Universidade do Porto

Faculdade de Engenharia

FEUP

Distributed Backup Service, Internet version (peer-to-peer)

Relatório

Sistemas Distribuídos
3º ano do Mestrado Integrado em Engenharia Informática e Computação

Turma 5 Grupo 11:
Anabela Costa e Silva - up201506034 - up201506034@fe.up.pt
Beatriz Souto de Sá Baldaia - up201505633 - up201505633@fe.up.pt
João Francisco Veríssimo Dias Esteves - up201505145 - up201505145@fe.up.pt
Renato Alexandre Sousa Campos - up201504942 - up201504942@fe.up.pt

27 de maio de 2018

Conteúdo

1 Introdução **1**

2 Arquitetura **1**
2.1 Package program 1
2.1.1 Peer 1
2.1.2 Leases 2
2.2 Package communication 2
2.2.1 Subpackage communication.messages 2
2.2.2 Server 2
2.2.3 Client 3
2.2.4 ParseMessageAndSendResponse 4
2.3 Package chord 4
2.3.1 ChordManager 4
2.3.2 FixFingerTable 5
2.3.3 Stabilize 5
2.3.4 CheckPredecessor 5
2.4 Package database 5
2.4.1 BackupRequest 5
2.4.2 ChunkInfo 5
2.4.3 Database 5
2.4.4 DBUtils 6
2.4.5 FileStoredInfo 6
2.5 Package runnableProtocols 6
2.6 Package utils 6
2.6.1 SingletonThreadPoolExecutor 6
2.6.2 MyRejectedExecutionHandler 6
2.6.3 Confidentiality 7
2.6.4 ReadInput 7
2.6.5 Utils 7

3 Implementação **7**
3.1 SSLSocket e SSLServerSocket 7
3.2 Join 7
3.3 Leases 8
3.4 Protocolo Backup 8
3.5 Protocolo Restore 9
3.6 Protocolo Delete 10
3.7 Base de dados 11
3.8 Concorrência 11

4 Assuntos Relevantes **11**
4.1 Segurança 11
4.2 Escalabilidade 11
4.3 Consistência 12
4.4 Tolerância a falhas 13

5 Conclusão **13**

1 Introdução

Este segundo projeto consiste no melhoramento do primeiro trabalho realizado para a unidade curricular de Sistemas Distribuídos: um serviço de backup.

Relembrando o primeiro projeto, este envolvia um serviço de backup de ficheiros onde servidores (*peers*) se ligavam a uma rede local, comunicando entre si através de canais *multicast*. O cliente ligava-se a um *peer* da rede dispondo dos subprotocolos Backup, Restore, Delete e Reclaim Space.

Já para o segundo trabalho alterámos a arquitetura optando por uma estratégia Peer-To-Peer (P2P), funcionando através da Internet.

Uma rede P2P segue uma arquitetura distribuída e descentralizada onde todos os membros (*peers*) têm os mesmos privilégios. Cada peer dedica parte dos seus recursos (memória) à rede, desempenhando tanto o papel de servidor como o de cliente.

Efetivamente, o sistema P2P implementa uma rede virtual sobre uma rede física já existente. Esta pode ser estruturada ou não-estruturada.

Uma rede não-estruturada estabelece conexões aleatórias entre os membros. Assim, a procura de um elemento escasso por parte de um peer sobrecarrega a rede com pedidos, aumentando o tráfego e processamento.

Por outro lado, uma rede estruturada proporciona uma procura de recursos eficiente, independentemente do grau de replicação do recurso. Deste modo, para fazermos uso deste tipo de rede, implementamos uma *distributed hash table* (DHT) que guarda pares (chave-valor). Para que a rede seja escalável com a introdução e remoção de *peers* nela, esta tabela é periodicamente atualizada por todos os *peers* pertencentes à rede.

Este relatório descreverá a arquitetura e implementação do nosso trabalho, salientando também pontos relevantes relativos à implementação, terminando com a conclusão e bibliografia.

2 Arquitetura

A nossa aplicação está dividida em seis packages: *chord, communication, database, program, runnableProtocols e utils.*

2.1 Package program

Este package contém as classes *Peer e Leases* e centra-se na inicialização do nosso programa:

2.1.1 Peer

Peer contém a função main. Ao receber os parâmetros do utilizador:

1. Cria o objeto *chordManager* que, como será à frente descrito, executa o protocolo chord para a *peer-to-peer distributed hash table* (DHT).

