criada uma mensagem *Resouce Register*, e através do método *lookup(identificador)* da classe *ChordPeer* é obtido o contacto do *peer* que deverá ser responsável pelo armazenamento do recurso. Após o pedido ser enviado, uma das três seguintes situações pode ocorrer:

- O pedido resulta em *timeout*
- O *peer* aceita o registo do recurso
- O *peer* recusa armazenar o recurso por não ser ele o responsável.

No primeiro caso, em que a resposta ao pedido não chega, originando um timeout, o processo é repetido novamente, existindo um máximo de três tentativas. Após três

tentativas falhadas, o processo termina sem sucesso. Caso o pedido de registo seja aceite pelo *peer*, o processo termina, pois o recurso foi inserido com sucesso no *overlay*. No terceiro caso, em que a inserção é rejeitada pelo facto da mensagem ter sido enviada para um *peer* que não é o responsável, este envia na sua mensagem de resposta, o contacto do *peer* que julga ser o responsável. O processo de registo é reiniciado, sendo desta vez a mensagem de registo do recurso enviada para o contacto retornado.

Sempre que um *peer* recebe um pedido P2PSIP, o processamento do pedido é sempre efectuado na classe *ChordPeer*. Neste caso, o processamento de pedidos para inserir ou remover um recurso, é feito no método *processResourceRegistration* dessa classe. Este método implementa um algoritmo idêntico ao utilizado para a admissão de *peers*. O algoritmo verifica se o *peer* é responsável por armazenar o recurso, e se for, envia uma mensagem de resposta com um código 200 (OK) confirmando a inserção ou remoção do recurso. Se não for, envia uma mensagem de reencaminhamento, do tipo 302 (Redirect), contendo o contacto do *peer* que julga ser o responsável pelo armazenamento do recurso.

A figura 5.7 representa um fluxograma que mostra como um *peer* procede quando recebe um pedido para inserir ou remover um recurso.

Para a localização de recursos no *overlay*, o mecanismo utilizado é idêntico ao mecanismo utilizado para inserir/remover um recurso. O processo de localização de recurso é feito através da classe *Get*, e o seu funcionamento é em tudo idêntico ao da classe *Put*. As principais diferenças são: as mensagens P2PSIP criadas são do tipo *Resource Query*, e existe a possibilidade de serem recebidas respostas com código *404 (Not Found)*. O processamento das respostas ao pedido enviado é igual ao processamento efectuado na classe *Put*, as únicas diferenças são: quando o recurso é encontrado, a mensagem de resposta com codigo *200 (OK)*, possuí a informação relativa ao recurso, e, existe a possibilidade de o recurso não existir no *overlay*, isto acontece quando o *peer* responsável envia uma mensagem de resposta do tipo *404 (Not Found)*.

O processamento de pedidos para localizar um recurso é feito no método *processResourceQuery* da classe *ChordPeer*. O algoritmo implementado neste método é praticamente igual ao do método *processResourceRegistration*, descrito anteriormente.

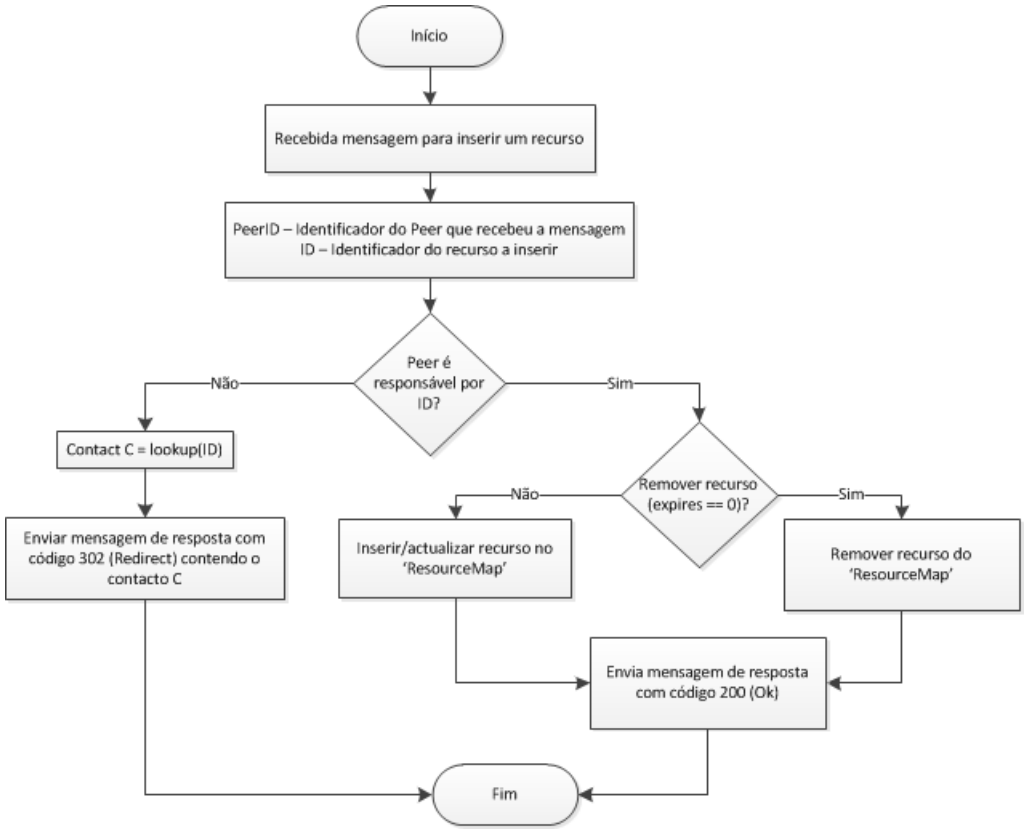

FIGURA 5.7: Fluxograma - Processamento de pedido para registo de um recurso

A figura 5.8 contêm um fluxograma que representa o algoritmo implementado no método *processResourceQuery*, responsável por processar pedidos de localização.

A classe *ChordPeer* implementa as interfaces *PutListener* e *GetListener* o que lhe permite ser notificada pelas classes *Put*, *Remove* e *Get* do resultado das operações de inserção ou localização. Nesta implementação essas notificações, imprimem apenas no output da aplicação o resultado, contudo no futuro podem ser utilizadas para notificar o utilizador, através de uma interface gráfica, do resultado da operação.

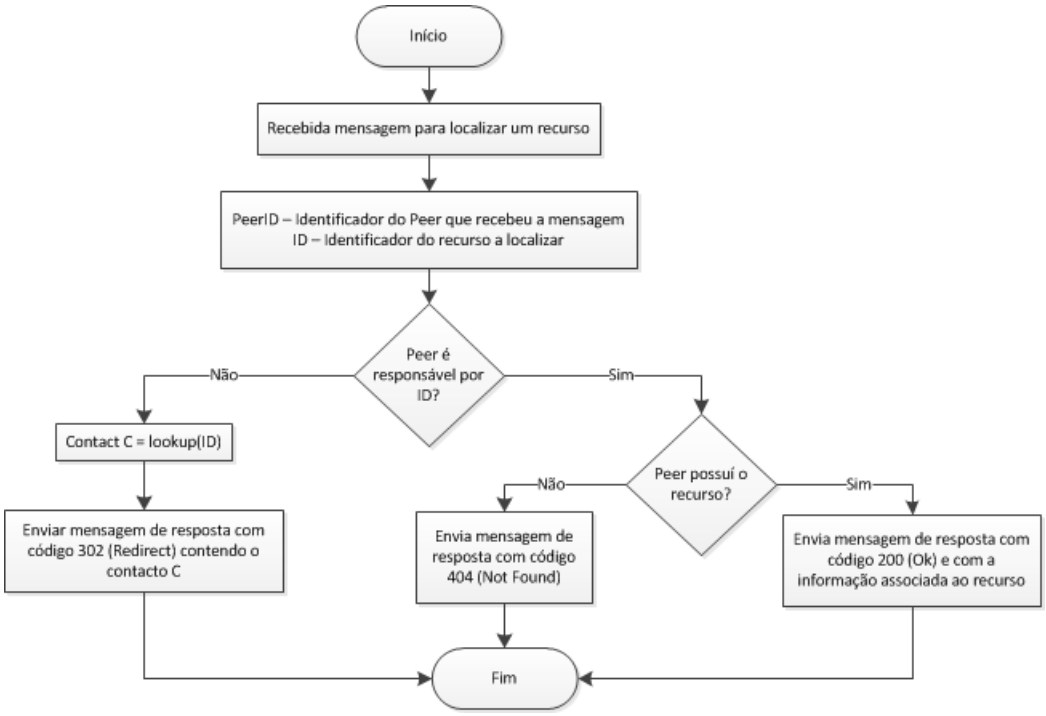

FIGURA 5.8: Fluxograma - Processamento de pedido para localização de um recurso

5.3 EpiChord

O algoritmo EpiChord implementado na aplicação desenvolvida é baseado na descrição do algoritmo feita em [4]. Este algoritmo diferencia-se do Chord tradicional pela utilização de uma *cache* em vez da tradicional *finger table* e também pelo envio de mensagens em paralelo para a localização de recursos.

À semelhança da implementação Chord, nesta implementação as classes base disponibilizadas pela camada DHT foram utilizadas sempre que possível, tendo algumas delas sido extendidas para implementar funcionalidades especificas deste algoritmo. As classes base *Peer*, *ResourceMap* e *EventScheduler* disponibilizadas pela camada DHT foram utilizadas. As duas últimas, como oferecem de base as funcionalidades necessárias, não houve necessidade de as extender, enquanto que a classe *Peer*, que por defeito, implementa e define alguns métodos genéricos, foi extendida, originando a classe *EpiChordPeer*.