2. Cria o servidor (objeto da classe *Server*) e corre-o.

3. Consoante o número de argumentos, junta-se à rede.

4. Inicializa o contrato *lease*.

5. Apresenta na consola o menu de utilização do nosso programa, onde o utilizador poderá usufruir das funcionalidades e serviços de *backup, restore e delete*.

2.1.2 Leases

Leases é uma classe que implementa a interface *Runnable* e que, após o tempo estabelecido, consulta a base de dados para saber quais os ficheiros que deve apagar e quais os que deve atualizar, isto é, voltar a pedir o backup.

A implementação deste processo será descrita mais à frente na secção 3.3.

2.2 Package communication

Este package contém as classes *Client, Server e ParseMessageAndSendResponse* e centra-se na receção, envio e tratamento (leitura) de mensagens. Na verdade, um nó da rede é tanto um cliente como um servidor pois, ao mesmo tempo, é capaz de pedir e fornecer serviços.

2.2.1 Subpackage communication.messages

Este subpackage contém a classe *MessageFactory* que é responsável por criar todas as mensagens com formato consistente. Contém ainda a enumeração MessageType que declara todos os tipos de mensagens suportados.

2.2.2 Server

Esta classe implementa a interface *Runnable* e, no seu método run(), cria um *SSLServerSocket* e, num ciclo infinito, para cada conexão aceite cria um novo *socket* válido apenas para esse mesmo pedido. Quando o servidor, depois de estabelecer uma conexão com o cliente, lê o pedido, processa-o. Para tal, por cada pedido ele instancia uma thread da classe *ParseMessageAndSendResponse* que filtra a mensagem e invoca a função correspondente que a trata. Assim o servidor fica livre para receber mais pedidos.

2.2.3 Client

O cliente cria um *socket* para enviar a um *peer* um pedido, sendo que a mensagem é encriptada. Dependendo do tipo de serviço, espera ou não por uma resposta do outro *peer*. Este *socket* criado é fechado logo após o envio do pedido (caso não tenha de esperar por uma resposta), após a receção da resposta ou após o *timeout* estabelecido para o canal.

As mensagens que o cliente é capaz de enviar são:

- **PING** - Enviada periodicamente pelo protocolo chord para verificar se o predecessor falhou.
- **LOOKUP** - Utilizada para obter o Peer que é responsável por um ficheiro.
- **NOTIFY** - Enviada de *n* para *n'*, para notificar *n'* que *n* pode ser o seu predecessor.
- **SUCCESSORS** - Enviada periodicamente pelo protocolo chord para informar o predecessor sobre quais os peers que o sucedem. Cada peer mantém uma lista dos 5 peers que o sucedem. Assim o protocolo chord fica tolerante à falha do sucessor.
- **STABILIZE** - Enviada periodicamente pelo protocolo chord para obter o predecessor do atual sucessor.
- **RESPONSIBLE** - Enviada para notificar o Peer que acabou de se juntar à rede sobre os ficheiros pelos quais ele é responsável.
- **PUTCHUNK** - Utilizada pelo protocolo *backup* para introduzir um chunk na rede, com um determinado grau de replicação.
- **KEEPCHUNK** - Quando o grau de replicação ainda não atingiu o valor desejado, esta mensagem permite que o pedido de backup continue.
- **STORED** - Após ser executado o backup, eventualmente todos os *peers* pelos quais passou uma mensagem KEEPCHUNK, enviarão para o seu predecessor a mensagem STORED com o grau de replicação do chunk que é sabido até então.
- **CONFIRMSTORED** - Após receber o STORED, o dono da chave do ficheiro a fazer backup envia ao nó que pediu o backup uma mensagem CONFIRMSTORED para que este saiba que o backup foi executado e o seu grau de replicação atual.
- **GETCHUNK** - Utilizada pelo protocolo *restore* para readquirir um *chunk* de um ficheiro.

- **INITDELETE** - Utilizada pelo protocolo *delete* para apagar um ficheiro da rede. Esta mensagem é enviada pelo cliente ao dono da chave do ficheiro a apagar.
- **DELETE** - Enquanto houver nós da rede a guardar chunks do ficheiro que foi pedido para ser apagado, esta mensagem é enviada para o sucessor do *peer* que acabou de a receber.

2.2.4 ParseMessageAndSendResponse

ParseMessageAndSendResponse é a classe que procede ao parsing e tratamento das mensagens recebidas.

Dependendo do tipo de mensagem recebida o seu respectivo tratamento é depois encaminhado para as classes responsáveis, já com os elementos importantes tratados. Ela pode ainda esperar por uma resposta e enviá-la de volta ao cliente.

2.3 Package chord

Este package contém as classes *AbstractPeerInfo*, *NullPeerInfo*, *PeerInfo*, *CheckPredecessor*, *ChordManager*, *FixFingerTable*, e *Stabilize*. Foca-se no funcionamento do algoritmo chord que mantém todos os *peers* a poderem ter acesso a todos os ficheiros, sem terem que saber onde é que eles estão guardados, assim torna eficiente a localização do peer que guarda o ficheiro desejado.

Este protocolo funciona mapeando uma chave com um ficheiro e nomeando um peer responsável por essa chave. Assim o responsável pelo ficheiro com uma determinada chave é o seu peer sucessor, ou seja, o peer com o menor id maior ou igual ao valor da chave.

Como os peer podem ligar-se e desligar-se da rede múltiplas vezes, por exemplo em caso de falha de um peer, este protocolo possui forma de estabilizar a rede nessas situações. Este é o principal papel deste package, assim como dada uma chave descobrir o seu peer responsável.

2.3.1 ChordManager

ChordManager é a classe principal deste package. Ela implementa a interface *Runnable*, e é a responsável por periodicamente correr os seguintes protocolos *Stabilize, CheckPredecessor e FixFingerTable*. Desta forma, garante que o lookup funciona correctamente, atualizando os seus sucessores.

Está também encarregue do funcionamento do lookup, usado para encontrar o responsável por um ficheiro.

2.3.2 FixFingerTable

FixFingerTable é a classe que exclusivamente atualiza a FingerTable. Ela implementa a interface *Runnable* e é chamada periodicamente para esse efeito.

Ela faz lookup do responsável pela chave $n + 2^i$ onde n é o id do *peer* e i o índice na FingerTable onde o membro da rede a procurar será guardado.

2.3.3 Stabilize

Stabilize é a classe que é usada para atualizar o nosso sucessor.

Ela pergunta ao nosso sucessor quem é o predecessor dele, e se for diferente de si próprio e estiver dentro do intervalo $]n, previousSucessor[$ alteramos o nosso sucessor apropriadamente. Independentemente do resultado notificamos o nosso successor atual que pensamos ser o seu predecessor.

Caso não consigamos perguntar ao nosso sucessor quem é o seu predecessor, assumimos que ele se desligou da rede e usamos o peer que tínhamos guardado como nosso segundo sucessor como nosso primeiro sucessor.

2.3.4 CheckPredecessor

CheckPredecessor é a classe que verifica se o predecessor falhou. Caso não consiga contactar com este, atribui-lhe o valor *null*.

2.4 Package database

Esta package contém as classes *BackupRequest*, *ChunkInfo*, *Database*, *DBUtils*, e *FileStoredInfo* e trata dos acessos a base de dados onde guardamos a informação sobre o *peer*, sobre os serviços de backup que já pediu e sobre os *chunk* que guarda.

2.4.1 BackupRequest

Esta classe representa um pedido de backup, guardando a informação do ficheiro a fazer backup, o seu grau de replicação desejado e a chave de encriptação.

2.4.2 ChunkInfo

Esta classe representa um chunk que temos guardado no nosso peer. Contém o tamanho deste chunk, assim como o seu id, o id do ficheiro que o contém e o grau de replicação que este peer pensa que este tem.

2.4.3 Database

A classe *Database* cria e liga-se à base de dados e é responsável pela conexão com esta.

Quando o peer se está a juntar à rede pela primeira vez, ele cria a base de dados assim como todas as tabelas de que ela necessita.

2.4.4 DBUtils

DBUtils é a classe que usamos para comunicar com a base de dados.

Nela utilizamos *prepared statements* para comunicarmos com a base de dados, e inserirmos, alterarmos, apagarmos, e inquirirmos a base de dados.