A classe *EpiChordPeer* é a classe principal da implementação do EpiChord. Todo o processamento de pedidos P2PSIP que um *peer* EpiChord receba, é processado de acordo com a especificação do algoritmo, nesta classe. O processamento deste tipo de mensagens,

inclui, entre outros, pedidos de admissão de novos *peers* e também pedidos para a inserção de recursos.

A gestão de recursos, da *cache* e a execução de determinadas tarefas periodicamente, foi, como na implementação Chord, implementado em diferentes componentes. Para a gestão de recursos e para o temporizador, foram utilizadas as classes disponibilizadas pela camada DHT. Dos três componentes referidos, foi apenas necessário implementar de raíz, a classe *EpiChordRoutingTable* associada ao componente responsável pela gestão da *cache*.

Como na implementação Chord, cada acção que um *peer* pode executar encontra-se implementada na sua própria classe. Por cada nova acção que um *peer* execute, é criada uma nova instância da classe associada a essa acção. É depois nessas classes que as mensagens com os pedidos P2PSIP são criadas e enviadas.

5.3.1 Manutenção do *Overlay*

A estabilidade do *overlay* pode ser afectada pela entrada de vários *peers* num curto espaço de tempo, ou pela saída abrupta de *peers*. A saída abrupta de *peers* pode fazer com que momentaneamente o espaço de endereçamento pelo qual esses *peers* eram responsáveis não seja atribuído a nenhum outro *peer*. Isto porque se um *peer* não anunciar a sua saída do *overlay* aos seus vizinhos, estes não sabem que o espaço de endereçamento pelo qual o *peer* era responsável é agora da sua responsabilidade. Já a entrada de vários *peers* cuja suas posições no anel são aproximadamente a mesma, pode provocar algumas inconsistências nas tabela de encaminhamento dos próprios *peers* assim como dos seus vizinhos. Por isso, e para manter a estabilidade do *overlay* é necessário que a informação que cada *peer* possuí sobre os seus vizinhos directos (sucessor e antecessor) esteja actualizada. A manutenção da informação sobre os vizinhos é feita pelo processo de estabilização do algoritmo.

Os autores do EpiChord definem dois processos de estabilização: um para garantir a estabilidade local, ou seja, manter actualizada a informação sobre os vizinhos, e um outro para detectar e corrigir inconsistências globais que possam originar ciclos.

Destes dois processos de estabilização, apenas o primeiro foi implementado, pois é o mais importante, e porque segundo os autores, a ocorrência de ciclos é altamente improvável pelo que para este trabalho não se justificava a implementação desse mecanismo de estabilização.

O processo de estabilização implementado, é executado periodicamente por cada *peer*, onde são enviadas mensagens P2PSIP do tipo *PeerQuery* para os vizinhos directos, sucessor e antecessor. Através do envio dessas mensagens o *peer* consegue determinar se os seus vizinhos ainda se encontram no *overlay*. E também, através da análise da informação de encaminhamento contida nas mensagens de resposta, detectar se estes se mantêm como os seus vizinhos directos. As mensagens P2PSIP trocadas entre os *peers*, contêm sempre informação sobre o sucessor e antecessor do *peer* que envia a mensagem, assim como alguma informação da sua *cache*.

Para além do processo de estabilização, que é executado periodicamente, no EpiChord, sempre que um *peer* recebe uma mensagem, analisa a informação de encaminhamento que esta contem. Através dessa análise é possível detectar a existência de novos sucessores/antecessores. Nestes casos, se a informação sobre o novo sucessor ou antecessor não tiver sido obtida directamente, mas sim através de outros *peers*, o *peer* não pode considerar automaticamente o novo *peer* como seu sucessor. Isto porque, a informação que recebeu pode estar desactualizada, por isso antes de considerar o novo *peer* como seu sucessor, necessita de comprovar que este ainda está no *overlay*, inicializando o processo de registo no *overlay* através do envio de uma mensagem P2PSIP do tipo *Peer Register* para esse novo *peer*. Só após ser recebida uma resposta positiva é que esse passa a ser o novo sucessor.

A figura 5.9 mostra um exemplo no qual o *peer* com identificador B, possuí momentaneamente informação desactualizada sobre quem é o seu verdadeiro sucessor. Assim que o processo de estabilização periódico for executado, o *peer* irá detectar a existência de um novo sucessor, corrigindo a informação desactualizada que possuía. Contudo, este exemplo mostra uma situação em que o processo de estabilização periódico não é necessário para corrigir a informação. Neste exemplo, o *peer* E inicializa a localização de um recurso, enviando para o *peer* B uma mensagem de localização. Na mensagem de localização enviada, é enviada informação sobre a sua cache, enviando neste caso informação sobre o *peer* C e B. Analisando a informação de encaminhamento contida na mensagem recebida, B,

detecta que existe um *peer*, C, que não conhecia e que está mais próximo de si (no sentido dos ponteiros do relógio) do que o seu sucessor actual. Isto significa que um novo *peer*, C, se ligou entre o A e B, sendo C o novo sucessor de B. Visto que, B tomou conhecimento da existência de C através de outro *peer*, necessita de verificar se C está realmente no *overlay* antes de o considerar como o seu novo sucessor. Para isso, inicializa o processo de registo junto do novo sucessor, e, assim que o C responde, B actualiza a informação sobre o seu sucessor, passando esta a estar actualizada.

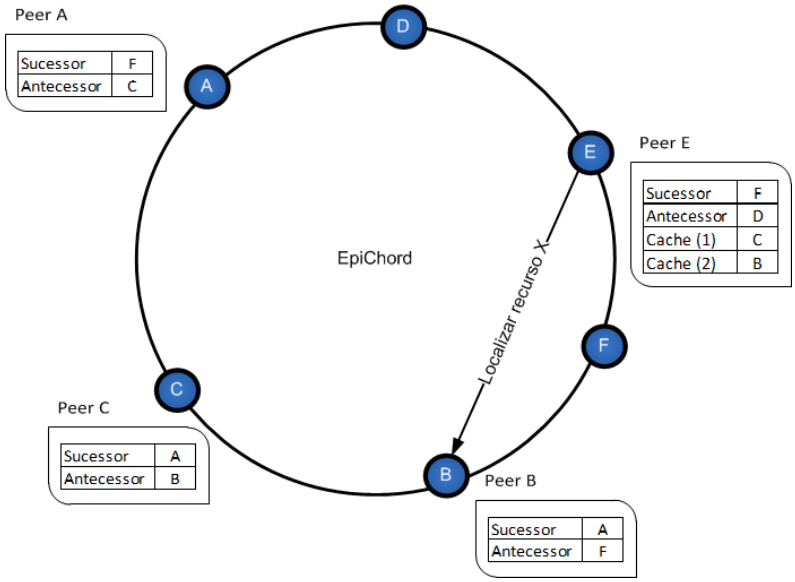

FIGURA 5.9: Exemplo no qual o *peer B* actualiza a sua tabela de encaminhamento com base no trafêgo que recebe

Na implementação EpiChord, os processos de estabilização periódicos foram implementados nas seguintes classes:

- *Stabilization* - Classe utilizada para verificar o estado do sucessor, verificando se este se mantêm o seu sucessor
- *CheckPredecessor* - Classe utilizada para verificar o estado do antecessor, verificando se este se mantêm o seu antecessor
- *KeepAlive* - Classe utilizada para que o *peer* periodicamente se registe novamente no seu sucessor, actualizando o parâmetro *lifetime* associado ao seu registo
- *ResourceTransfer* - Classe utilizada sempre que é necessário efectuar a transferência de recursos entre *peers*.

Cada uma dessas classes é responsável pelo envio dos pedidos P2PSIP associados ao tipo de tarefa a executar, assim como efectuar o processamento das respostas recebidas. De forma idêntica à implementação Chord, a classe *EpiChordPeer* é a responsável por inicializar e executar estas tarefas. Recorre também a um temporizador de eventos, no qual regista as tarefas das quais deseja ser notificado quando o período de tempo associado a cada tarefa expirar.

5.3.2 Gestão da tabela de encaminhamento

A tabela de encaminhamento de um *peer* EpiChord é composta por apontadores para os seus vizinhos directos, sucessor e antecessor, e também uma *cache*, que possuí informação sobre diversos *peers* do *overlay*.

A informação contida na *cache* é obtida através da análise da informação de encaminhamento do tráfego P2P que o *peer* vai recebendo. Cada entrada da *cache*, para além da informação básica sobre o *peer* (identificador, endereço IP e porta), possuí um outro parâmetro, *lifetime* que guarda o instante de tempo em que a informação foi obtida. Para que a informação mantida na *cache* esteja minimamente actualizada, e evitar que esta cresça em demasia, os autores do EpiChord, recomendam que se utilizem dois parâmetros para decidir quando uma entrada deve ser removida. Os parâmetros recomendados são: o número de vezes que o *peer* falhou, e o valor do parâmetro *lifetime*. Neste trabalho, a implementação do EpiChord, utiliza ambos os parâmetros, contudo, o número de vezes que o *peer* falhou não é utilizado para remover entradas da *cache*. Em vez disso, estas ficam em segundo plano, sendo apenas utilizadas se não existirem outras entradas em melhores condições.

Toda a gestão da tabela de encaminhamento, é feita na classe *EpiChordRoutingTable*, sendo a informação relativa a cada *peer*, identificador, endereço IP, porta e *lifetime* armazenada em instâncias da classe *EpiContact*. Periodicamente, a classe *EpiChordPeer*, através do seu temporizador de eventos inicializa o processo de validação da *cache*. Esse processo consiste em remover todas as entradas cujo valor do *lifetime* (em segundos) seja superior a um determinado limite. Por defeito, o valor máximo do *lifetime* é de 160 segundos, contudo, este é um dos vários parâmetros configuráveis da aplicação.