2.4.5 FileStoredInfo

Esta classe representa um ficheiro do qual temos pelo menos um chunk guardado, ou somos o responsável. Contém o id do ficheiro, assim como o grau de replicação desejado. Contém também um booleano que indica se somos o peer responsável, e o id do peer que iniciou o backup deste ficheiro.

2.5 Package runnableProtocols

Esta package contém as classes *SendGetChunk*, *SendInitDelete*, e *SendPutChunk*. Todas elas implementam a interface *Runnable* e são chamadas numa thread separada por instância, para o programa ser mais concorrente. As suas implementações serão descritas mais à frente nas secções 3.4, 3.5, e 3.6, respectivamente.

2.6 Package utils

Esta package contém as classes *Confidentiality*, *MyRejectedExecutionHandler*, *ReadInput*, *SingletonThreadPoolExecutor* e *Utils*, e fornece funcionalidades auxiliares ao nosso programa.

2.6.1 SingletonThreadPoolExecutor

Esta classe representa a nossa *thread pool* que executa os protocolos responsáveis pelos serviços que o servidor é capaz de fornecer.

2.6.2 MyRejectedExecutionHandler

A *SingletonThreadPoolExecutor* para onde são enviados os protocolos, para serem executados, tem uma capacidade máxima (capacidade *Runtime.getRuntime().availableProcessors()* que retorna o número de processadores disponíveis para correr o código). Assim, se esta *thread pool* ficar cheia, o seu *handler "MyRejectedExecutionHandler"* reagenda a tentativa de executar os protocolos que chegaram, mas que não foram aceites.

2.6.3 Confidentiality

Esta classe é a responsável por encriptar e desencriptar os chunks aos quais fazemos backup. A sua implementação é descrita mais detalhadamente na secção 4.1.

2.6.4 ReadInput

Esta classe é a responsável pela interface da linha de comandos. Ela pergunta ao utilizador o que quer fazer e recolhe os dados para essa operação, remetendo depois para a classe Peer a sua realização.

2.6.5 Utils

Esta classe é responsável pelos *logs* gerados no decorrer do programa, assim como a escrita e leitura de ficheiro, a transformação da hash para o id, quer do peer, quer do ficheiro e a aritmética realizada em módulo com estes.

3 Implementação

3.1 SSLSocket e SSLServerSocket

SSL, ou na verdade TLS que é o sucessor de SSL, garante a privacidade dos dados na comunicação entre peers através de sockets encriptando a ligação.

Cada peer tem um *Server* e um *Client*. O *Client* é usado para enviar pedidos a um determinado peer, criando para o efeito uma *SSLSocket* ligada a este, enquanto que o *Server* tem uma *SSLServerSocket* que recebe cada pedido de ligação de outros peers e cria uma *SSLSocket* para cada um.

3.2 Join

Para um *peer* n juntar-se à rede, terá de conhecer previamente o ip e porta de outro *peer* n'. Tendo esse conhecimento, n envia uma mensagem LOOKUP(n) para n'. Se n' conhecer o sucessor de n, responder-lhe-á com uma mensagem SUCCESSOR(s), sendo s o sucessor de n. Caso contrário, com uma mensagem ASK(p), sendo p um *peer* que estará mais perto do sucessor de n. Enquanto receber uma mensagem ASK(p), n continuará a enviar a mensagem LOOKUP(n) para p até receber uma mensagem SUCCESSOR(s) (classe chord . ChordManager, linha 90). Após n obter o seu sucessor s, n irá eventualmente notificar s e s irá atualizar o seu predecessor. Neste processo de atualização, s envia a n informação(id do ficheiro e replicação desejada) sobre cada um dos ficheiros que n é responsável.

Através do protocolo *stabilize* (classe chord . ChordManager, linha 177) e *notify* (classe chord . ChordManager, linha 201) que executam periodicamente

em todos os *peers*, *n* irá eventualmente conhecer o seu sucessor e terá sido integrado com sucesso no *chord ring*.

3.3 Leases

Guardar dados em cache introduz sobrecarga e complexidade em assegurar a consistência, reduzindo alguns benefícios do seu desempenho. Num sistema distribuído, *caching* deve ser capaz de lidar com as complicações adicionais da comunicação e falhas do *host*.

Leases são propostas como mecanismos baseados em tempo que proporcionam um acesso eficiente e consistente aos dados armazenados em cache no sistema distribuído.

No nosso projeto após se atingir o período da *lease*, o *peer* consulta a base de dados para saber para que ficheiros pediu backup. Para estes ele volta a executar o protocolo de backup com o fim de manter o ficheiro na rede. É importante relembrar que quando um nó recebe uma mensagem KEEPCHUNK ou PUTCHUNK para um *chunk* que já está a guardar, é atualizada, para o momento atual, a última vez que o *chunk* foi guardado. O *peer* também acede a base de dados para concluir que ficheiros deve apagar. De facto, devem ser apagados os que a data da última vez que foram guardados excede o "tempo de vida"que lhes foi fixado.

Assim, mesmo que o dono de um ficheiro, que foi usado num serviço backup, for abaixo e os outros *peers* que guardam *chunks* deste ficheiro, por deixarem de receber mensagens KEEPCHUNK (decorrentes do protocolo de backup) apagarem esses *chunks*, quando o detentor do ficheiro voltar a ligar-se à rede, vai verificar a base de dados e executar novamente o backup.

3.4 Protocolo Backup

O protocolo *backup* funciona da seguinte maneira:

1. O utilizador escolhe o ficheiro e o seu grau de replicação desejado (classe utils.ReadInput);

2. É gerado um ID de ficheiro com base no nome do ficheiro e a última data de modificação (program.Peer, linha 184);

3. Várias informações do ficheiro, incluindo o ID de ficheiro e uma chave de encriptação, são guardadas na base de dados (program.Peer, linha 202);

4. O ficheiro vai sendo separado em *chunks* de tamanho até 64000 bytes, sendo cada um encriptado com a chave de encriptação referida anteriormente (program.Peer, linhas 204-210);

5. Por cada *chunk*, é enviada uma mensagem PUTCHUNK ao sucessor responsável pelo ficheiro, contendo, entre outros, o *chunk* encriptado e o grau de replicação desejado (runnableProtocols.SendPutChunk, linha 28);

6. Quando um peer recebe este PUTCHUNK (communication . ParseMessageAndSendResponse, linha 345), guarda o *chunk* encriptado, exceto se for o mesmo que pediu o backup (não guardando o chunk), e caso tenha espaço livre suficiente e, se o grau de replicação desejado for maior que 1, manda um KEEPCHUNK para o próximo sucessor com as mesmas informações do PUTCHUNK recebido exceto que tem um grau de replicação decrementado;

7. Quando um peer recebe um KEEPCHUNK (communication . ParseMessageAndSendResponse, linha 410), realiza as mesmas ações que na receção de um PUTCHUNK exceto se o peer for o dono da chave do ficheiro (significa que já se deu uma volta completa na rede). Neste último caso e/ou quando o grau de replicação recebido na mensagem for 1, é enviado um STORED, com grau de replicação 1, ao predecessor para confirmar o fim do backup e divulgar o grau de replicação do ficheiro sabido até então. Também é guardado na base de dados a informação do *chunk*, incluindo o grau de replicação enviado na mensagem STORED;

8. Aquando da receção de um STORED (communication . ParseMessageAndSendResponse, linha 301), o peer envia um STORED ao seu predecessor com grau de replicação incrementado;

9. Chegando o STORED ao peer responsável pelo ficheiro, a mensagem terá o grau de replicação real sendo isto gravado na sua base de dados e é enviado um CONFIRMSTORED ao peer que iniciou o *backup*;

10. Obtendo um CONFIRMSTORED (communication . ParseMessageAndSendResponse, linha 145), é imprimido para o log que o *chunk* foi guardado com o respetivo grau de replicação real.