5.3.3 Algoritmo de localização

No EpiChord, da mesma forma que no Chord, cada *peer* é responsável por armazenar a informação relativa aos recursos cujos identificador estejam compreendidos entre o seu identificador e o identificador do *peer* que o antecede no *overlay*.

O algoritmo de localização do EpiChord é baseado no envio de várias mensagens de localização em paralelo, utilizando um mecanismo de encaminhamento iterativo.

Quando um *peer* deseja iniciar o processo de localização de um recurso ou *peer*, necessita de enviar P mensagens de localização. As P mensagens são enviadas para os *peers* da *cache*, que se encontram mais próximos do identificador do *peer* ou recurso a localizar. É enviada uma mensagem para o sucessor do identificador e as restantes P-1 mensagens são enviadas para os antecessores do identificador. Para além do envio das P mensagens, são definidos dois apontadores para o melhor sucessor e antecessor do identificador que responderam. Inicialmente, como ainda nenhum *peer* respondeu, referenciam o próprio *peer*, mas à medida que as respostas vão chegando, os apontadores são actualizados consoante o *peer* que respondeu for melhor sucessor ou antecessor. O objectivo destes apontadores é evitar o envio de mensagens para *peers* que não estão mais próximos do destino.

Quando um *peer* recebe uma mensagem de localização, se for o responsável pelo identificador, responde positivamente, enviando a informação sobre o recurso, caso seja uma localização de recursos e este exista. Se não for o responsável, o *peer* consulta a sua *cache* e envia uma mensagem de resposta contendo L contactos que deverão ser contactados. A forma como os L contactos são escolhidos depende da posição do *peer* relativamente ao identificador e ao *peer* que o pretende localizar.

O fluxograma da figura 5.10 mostra a forma como um *peer* obtém os L melhores contactos para localizar um determinado identificador. O algoritmo representado na figura, foi implementado na classe *EpiChordPeer*, e é utilizado sempre que é necessário obter a lista dos L melhores contactos para um identificador. Este algoritmo é utilizado em todas as situações em que um *peer* sabe que o identificador não é da sua responsabilidade, e que necessita de obter através da sua *cache*, uma lista com os L *peers* que deverão ser responsáveis. Por exemplo, quando uma mensagem de localização de um recurso ou *peer* é recebida, se o *peer* que a recebe não for o responsável, executa este algoritmo de modo

a obter os L contactos que deverão ser contactados para localizar o recurso ou *peer*. Neste exemplo, o contacto do *peer* seria enviado numa mensagem de resposta com o código 302 (Redirect).

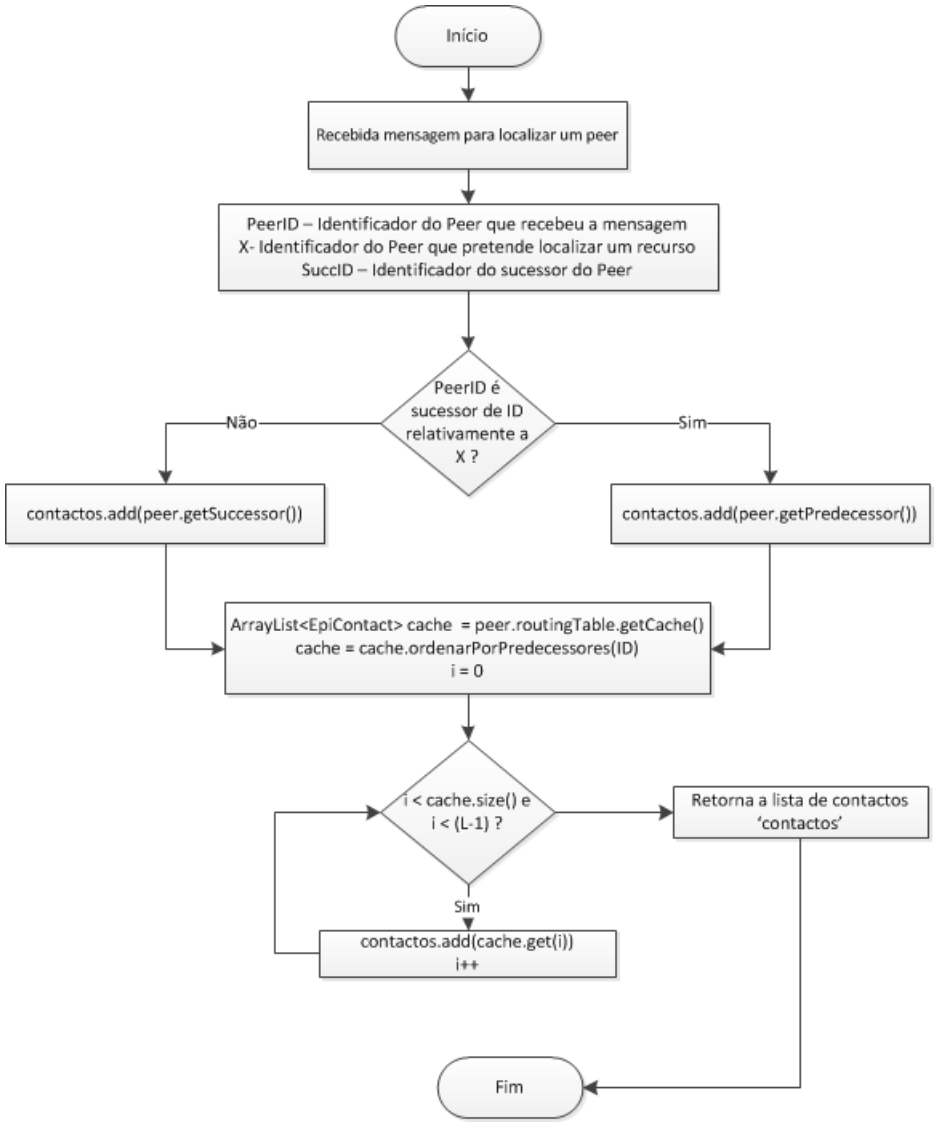

FIGURA 5.10: Fluxograma - Algoritmo de localização do EpiChord

Assim que as mensagens de resposta vão sendo recebidas, contendo L contactos, o *peer* verifica se os contactos que recebeu pertencem a *peers* que se encontram mais próximos no espaço de endereçamento, do identificador. Comparando-os com os apontadores para o melhor sucessor e antecessor conhecidos para o identificador. Caso algum desses *peers* seja melhor, é lhe enviada uma mensagem de localização. O processo é repetido até que o *peer* responsável pelo identificador é encontrado.

Este algoritmo é utilizado em diversas situações, como por exemplo na inserção ou localização de recursos, através das classes *ParallelPut* e *ParallelGet*. Os parâmetros P e L deste algoritmo, que definem o número de mensagens em paralelo a enviar, e o número de contactos a retornar, são parâmetros configuráveis da aplicação desenvolvida.

5.3.4 Admissão de *peers*

A admissão de novos *peers* ao *overlay* é feita de forma semelhante à do Chord. O *peer* responsável pelo espaço de endereçamento ao qual o identificador do novo *peer* pertence é quem deve tratar do registo do novo *peer*.

A principal diferença relativamente à implementação Chord, tem a ver com o redirecionamento. Quando um *peer* não é o responsável por admitir um outro *peer* no *overlay*, envia lhe uma mensagem de redirecionamento do tipo 302 Redirect na qual envia uma lista de L contactos que o novo *peer* deve tentar contactar para se juntar ao *overlay*. Os L contactos retornados na mensagem de redirecionamento, são obtidos de acordo com o algoritmo, ilustrado no fluxograma da figura 5.10.

Uma vez encontrado o *peer* responsável pela sua admissão, este envia na mensagem de resposta do tipo 200 (OK) informação de encaminhamento, contendo todas as entradas da sua *cache*. Desta forma, o novo *peer* começa com um número de entradas na *cache*, que à partida devem ser suficientes, para lhe permitir obter no imediato um bom desempenho na localização de *peers* ou recursos.

Assim como no Chord, a admissão do novo *peer*, pode significar que alguns dos recursos armazenados pelo *peer* que o admitiu, passam a ser da responsabilidade do novo *peer*. Nesse caso, é iniciado o processo de transferência de recursos.

Na implementação EpiChord, o processamento de pedidos P2PSIP para registo de *peers* é todo ele feito na classe *EpiChordPeer*. A classe *Join* é a responsável por efectuar o registo do *peer* no *overlay*, enviando os pedidos e processando as respectivas respostas.

A classe *Join* implementa a interface *IP2PSIPResponseListener* o que lhe permite ser notificada directamente pela camada P2PSIP da chegada de respostas aos pedidos efectuados. Assim que o processo de registo é concluído, com ou sem sucesso, a classe

EpiChordPeer é notificada através da interface *JoinListener* do resultado da operação. O processo de registo pode ser concluído sem sucesso, se por exemplo o *peer* com o qual se tentou efectuar o registo não responder e o *peer* não possuir informação sobre outros *peers* pertencentes ao *overlay*.

5.3.5 Inserção/remoção e localização de recursos

Na implementação do algoritmo EpiChord, a forma como são inseridos, removidos ou localizados recursos no *overlay* é semelhante à do *overlay* Chord. Os tipos de mensagens criados são os mesmos, as principais diferenças tem a ver com o envio de mensagens em paralelo, e com o recurso à informação contida na *cache* em vez da *finger table*.

Sempre que um *peer* necessita de inserir, remover ou localizar um recurso, tem de conseguir encontrar o *peer* responsável pelo espaço de endereçamento ao qual pertence o identificador do recurso. Para isso, utiliza o algoritmo de localização do EpiChord e envia a mensagem de inserção, ou localização para P *peers* em paralelo.