3.5 Protocolo Restore

O protocolo *restore* funciona da seguinte maneira:

1. O utilizador escolhe o ficheiro a restaurar, sendo este um para o qual já pediu backup;

2. O peer pergunta a base de dados por quantos chunks este ficheiro é constituído, e remete para a classe *SendGetChunk* o envio das mensagens GETCHUNK;

3. Esta classe pergunta ao chord que peer é que se encontra responsável por este ficheiro e começa a enviar as mensagens GETCHUNK;

4. A aquando da receção desta mensagem, o peer responsável verifica se ele guardou o chunk. Se sim, envia-o de volta para o peer original (mensagem CHUNK), se não, remete a questão (mensagem GETCHUNK) para seu peer sucessor (classe communication . ParseMessageAndSendResponse, linha 240);

5. Quando o peer que iniciou o protocolo restore recebe um chunk, ele desencripta-o;

6. Se for o primeiro chunk deste ficheiro a ser recebido, cria-o (classe communication . ParseMessageAndSendResponse, linha 188;

7. Finalmente o peer usa um *AsynchronousFileChannel* para escrever no ficheiro assincronamente.

3.6 Protocolo Delete

Quando o utilizador pede para apagar do sistema um ficheiro ao qual já pediu backup, é apagado da base de dados, na tabela *backupsrequested*, a informação sobre o pedido de backup feito a este ficheiro e é executado o *SendInitDelete*. Esta classe *runnable* procede da seguinte forma:

1. A partir do algoritmo chord, procura o dono da chave do ficheiro. Um *peer* é o dono/sucessor da chave se o valor dessa estiver entre o valor do seu identificador e do identificador do seu predecessor. Poderá ser necessário recorrer ao lookup caso o dono da chave não seja o próprio *peer* nem o seu primeiro sucessor (classe chord . ChordManager, linha 211);

2. A mensagem INITDELETE é enviada para o sucessor da chave;

3. O sucessor da chave a receber a mensagem apaga todos os *chunks* do ficheiro que está a guardar, atualiza a memória livre para armazenamento de dados e apaga na base de dados, na tabela *filesstored*, a informação sobre o ficheiro (classe communication . ParseMessageAndSendResponse, linha 266);

4. Se, consoante o grau de replicação que o *peer* tem guardado, houver mais nós na rede com *chunks* do ficheiro ou se o nó atual não está a guardar *chunks*, é enviado para o seu sucessor a mensagem DELETE, verificando-se um comportamente semelhante ao descrito no ponto anterior caso o sucessor esteja a guardar *chunks* e assim sucessivamente.

3.7 Base de dados

Para garantir a persistência dos dados mesmo em caso de falha, optamos por guardar a informação em base de dados. Para este efeito, usamos a base de dados JavaDB pois já vem incluída com o JDK. Esta base de dados garante a integridade dos dados por ser uma base de dados relacional, obedecendo às propriedades ACID.

Assim, cada *peer* corre localmente a sua instância da base de dados onde guarda informação como: *backups* requisitados, ficheiros e *chunks* guardados e *peers* existentes na rede.

3.8 Concorrência

O programa recorre a várias *threads* ao longo da sua execução, sendo elas geridas por uma única *thread pool* com a *SingletonThreadPoolExecutor*.

Cada *Peer* corre paralelamente o *ReadInput* para interação com o utilizador, um *Server* para responder a pedidos de outros *peers* e um *ChordManager* para gerir as suas ligações com os outros *peers*.

A *ReadInput* consiste na leitura contínua do utilizador e na execução de uma thread por ação.

O *Server* executa, por cada mensagem que recebe, uma *thread* para a processar.

O *ChordManager* por sua vez subdivide-se em 3 *threads* que executam periodicamente:

- **Stabilize** - que atualiza o nosso sucessor;
- **CheckPredecessor** - que verifica se o nosso predecessor foi a baixo;
- **FixFingerTable** - que atualiza a FingerTable para lidar com a mudança de peers.

4 Assuntos Relevantes

4.1 Segurança

Usamos SSL para as ligações entre peers, de forma a assegurar a autenticação destes e a privacidade e a integridade das comunicações.

A confidencialidade dos ficheiros é garantida pela encriptação destes nos peers, ou seja, só o peer que faz o backup de um ficheiro na rede é que o pode restaurar já que só este guarda a chave de encriptação usada.