Neste trabalho, a implementação deste algoritmo, para inserir ou localizar recursos, foi feita nas classes *ParallelPut* e *ParallelGet*. Estas classes são responsáveis por implementar o algoritmo de localização do EpiChord, gerindo todo esse processo, e utilizando para o envio de cada uma das mensagens instâncias das classes *Put* e *Get*.

As classes *Put* e *Get* são responsáveis pelo envio de uma mensagem de inserção/remoção ou localização para um único *peer*, processando as mensagens de resposta associadas ao pedido que enviam. As classes responsáveis pela gestão global do processo, *ParallelPut* e *ParallelGet* são notificadas da resposta de um *peer*, obtida pelas instâncias das classes *Put* e *Get*, através das interfaces que implementam.

Durante o processo de inserção ou localização de recursos no *overlay*, os *peers* que são contactados, respondem a mensagens de redirecionamento, do tipo *302 (Redirect)* quando não são os responsáveis pelo recurso. Respondem com uma mensagem do tipo *200 (OK)*, quando o *peer* contactado é responsável pelo recurso, retornando o seu valor, no caso de ser uma mensagem de localização. Os tipos de mensagem *404 (Not Found)* são enviados

pelos *peers* responsáveis por recursos, quando estes recebem uma mensagem de localização para um determinado recurso, e este não existe.

Da mesma forma que na implementação Chord, o processamento de pedidos para localizar um recurso é feito no método *processResourceQuery* da classe *EpiChordPeer*. O algoritmo implementado neste método é praticamente igual ao do método *processResourceRegistration*, descrito anteriormente.

A figura 5.11 contêm um fluxograma que representa o algoritmo implementado no método *processResourceRegistration*, responsável por processar pedidos para a inserção, ou remoção de recursos.

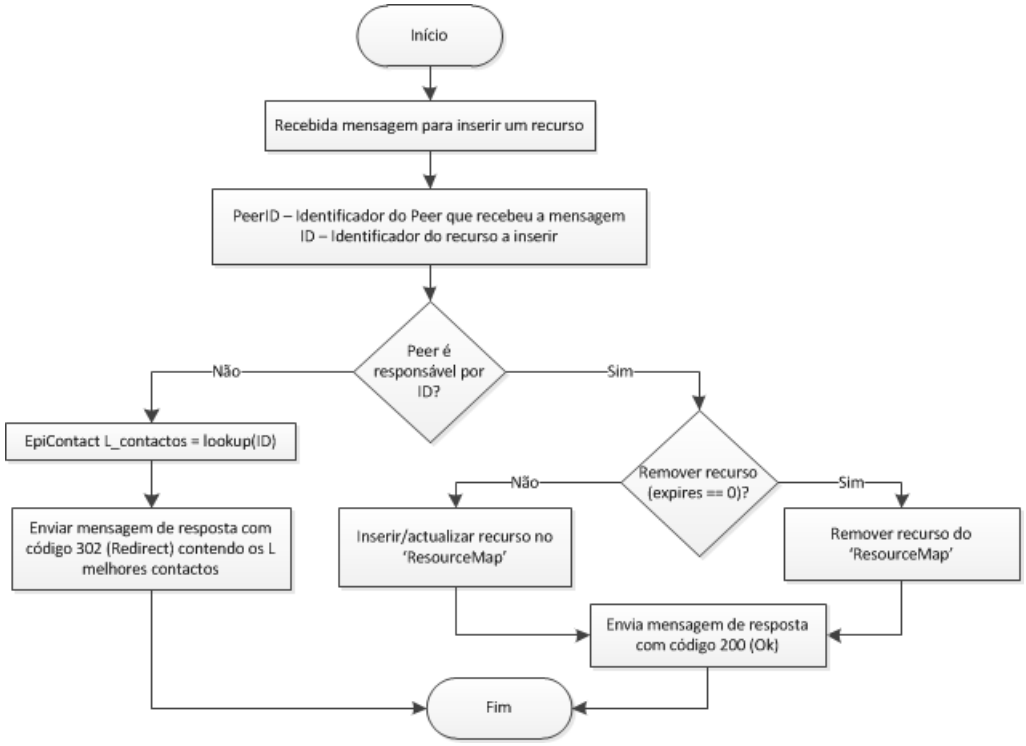

Figura 5.11: Fluxograma - Processamento de pedido para o registo de um recurso

A classe *EpiChordPeer* implementa as interfaces *PutListener* e *GetListener* o que lhe permite ser notificada pelas classes *ParallelPut* e *ParallelGet* do resultado das operações de inserção ou localização. Nesta implementação essas notificações, imprimem apenas no output da aplicação o resultado, contudo no futuro podem ser utilizadas para notificar o utilizador, através de uma interface gráfica, do resultado da operação.

5.4 Cliente

O cliente no contexto deste trabalho, é um elemento simples, que não participa no *overlay*, mas que usufruí dos serviços que este disponibiliza. Assim como um *peer*, um cliente necessita de conhecer um ou vários *peers* pertencentes ao *overlay*, denominados por *peers* de *bootstrap*, de forma a conseguir estabelecer uma ligação com um *peer*. Após possuir uma ligação com um *peer* pertencente ao *overlay*, o cliente pode inserir, remover ou localizar recursos, necessitando para isso, de enviar o seu pedido para um *peer*.

A classe principal do cliente é a classe *Cliente*, e o seu funcionamento é simples, possuí alguns métodos que permitem a inserção, remoção e localização de recursos no *overlay*. Como o cliente não participa activamente no *overlay*, o único tipo de mensagem que recebe, que não é uma resposta a um pedido por si efectuado, é a mensagem do tipo *Peer Leave*. Este tipo de mensagem é recebido quando o *peer* ao qual o cliente estava ligado, informa o cliente de que vai abandonar o *overlay*, enviando na mensagem, contactos de outros *peers* pertencentes ao *overlay*. Com essa informação, o cliente inicializa novamente o processo de admissão, enviando uma mensagem para um dos contactos retornados na mensagem recebida. Os contactos retornados são sempre guardados, de modo a que, se posteriormente, o *peer* abandonar o *overlay* sem aviso, o cliente possa, após detectar a ausência por timeout, estabelecer nova ligação com um outro *peer* do *overlay*.

Todas as acções que um cliente pode executar, como o processo de estabelecer uma ligação com um *peer*, inserir ou localizar um recurso, encontram-se implementadas na suas próprias classes, existindo por isso, para além da classe *Cliente* as classes *Join*, *Put* e *Get*.

5.4.1 Inserção/remoção e localização de recursos

Como um cliente não participa activamente no *overlay*, para usufruir dos serviços que este disponibiliza, necessita apenas de enviar para um *peer*, um pedido. O *peer*, independentemente do algoritmo em utilização (Chord ou EpiChord), é o responsável por executar o pedido efectuado pelo cliente, retornando lhe apenas o resultado da sua operação. O

cliente nunca é informado do progresso da sua operação, isto é, se houve reencaminhamento de mensagens, timeouts, etc.. Todo o processo é feito de forma transparente, após executar um pedido, o cliente recebe do *peer* apenas o resultado final da operação.

A figura 5.12 mostra um exemplo, em que um cliente enviou um pedido de inserção de um recurso para um *peer*. Na figura é possível verificar que o *peer* efectuou o pedido de inserção, e que este foi redirecionado várias vezes, até que por fim foi inserido no *overlay*. É visível na figura, que o cliente apenas recebe o resultado final da operação quando esta é concluída, não se apercebendo de todas as trocas de mensagens que o seu pedido originou.

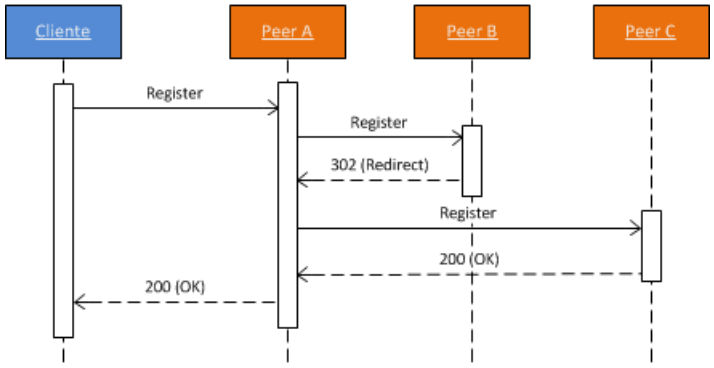

FIGURA 5.12: Fluxograma - Processamento de pedido para localização de um recurso

Na implementação do cliente, a inserção ou remoção de recursos é feita na classe *Put*, para localizar um recurso é criada uma nova instância da classe *Get*.

Capítulo 6

Testes e Resultados

Os testes efectuados focam-se no desempenho dos algoritmos Chord e EpiChord em situações em que o volume de tráfego de localização de recursos é muito grande. Onde cada *peer* efectua aproximadamente duas localizações por segundo.

Os parâmetros analisados nos testes aos algoritmos Chord e EpiChord, para medir o desempenho destes na localização de recursos foram:

- Tempo - Tempo médio necessário para localizar um recurso no *overlay*
- Número de saltos - Número de saltos necessários para se localizar um recurso

6.1 Ambiente de teste

Os primeiros testes, foram feitos numa máquina na qual se utilizou o *Common Open Research Emulator* (CORE)[24], tendo posteriormente, sido utilizado o *cluster* SeARCH[25] do Departamento de Informática da Universidade do Minho.

O *Common Open Research Emulator* (CORE)[24], é uma ferramenta 'que permite emular redes, redes essas que podem se ligar a outras redes emuladas e/ou redes reais'[24]. Os testes efectuados no CORE, serviram para validar a implementação dos algoritmos.

Foi criada no CORE uma topologia composta por 3 *Routers*, 3 *Switches* e 31 *Hosts* (*peers* ou clientes). A figura 6.1 mostra como os vários componentes da topologia se

encontram ligados.

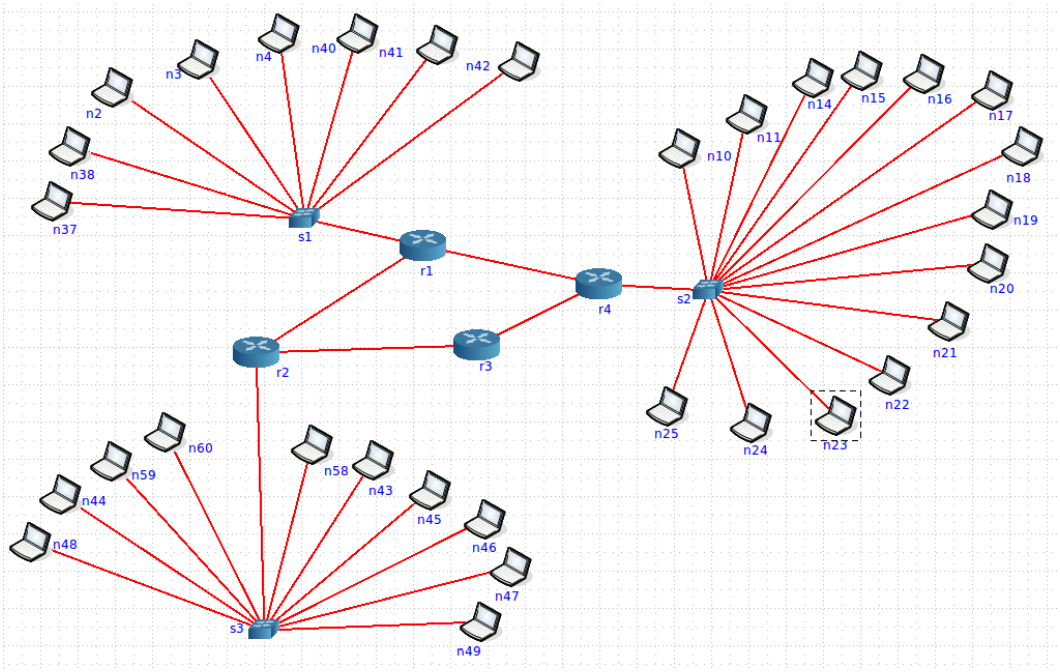

FIGURA 6.1: Topologia utilizada para efectuar os testes

Um dos objectivos dos testes efectuados era a validação da implementação dos algoritmos Chord e EpiChord, por isso, foi necessário configurar a topologia de modo a que as características das ligações dos *peers* fossem idênticas às características das ligações dos *peers* nos testes efectuados pelos autores do EpiChord. Para isso, nas ligações entre os Hosts e os Switches da topologia utilizada, foi definido um atraso de 80ms, fazendo com que o RTT(*Round Trip Time*) médio entre *peers* fosse igual ao valor utilizado nos testes efectuados pelos autores do EpiChord: 0.16seg.

Devido às limitações da máquina na qual o CORE foi executado, o número máximo de *peers* suportado era 31. Tendo sido, por isso, necessário recorrer ao SeARCH para efectuar testes com um maior número de *peers*. No SeARCH, por diversos motivos, não foi possível instalar e utilizar o CORE, pelo que foi necessário encontrar uma outra solução.

A solução encontrada foi utilizar vários nós do *cluster*, nos quais foram executadas múltiplas instâncias da aplicação desenvolvida sem recorrer ao CORE para emular uma topologia de rede. Enquanto que numa topologia real, à partida cada *peer* possuí um

endereço IP diferente, neste caso, todas as instâncias da aplicação em execução num mesmo nó partilhavam o mesmo endereço IP. Por isso, em vez de ser o endereço IP a variar de *peer* para *peer*, variou-se a porta a utilizar.

Dos vários parâmetros a analisar nos testes, o único parâmetro que seria afectado por esta solução era o tempo de localização de recursos, pois o número de saltos como se refere ao número de saltos na rede de *overlay* e não na rede física, não é afectado. O tempo de localização seria afectado, pois as ligações entre os *peers* eram muitas vezes feitas em *localhost*, pelo que o atraso na entrega das mensagens era praticamente nulo. Para solucionar este problema, a aplicação desenvolvida, foi modificada de modo a simular atrasos, ou seja, quando uma mensagem é recebida, em vez de ser processada imediatamente, a aplicação efectua um pequeno compasso de espera, de forma a simular o atraso que a mensagem teria numa rede real. O valor do atraso é passado para a aplicação através de dois parâmetros, *rttMin* e *rttMax*, que, uma vez tendo valores diferentes, faz com que o atraso em cada mensagem recebida, seja um valor aleatório entre *rttMin* e *rttMax*.

Antes de serem efectuados testes com um maior número de *peers*, foi necessário verificar se os resultados obtidos originalmente com 31 *peers* no CORE, eram idênticos aos resultados obtidos com 31 *peers* no *cluster* utilizando a solução descrita, com os mesmos atrasos utilizados no CORE. Os resultados obtidos em ambos os testes foi semelhante, pelo que se concluiu que a solução encontrada funciona.

Os restantes testes, foram efectuados no *cluster* com 100 e 200 nós (entre *peers* e clientes). Os testes efectuados com 100 nós recorreram a um único nó do *cluster*, com 8GB de memória RAM e 4 CPUs. Com 200 nós, foi necessário utilizar 2 nós do *cluster*, com as mesmas características enunciadas anteriormente.

Para além das configurações dos atrasos da topologia, a aplicação desenvolvida foi modificada para que cada *peer* efectuasse cerca de dois *lookups* por segundo. A lista com os recursos que cada *peer* deve tentar localizar, foi criada e armazenada em ficheiros previamente, de forma a que cada *peer* efectuasse sempre a mesma sequência de *lookups* em cada teste. Assim, os resultados obtidos em testes diferentes, podem ser comparados, uma vez que não há alterações nos *lookups* a efectuar que possam alterar os resultados.

Nos testes efectuados, alguns *peers* foram denominados como *peers* de *bootstrap*, isto é, todos os *peers* ao serem inicializados recebiam uma lista com os *peers* de *bootstrap*, aos quais se deveriam tentar ligar para se juntarem ao *overlay*. Cada *peer* ao inicializar estabelecia uma ligação com o primeiro *peer* da lista, sendo os outros utilizados caso o *peer* seleccionado não respondesse.
Para evitar que um servidor de *bootstrap* fosse o mais utilizado pelos restantes *peers*, foi utilizada uma técnica de *Round Robin*, sendo rodada a lista que cada *peer* recebe com os *peers* de *bootstrap*, desta forma preveniu-se que um *peer* ficasse inicialmente sobrecarregado devido a ser o primeiro da lista, e todos os outros estarem a tentar juntar-se ao *overlay* através dele.

A inicialização dos *peers* e clientes de cada teste foi efectuada recorrendo a um *script* desenvolvido neste trabalho, o qual inicializa a aplicação JAVA de cada *peer* ou cliente com os vários parâmetros, de acordo com o teste a efectuar. Em primeiro lugar são inicializados os *peers* de *bootstrap*, de seguida os restantes *peers*. Se o teste utilizar clientes, estes são os últimos a ser inicializados.

As aplicações JAVA inicializadas, recebem vários parâmetros, que permitem entre outras coisas, configurar o algoritmo DHT em utilização. Um desses parâmetros é utilizado para definir quanto tempo a aplicação deve esperar até iniciar os testes de localização de recursos. Visto que os testes que foram efectuados se focam no desempenho dos algoritmos na localização de recursos, o valor desse parâmetro foi definido em 90s. Desta forma quando os testes são inicializados o *overlay* encontra-se estabilizado, sendo por isso os resultados mais fiáveis.

6.2 Validação da Implementação

Para validar a implementação dos algoritmos Chord e EpiChord, foram realizados testes com um *overlay* formado por 100 e 200 *peers*.

Para estes testes foram escolhidos seis *peers* para actuarem como *peers* de *bootstrap*. Isto é, é a partir desses *peers* que todos os outros se juntam ao *overlay*. Os parâmetros dos algoritmos Chord e EpiChord utilizados neste teste, foram os mesmos que os autores

do EpiChord utilizaram nos seus testes, contudo, alguns parâmetros não puderam ser utilizados por não serem referidos nos testes do EpiChord, como por exemplo o número de entradas da *finger table* do Chord.

No caso do algoritmo Chord, foi definido um minuto (60seg) como o período de estabilização, isto significa que os *peers* verificam a cada minuto o estado do seus vizinhos directos (sucessor e antecessor). Para além do período de estabilização, foi também definido o período de tempo em que as entradas na *finger table* devem ser verificadas. O valor deste parâmetro não se encontra especificado nos testes efectuados ao EpiChord, pelo que foi definido como 70seg, valor pertencente à gama de valores recomendados pelos autores da implementação Chord do dSIP [21]. O último parâmetro do algoritmo Chord a configurar é o número de entradas na *finger table*, sendo que os autores da implementação Chord do dSIP [21] recomendam que se utilize 16 entradas na tabela para redes pequenas e 32 para redes maiores, tendo sido este último, o valor utilizado nos testes efectuados.

Quanto à implementação EpiChord, foi utilizado o mesmo valor do Chord (60seg) para o período de estabilização, este valor, é o valor indicado pelos autores do EpiChord nos testes que estes efectuaram. Nesses mesmos testes, definem também que as entradas na *cache* devem ter um *lifetime* máximo de 120s, tendo por isso sido este o valor utilizado nos testes efectuados. Relativamente ao nível de paralelismo na localização de recursos, foram efectuados testes com três níveis diferentes de paralelismo, tendo sido testadas as seguintes combinações: P=1 L=1, P=3 L=3 e P=5 L=3. Onde o parâmetro P indica o número de mensagens em paralelo a enviar para iniciar o processo de localização de um recurso ou *peer*, e L o número de contactos que cada *peer* envia nas suas mensagens de redirecionamento.