4.2 Escalabilidade

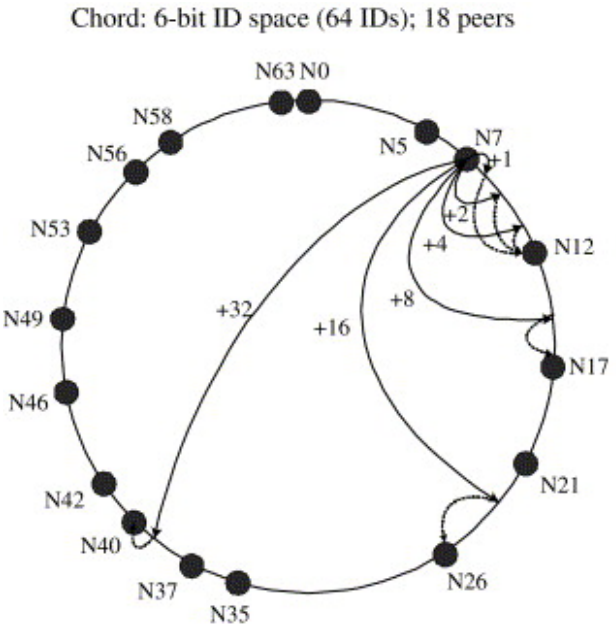
Escalabilidade é a capacidade do sistema tratar e suportar o crescimento da quantidade de informação.

Como foi referido na secção 2.3 o nosso programa implementa o protocolo chord para tornar eficiente a procura de ficheiros nos *peers*. Ele garante a escalabilidade pois cada nó tem uma *"finger table"* que guarda m apontadores para nós da rede que são responsáveis pelas chaves:

$$k_i = (n + 2^i) \pmod{m}, \quad \text{i} = 0, 1, ..., \text{m-1}$$

n representa o id do dono da *finger table*.

O custo do lookup cresce com o logaritmo do número de nós[1], visto que a distância entre um *peer* e o seu "alvo"a guardar na *finger table* é dividida a metade a cada iteração, ou seja, o custo do aumento de informação é razoavelmente pequeno.



Finger Table of N7

Entry	The closest ID	Contact
1	N7+1	N12
2	N7+2	N12
3	N7+4	N12
4	N7+8	N17
5	N7+16	N26
6	N7+32	N40

Figura 1: Vista do chord e *finger table* do nó 7

4.3 Consistência

Consistência significa que o resultado de leituras, escritas e atualizações da memória será previsível.

A consistência é mantida com o recurso às *leases*. Elas garantem que, caso um *peer* se desligue, ao voltar a conectar-se eventualmente apagará os *chunks* que guarda, mas cujo ficheiro já foi apagado da rede, e caso tenha sido o dono da chave associada a um ficheiro a ir a baixo, este restabelecerá o backup do ficheiro que, enquanto e devido à sua inatividade, foi apagado da rede.

A comunicação TCP também proporciona consistência visto que garante uma entrega de mensagens segura, ordenada e com verificação de erros.

4.4 Tolerância a falhas

A informação de cada peer é mantida numa base de dados, pelo que esta é mantida mesmo que um peer seja desligado.

A integridade da rede é assegurada pelo *Chord*, no sentido que caso um peer seja desconectado este é compensado atribuindo novos sucessores aos predecessores deste através do protocolo de estabilização. Este protocolo também assegura a replicação dos chunks guardados graças também ao sistema de *leases*, que basicamente repete periodicamente o protocolo de *backup* para cada chunk resultando em, caso um peer que contenha chunks seja desconectado, um outro *peer* guardar os chunks por este.

5 Conclusão

Este projecto foi realizado com sucesso. Implementa um serviço de backup peer-to-peer disponível pela internet.

Utilizando do algoritmo chord garantimos uma escalabilidade sustentável do sistema e o nosso uso de *leases* garante que o sistema tende para a consistência.

Garantimos também alguma tolerância a falhas mantendo a informação crítica do sistema em memória não volátil e usando o algoritmo chord para garantir que a rede continua a funcionar normalmente.

Em relação a segurança usamos a comunicação SSL e a encriptação dos chunks, para que nem os peers que não fizeram backup os possam desencriptar.

Como melhoramento deste projecto poderíamos alterar o nosso trabalho para permitir que peer se encontrem em diferentes redes privadas, isto é, fazer o mapeamento NAT dos endereços.

Referências

[1] I I. Stoica et al., "Chord: A scalable peer-to-peer lookup protocol for Internet applications", IEEE/ACM Transactions on Networks, (11)1:17-32, Feb 2003

[2] Cary G. Gray and David R. Cheriton, "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency ", Computer Science Department, Stanford University