Um outro parâmetro, comum a ambos os algoritmos, definido pelos autores do EpiChord nos seus testes, é o valor do *timeout*, tendo estes definido 0.5seg. Contudo, com esse valor de *timeout*, o número de perdas de mensagens era demasiado elevado, pelo que nos testes efectuados o valor definido para o *timeout* foi de 5seg.

A razão pela qual o valor utilizado pelos autores do EpiChord para o *timeout* parece não funcionar na prática deve-se a diversos factores, como o facto de esse valor ter sido utilizado em simulação e não num ambiente real. O facto de que neste trabalho o EpiChord ter sido implementado sobre um outro protocolo, o SIP, e que, apesar de a aplicação

ter sido desenvolvida para suportar concorrência (*multi-threading*), a própria *stack* SIP utilizada definia como valor óptimo de *timeout* 32seg. Por isso, neste trabalho o valor de *timeout* utilizado foi de 5 segundos.

As tabelas 6.1 e 6.2 contêm os resultados dos testes efectuados com 100 e 200 *peers*. Através da analise dos resultados das tabelas 6.1 e 6.2, e comparando-os com os resultados

	Chord	**EpiChord (P=1 L=1)**	**EpiChord (P=3 L=3)**	**EpiChord (P=5 L=3)**
Tempo de *lookup* médio(seg)	0,562	0,164	0,141	0,140
Número de saltos médio	4,124	1,163	1,015	1,004

TABELA 6.1: Resultados do teste de *lookup* intensivo com 100 *peers*

	Chord	**EpiChord (P=1 L=1)**	**EpiChord (P=3 L=3)**	**EpiChord (P=5 L=3)**
Tempo de *lookup* médio(seg)	0,631	0,223	0,162	0,159
Num. Saltos	4,531	1,499	1,081	1,047

TABELA 6.2: Resultados do teste de *lookup* intensivo com 200 *peers*

obtidos pelos autores do EpiChord nos seus testes (Figura 11 e 12 de [4]) é possível verificar que a implementação dos algoritmos Chord e EpiChord parece estar correcta. Contudo, o número de saltos médio na localização de recurso do algoritmo Chord, é ligeiramente superior na nossa implementação. O motivo para esse aumento na nossa implementação, poderá dever-se ao número de entradas da finger table, pois enquanto nos nossos testes foram utilizadas *finger tables* com 32 entradas, nos testes efectuados pelos autores do EpiChord, esse valor não é mencionado.

Para além de comprovarem o funcionamento das implementações, estes resultados mostram também que a utilização do EpiChord, permite melhorar o desempenho da localização de recursos, melhorando em cerca de 72% o tempo de localização de recursos, assim como melhora cerca de 73% em média o número de saltos necessários para se localizar um recurso.

6.3 Peers com limitações

Para verificar se um *overlay* beneficia da existência de uma hierarquia composta por *peers* e clientes, onde os clientes podem ser *peers* com menores capacidades, foram efectuados testes onde se introduziram *peers* com limitações ao nível da conectividade. De forma a que a comparação dos resultados fosse mais justa, estes testes foram efectuados nas mesmas condições dos testes efectuados em 6.2, onde a única diferença existente tem a ver com as limitações introduzidas em alguns dos *peers* da topologia.

As limitações introduzidas na conectividade de alguns *peers*, foram simplesmente no atraso das suas ligações foi alterado de 80ms para 200ms. Desta forma, o RTT dos *peers* do *overlay* varia entre os 0.16 e os 0.40 seg. Para verificar o impacto deste tipo de *peers* no *overlay*, foram criados diferentes cenários, onde se foram introduzindo mais *peers* com limitações, de modo a avaliar qual o impacto destes no desempenho do *overlay* na localização de recursos. Os cenários criados neste teste foram os seguintes:

- Cenário 1 - Foram colocadas limitações na conectividade de 16% dos Peers, correspondente a 16 *peers* em 100, e 32 *peers* num *overlay* de 200.
- Cenário 2 - Foram colocadas limitações na conectividade de 33% dos Peers, correspondente a 33 *peers* em 100, e 66 *peers* num *overlay* de 200.
- Cenário 3 - Foram colocadas limitações na conectividade de 50% dos Peers, correspondente a 50 *peers* em 100, e 100 *peers* num *overlay* de 200.

As tabelas 6.3 e 6.4 contêm os resultados dos testes efectuados num *overlay* com 100 e 200 *peers*, respectivamente. Analisando os dados das tabelas 6.3 e 6.4, e comparando-os com os das tabelas 6.1 e 6.2 é perceptível, que a existência de *peers* com limitações na conectividade influência o desempenho do *overlay* na localização de recursos. O gráfico da figura 6.2 mostra a média dos tempos de localização deste teste assim como os valores dos testes da validação da implementação, nos quais não existiam *peers* com limitações.

Depois de analisadas as tabelas e os gráficos, concluiu-se, que como seria espectável, foi no cenário 3 que se verificou o maior aumento no tempo da localização de recursos, pois é neste cenário que se encontra o maior número de *peers* com limitações. O valor médio

	Chord	**EpiChord (P=1 L=1)**	**EpiChord (P=3 L=3)**	**EpiChord (P=5 L=3)**
		Cenário 1 - 16% *Peers* Limitados		
Tempo de *lookup* médio(seg)	0,739	0,240	0,196	0,190
Número de saltos médio	4,116	1,227	1,019	1,019
		Cenário 2 - 33% *Peers* Limitados		
Tempo de *lookup* médio(seg)	0,881	0,250	0,219	0,204
Número de saltos médio	4,062	1,147	1,009	1,004
		Cenário 3 - 50% *Peers* Limitados		
Tempo de *lookup* médio(seg)	1,037	0,297	0,258	0,245
Número de saltos médio	4,105	1,161	1,014	1,006

Tabela 6.3: Resultados do teste de *lookup* intensivo com 100 *peers* - Cenário 1, 2 e 3

	Chord	**EpiChord (P=1 L=1)**	**EpiChord (P=3 L=3)**	**EpiChord (P=5 L=3)**
		Cenário 1 - 16% *Peers* Limitados		
Tempo de *lookup* médio(seg)	0,823	0,277	0,211	0,203
Número de saltos médio	4,507	1,456	1,072	1,062
		Cenário 2 - 33% *Peers* Limitados		
Tempo de *lookup* médio(seg)	1,015	0,340	0,247	0,245
Número de saltos médio	4,5	1,480	1,090	1,044
		Cenário 3 - 50% *Peers* Limitados		
Tempo de *lookup* médio(seg)	1,194	0,396	0,291	0,285
Número de saltos médio	4,496	1,467	1,078	1,068

Tabela 6.4: Resultados do teste de *lookup* intensivo com 200 *peers* - Cenário 1, 2 e 3

do tempo de localização de recursos neste cenário, aumentou entre 84% e 89% no Chord, e 77% a 83% nas variantes do EpiChord. Nos restantes cenários é também evidente um aumento no tempo médio da localização de recursos, onde como também era esperado, se vê que o aumento do tempo de localização de recursos está directamente ligado ao aumento do número de *peers* com limitações. Estes resultados mostram também, que o envio de mensagens em paralelo do EpiChord, o torna mais imune aos problemas causados pelos *peers* limitados.

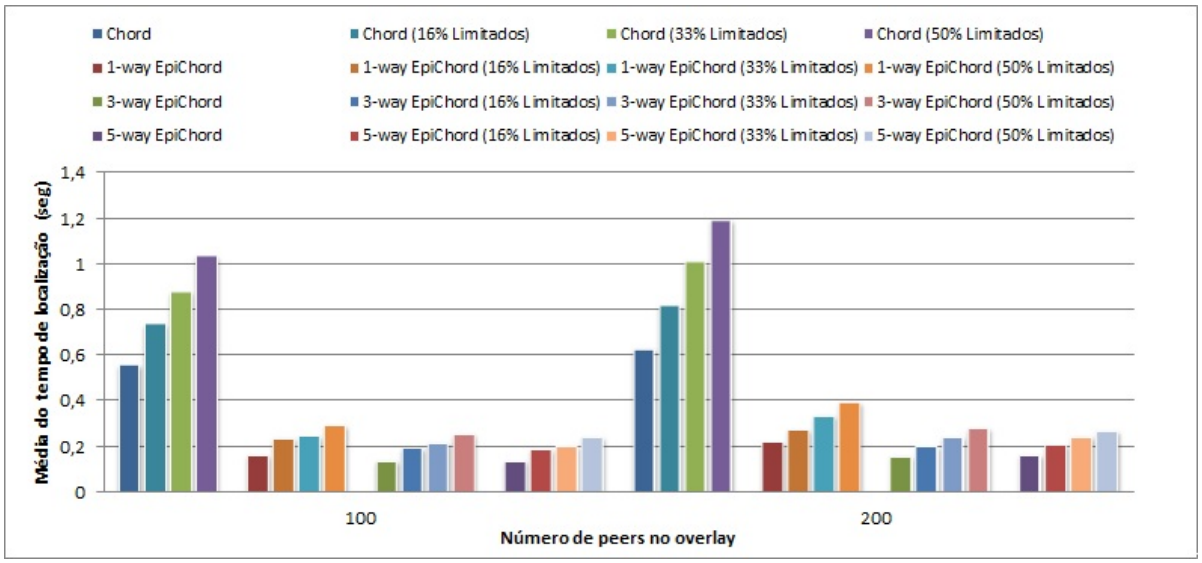

FIGURA 6.2: Comparação dos resultados dos tempos médios de localização de recursos com e sem *peers* limitados

6.4 Clientes com limitações

Para demonstrar o beneficio da existência de uma hierarquia num *overlay* P2P, foram repetidos os testes anteriores, tendo os *peers* com limitações sido trocados por clientes, passando a existir uma hierarquia de dois níveis. Os testes foram efectuados com 100 e 200 nós (entre *peers* e clientes), tendo os cenários de teste sido os seguintes:

- Cenário 1 - Foram colocados como clientes 16% dos nós, correspondente a 16 clientes e 84 *peers*, e 32 clientes e 168 *peers* de um total de 100 e 200 nós respectivamente.
- Cenário 2 - Foram colocados como clientes 33% dos nós, correspondente a 33 clientes e 67 *peers*, e 66 clientes e 134 *peers* de um total de 100 e 200 nós respectivamente.
- Cenário 3 - Foram colocados como clientes 50% dos nós, correspondente a 50 clientes e 50 *peers*, e 100 clientes e 100 *peers* de um total de 100 e 200 nós respectivamente.

Os resultados dos testes com a existência de uma hierarquia formada por *peers* e clientes encontram-se nas tabelas 6.5 e 6.6.

Analisando as tabelas e comparando-as com as tabelas do teste anterior, é visível que o desempenho do *overlay* na localização de recursos, beneficia da existência de uma

hierarquia. Os resultados mostram, que a troca dos *peers* com limitações, por clientes, que mantém essas limitações, permite que o desempenho do *overlay* melhore. Analisando os resultados do cenário 1 e 3, concluimos que o desempenho do *overlay* pode melhorar em cerca de 33 a 100% no Chord, de 26 a 93% na variante do EpiChord sem paralelismo, variando de 24 a 50% nas variantes do EpiChord que recorrem ao envio de mensagens em paralelo.

	Chord	**EpiChord (P=1 L=1)**	**EpiChord (P=3 L=3)**	**EpiChord (P=5 L=3)**
	Cenário 1 - 16 Clientes (com limitações) e 84*Peers*			
Tempo de *lookup* médio(seg)	0,545	0,181	0,156	0,156
Número de saltos médio	3,860	1,161	1,000	1,000
	Cenário 2 - 33 Clientes (com limitações) e 67 *Peers*			
Tempo de *lookup* médio(seg)	0,550	0,183	0,173	0,172
Número de saltos médio	3,760	1,074	0,994	0,993
	Cenário 3 - 50 Clientes (com limitações) e 50 *Peers*			
Tempo de *lookup* médio(seg)	0,516	0,194	0,186	0,183
Número de saltos médio	3,438	1,048	0,988	0,981

Tabela 6.5: Resultados do teste de *lookup* intensivo com 100 nós (peers e clientes) - Cenário 1, 2 e 3

	Chord	**EpiChord (P=1 L=1)**	**EpiChord (P=3 L=3)**	**EpiChord (P=5 L=3)**
	Cenário 1 - 32 Clientes (com limitações) e 168 *Peers*			
Tempo de *lookup* médio(seg)	0,616	0,220	0,164	0,160
Número de saltos médio	4,391	1,415	1,048	1,034
	Cenário 2 - 66 Clientes (com limitações) e 134 *Peers*			
Tempo de *lookup* médio(seg)	0,609	0,220	0,178	0,177
Número de saltos médio	4,244	1,266	1,029	1,016
	Cenário 3 - 100 Clientes (com limitações) e 100 *Peers*			
Tempo de *lookup* médio(seg)	0,594	0,205	0,190	0,189
Número de saltos médio	3,996	1,122	1,005	0,997

Tabela 6.6: Resultados do teste de *lookup* intensivo com 200 nós (peers e clientes) - Cenário 1, 2 e 3

Os gráficos das figuras 6.2 e 6.3 mostram o valor médio do tempo de localização de recursos, em todos os testes efectuados neste trabalho. Em 6.2 estão representados os valores obtidos nos testes com e sem *peers* limitados, com *overlays* compostos por 100 e 200 *peers*. O gráfico 6.3 contém os valores dos testes em que não havia nós com limitações, e também dos testes nos quais foram introduzidos clientes com limitações.

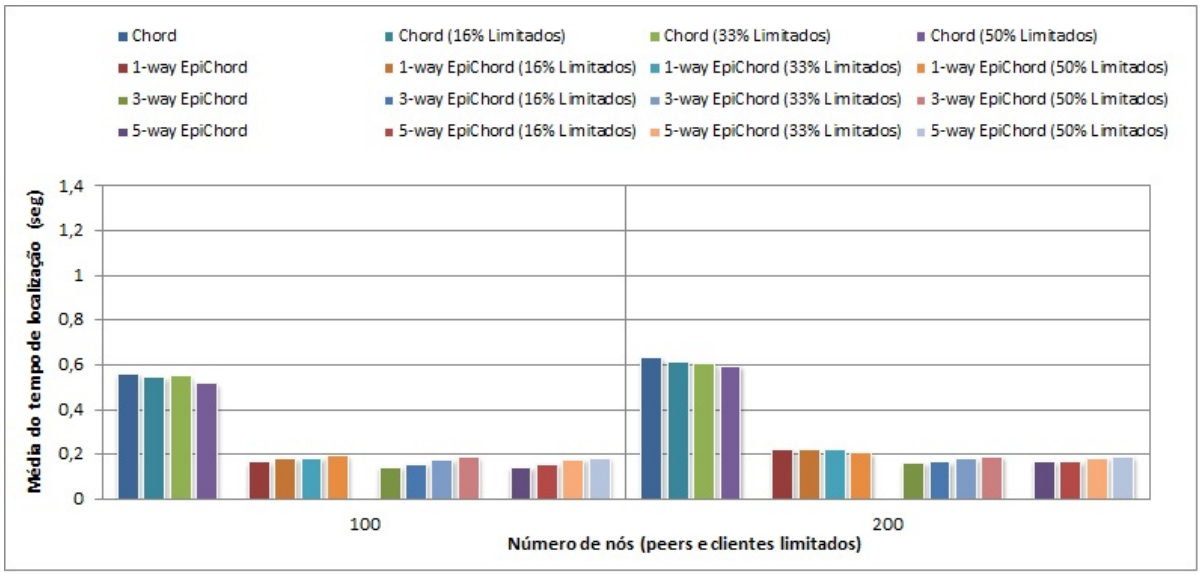

FIGURA 6.3: Comparação dos resultados dos tempos médios de localização de recursos com e sem clientes

O impacto que *peers* com limitações podem ter num *overlay*, discutido na analise dos valores dos testes, é facilmente perceptível através da comparação dos gráficos 6.2 e 6.3.

Uma vez que estes gráficos possuem informação de todos os testes efectuados, é necessário relembrar que os resultados obtidos nos testes de validação da implementação, foram feitos com *overlays* formados por 100 e 200 *peers*, enquanto que nestes testes o número máximo de *peers* foi de 84 e 168. Daí que, comparando os resultados obtidos neste teste, com os resultados obtidos na validação, pareça que em alguns casos o desempenho melhora com a existência de clientes. A verdade é que, o teste com clientes utiliza menos *peers*, pelo que é normal que o número médio de saltos e o tempo de localização possam ser inferiores.

Capítulo 7

Conclusão

Nesta dissertação foi feito um estudo das redes *peer-to-peer*, tendo sido estudados varios protocolos, para redes P2P estruturadas e não estruturadas. Dos protocolos estudados destacam-se: o gnutella e Chord. O gnutella foi um dos primeiros protocolos P2P a surgir, e é utilizado para criar redes P2P não estruturadas, onde a forma como os *peers* estabelecem ligações entre si no *overlay* não é definida, e a localização de recursos é feita através de técnicas que 'inundam' a rede com mensagens de localização.

O Chord, é um dos protocolos mais comuns nas redes P2P estruturadas baseadas em DHTs. Neste tipo de rede P2P, os *peers* formam um anel no qual a posição de cada um é obtida de forma determinística, através do identificador que lhes é atribuído. Para além disso, os recursos armazenados no *overlay*, são também eles posicionados de forma determinística, permitindo melhorar significativamente o desempenho da localização de recursos. No Chord, os *peers* possuem uma tabela de encaminhamento na qual possuem informação sobre os seus vizinhos directos(o *peer* que o sucede e antecede no anel) e uma *finger table*, que é constituída por um conjunto de entradas com informação sobre *peers* posicionados numa determinada posição do *overlay*.

Para além do estudo dos protocolos P2P mais comuns, foi também estudado o Epi-Chord, que é uma variante do protocolo Chord, e que, segundo os seus autores, permite melhorar significativamente o desempenho do *overlay* na localização de recursos.

Foi estudado o protocolo de sinalização SIP, que possuí grande relevância neste trabalho, visto que é o protocolo utilizado para a comunicação entre os elementos da rede *peer-to-peer*. O SIP é um protocolo de texto simples, bastante maduro, pois o seu funcionamento e desempenho é já bem conhecido, e para além disso, é um protocolo extensível. O principal objectivo do SIP é o estabelecimento de sessões, permitindo também a negociação de parâmetros quer no estabelecimento de uma sessão como numa sessão já estabelecida, enquadrando-se por isso, perfeitamente no tipo de protocolo necessário para a comunicação entre *peers* de um *overlay*.

Um dos objectivos desta dissertação era o estudo dos protocolos P2PSIP existentes de forma a verificar se já existia algum que fosse totalmente baseado em SIP. Por isso, foram estudadas algumas das propostas existentes, como o dSIP, o P2PP e o RELOAD. Das várias propostas P2PSIP estudadas, foi escolhido o protocolo dSIP como protocolo a implementar, pois cumpria os objectivos pretendidos desta dissertação, nomeadamente o facto de ser um protocolo totalmente baseado em SIP.
As restantes propostas para protocolos P2PSIP, como por exemplo o RELOAD (que é a mais recente proposta, dos mesmos autores de dSIP), propõem a criação de novos protocolos, binários, para efectuar a comunicação entre os elementos do *overlay*. Nestas propostas, o SIP é utilizado apenas numa camada superior, não sendo utilizado para a criação e gestão do *overlay*.

As propostas P2PSIP mais recentes tem vindo a propor a utilização de protocolos binários em vez do SIP, para gestão do *overlay*. Contudo, e apesar de o SIP ser um protocolo de texto, isso não deve ser visto como uma desvantagem, pois num mundo cada vez mais virado para a Web, onde os protocolos de texto são cada vez mais utilizados, desde a visualização de páginas *web*, até aos *web-services*. A escolha de um protocolo maduro, e experimentado como o SIP, permite que o protocolo P2PSIP utilizado possa também usufruir dos métodos utilizados pelo SIP para ultrapassar algumas barreiras criadas por firewalls e NAT's. Por tudo isto, faz todo o sentido que a comunicação entre os *peers* de um *overlay* seja feita recorrendo a uma solução totalmente baseada em SIP.

Para além da implementação de um protocolo P2PSIP para a criação de um *overlay* P2P, nesta dissertação, foi proposta uma solução para um *overlay* P2P com dois níveis

hierárquicos. O objectivo da criação de uma hierarquia de dois níveis é melhorar o desempenho do *overlay*. Assim, o primeiro nível da hierarquia é composto por *peers* que formam o *overlay*, enquanto que, o segundo nível é composto por clientes, que tem a capacidade de actuarem como *peers*, mas que, por algum motivo (por exemplo limitações de hardware, ou conectividade) não participam activamente no *overlay*. Um cliente, estabelece ligação com um ou vários *peers*, usufruindo dos serviços disponibilizados pelo *overlay* através destes. Desta forma, se *peers* que influenciam negativamente o desempenho de um *overlay*, passarem a actuar como clientes, o desempenho do *overlay* na localização de recursos pode melhorar significativamente.

Na implementação JAVA desenvolvida no âmbito desta dissertação, foram implementados os algoritmos DHT Chord e EpiChord, sendo que a comunicação entre os nós da rede é toda ela feita através do protocolo P2PSIP dSIP. A implementação desenvolvida, permite a criação de *overlays* P2P, com um ou dois níveis hierárquicos. Para suportar uma hierarquia de dois níveis, aa implementação do protocolo dSIP, teve de ser extendida, tendo sido especificados novos cabeçalhos para permitir a comunicação entre *peers* pertencentes ao *overlay*, e clientes.

Através da análise aos resultados dos testes efectuados, e comparando-os com os resultados obtidos pelos autores do EpiChord, concluímos que os algoritmos implementados se encontram a funcionar correctamente. Os resultados mostram também que o aumento do número de *peers* com limitações no *overlay*, influência o desempenho global do *overlay* na localização de recursos.
Os resultados dos testes, nos quais os *peers* com limitações foram substituídos por clientes com as mesmas limitações, provam que, a existência de uma hierarquia de dois níveis, na qual os *peers* com limitações passam a clientes, beneficia o desempenho do *overlay*. O desempenho neste caso, é idêntico ao desempenho do overlay quando este foi testado exclusivamente com peers, dos quais nenhum possuía qualquer tipo de limitação.
Comparando os resultados dos vários testes efectuados, com o Chord, e com o EpiChord sem paralelismo, isto é, parâmetros P e L com valor 1, é visível que a utilização da cache no EpiChord, permite melhorar o desempenho do overlay.

Como trabalho futuro, e uma vez tendo sido provado o bom funcionamento da implementação desenvolvida, pretendemos efectuar testes com um número significativamente

maior de *hosts*(*peers* e clientes). Dessa forma, poderemos analisar num cenário mais realista, as vantagens e desvantagens da utilização do SIP para a criação de um *overlay* P2P assim como avaliar melhor o desempenho dos algoritmos Chord e EpiChord num *overlay* com e sem dois níveis hierárquicos.

Seria também interessante implementar um mecanismo que dinamicamente promovesse clientes a *peers*, e que despromovesse *peers* a clientes. Os parâmetros a ter em consideração para a promoção e despromoção teriam obviamente que ser bem analisados, de forma a prevenir que este mecanismo tivesse um efeito negativo no desempenho do *overlay*.

Seria também interessante como trabalho futuro, implementar uma das propostas P2PSIP que utilizam um protocolo binário em vez do SIP. De forma a verificar se a utilização de um protocolo binário em vez do SIP (proposto em algumas das propostas P2PSIP estudadas) melhora ou não o desempenho de um *overlay*, assim como as possíveis contrapartidas da substituição do SIP.

Bibliografia

[1] G. Camarillo, "Peer-to-Peers (P2P) architecture: Definition, taxonomies, examples, and applicability," Nov. 2009.

[2] C. Jennings, S. Baset, H. Schulzrinne, B. Lowekamp, and E. Rescorla, "A SIP usage for RELOAD," *Internet Draft*, Jul. 2010. [Online]. Available: http://tools.ietf.org/html/draft-ietf-p2psip-sip-05

[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, 2001, pp. 149–160.

[4] B. Leong, B. Liskov, and E. D. Demaine, "Epichord: Parallelizing the chord lookup algorithm with reactive routing state management," *Computer Communications*, vol. 29, no. 9, pp. 1243 – 1259, 2006, iCON 2004 - 12th IEEE International Conference on Network 2004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140366405003750

[5] S. Saroiu, P. Gummadi, S. Gribble *et al.*, "A Measurement Study of Peer-to-Peer file Sharing Systems," in *proceedings of Multimedia Computing and Networking*, vol. 2002. Citeseer, 2002, p. 152.

[6] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys and Tutorials*, vol. 7, pp. 72–93, 2005.

[7] B. Pourebrahimi, K. Bertels, and S. Vassiliadis, "A survey of peer-to-peer networks," in *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, 2005.

[8] D. Clip2, "The annotated gnutella protocol specification v0.4," 2000. [Online]. Available: http://rfc-gnutella.sourceforge.net/developer/stable/index.html

[9] T. Klingberg and R. Manfredi, "Gnutella 0.6 - protocol development," Jun. 2002. [Online]. Available: http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html

[10] G. Camarillo, M. Handley, J. Peterson, J. Rosenberg, A. Johnston, H. Schulzrinne, and R. Sparks, "SIP: session initiation protocol," http://tools.ietf.org/html/rfc3261, Jun. 2002. [Online]. Available: http://tools.ietf.org/html/rfc3261

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC2616: Hypertext Transfer Protocol–HTTP/1.1," *RFC Editor United States*, 1999.

[12] X. Zheng and V. Oleshchuk, "A Secure Architecture for P2PSIP-based Communication Systems," in *Proceedings of the 2nd international conference on Security of information and networks*. ACM, 2009, pp. 75–82.

[13] D. Bryan, "dSIP: a P2P approach to SIP registration and resource location draft standard," *Internet Draft*, Feb. 2007. [Online]. Available: http://tools.ietf.org/html/draft-bryan-p2psip-dsip-00

[14] D. Bryan, B. Lowekamp, and C. Jennings, "Sosimple: A Serverless, Standards-based, P2P SIP Communication System," 2005.

[15] D. Wing, P. Matthews, R. Mahy, and J. Rosenberg, "Session traversal utilities for NAT (STUN)," Oct. 2008. [Online]. Available: http://tools.ietf.org/html/rfc5389

[16] J. Rosenberg, R. Mahy, and P. Matthews, "Traversal using relays around NAT (TURN): relay extensions to session traversal utilities for NAT (STUN)," Apr. 2010. [Online]. Available: http://tools.ietf.org/html/rfc5766

[17] J. Rosenberg, R. Mahy, and P. Matthews, "Interactive connectivity establishment (ICE): a protocol for network address translator NAT traversal for offer/answer

protocols," Apr. 2010. [Online]. Available: http://merlot.tools.ietf.org/pdf/rfc5245.pdf

[18] H. Schulzrinne, M. Matuszewski, and S. Baset, "Peer-to-Peer protocol (P2PP)," *Internet Draft*, Nov. 2007. [Online]. Available: http://tools.ietf.org/html/draft-baset-p2psip-p2pp-01

[19] C. Jennings, S. Baset, H. Schulzrinne, B. Lowekamp, and E. Rescorla, "REsource LOcation and discovery (RELOAD) base protocol," *Internet Draft*, Oct. 2010. [Online]. Available: http://tools.ietf.org/html/draft-ietf-p2psip-base-13

[20] C. Jennings, J. Rosenberg, and E. Rescorla, "Address settlement by peer to peer," *Internet Draft*, Jul. 2007. [Online]. Available: http://tools.ietf.org/html/draft-jennings-p2psip-asp-00

[21] D. Bryan and M. Zangrilli, "A chord-based DHT for resource lookup in P2PSIP," Feb. 2007. [Online]. Available: http://tools.ietf.org/html/draft-zangrilli-p2psip-dsip-dhtchord-00

[22] M. Ranganathan, P. O'Doherty, and J. van Bemmel, "JAIN-SIP: Stack sip for java." [Online]. Available: http://jsip.java.net/

[23] S. Cirani and L. Veltri, "A kademlia-based DHT for resource lookup in P2PSIP," http://tools.ietf.org/html/draft-cirani-p2psip-dsip-dhtkademlia-00, Oct. 2007. [Online]. Available: http://tools.ietf.org/html/draft-cirani-p2psip-dsip-dhtkademlia-00

[24] J. Ahrenholz, C. Danilov, T. Henderson, and J. Kim, "CORE: A real-time network emulator," in *IEEE Military Communications Conference*, 2008.

[25] Universidade do Minho - Dep. de Informática, "SeARCH - services and advanced research computing." [Online]. Available: http://search.di.uminho.pt