Mestrado Integrado em Engenharia Informática

Sistemas Distribuídos em Larga Escala
Trabalho Prático

Relatório
Decentralized Timeline

Grupo B
Bruno Santos pg44414
Flávio Martins a65277
Francisco Morais pg10293
Pedro Costa a85700

8 de junho de 2021

Conteúdo

1 Introdução

Este projeto visa a criação de um serviço de timeline descentralizado obtido de dispositivos peer-to-peer ou edge. Os utilizadores possuem uma identidade e publicam pequenas mensagens de texto na sua máquina local, formando uma linha do tempo local. Podem subscrever outros utilizadores, que irão armazenar e encaminhar o conteúdo das suas timelines. O conteúdo remoto está disponível quando uma fonte ou subscritor da fonte está online e pode encaminhar as informações. As informações de fontes subcritas podem ser efêmeras e apenas armazenadas e encaminhadas por um determinado período de tempo.

O código-fonte desenvolvido durante a elaboração deste projeto encontra-se no repositório GitHub acessível em *https://github.com/BrunoXBSantos/SDLE-TP.git*.

2 Descrição do problema

Uma "timeline"(linha do tempo) é a exibição de uma lista de eventos numa ordem cronológica. Em oposição a uma localização centralizada, uma "timeline"descentralizada é construída em torno de um sistema descentralizado de utilizadores que armazenam parte dos dados da timeline, criando um sistema resiliente de armazenamento e partilha de dados. Qualquer utilizador na rede pode servir esse bloco de dados para o seu endereço, e outros pares na rede podem encontrar e solicitar esse bloco de dados de qualquer nó na rede. Desta forma, surgem alguns desafios na conceção e implementação de um sistema descentralizado.

Um problema fundamental que confronta as aplicações "peer-to-peer"é localizar com eficiência o nó que armazena uma determinada informação. Alguns algoritmos P2P procuram resolver este problema, com diferentes abordagens. Vamos considerar a abordagem de superpeers e DHT, e selecionar um algoritmo para responder a estes desafio mais em detalhe na secção seguinte.

Outro problema fundamental, é o de ordenar os blocos de informação por ordem cronológica, uma vez que são distribuídos, e não existe uma referência temporal centralizada. Para este desafio vamos considerar a utlização de "causal histories"para codificar a causalidade, ou seja, a memória dos eventos que são as mensagens da timeline. E considerar a utilização de "vector clocks"para esta ordenação, como mecanismo de codificar de forma eficiente as "causal histories".

Outro problema é o de como juntar um novo nó à rede, e saber a qual nó existente da rede se ligar. Para este desafio vamos considerar que deve haver um nó que todos os outros nós conhecem, chamado de nó "bootstrap", para auxiliar o algoritmo de procura no processo de adesão de novos nós a uma rede P2P.

3 Discussão das abordagens

3.1 Super-peers v DHT

Dada a escalabilidade e eficiência alcançadas pela rede descentralizada gnutella, com a utilização de "super-peers", ponderamos o uso deste tipo de arquitetura, em comparação com tabelas hash distribuídas ("Distributed Hash Tables"ou DHT). Um "super-peer"é um nó numa rede "peer-to-peer"que opera tanto como um servidor para um conjunto de clientes, como um igual numa rede de "super-peers". As redes "super-peers"fornecem um balanceamento entre a eficiência da pesquisa centralizada, e a autonomia, balanceamento de carga e robustez aos ataques fornecidos pela pesquisa distribuída. Além disso, possuem vantagem inerentes à sua heterogeneidade do papel que representam (por exemplo, largura de banda, poder de processamento) entre pares. Acabam por formam uma arquitetura de duas camadas, em que medeiam as procuras, e só contactam os nós de procura com grande certeza de conterem a informação procurada. Mas mesmo numa arquitectura destas, a arquitectura de procura é baseada em algoritmos de "flooding".

Algoritmos de procura baseados em "flooding"apresentam custos de desperdício de largura de banda, pois apesar da mensagem ter apenas um destino, tem de ser enviada a todos os nós. Por outro lado, as mensagens podem ser duplicadas aumentando mais a carga na rede, bem como o correspondente processamento para eliminar a duplicação, sendo que nalguns caos, os pacotes duplicados podem circular para sempre, a menos que se usem outras medidas, como atribuir um Time To Live à mensagem ou controlo da topologia de rede sem loops, por exemplo.

As mensagens podem ser duplicadas na rede, aumentando ainda mais a carga na rede, bem como exigindo um aumento na complexidade do processamento para desconsiderar as mensagens duplicadas. Os pacotes duplicados podem circular para sempre, a menos que sejam tomadas certas precauções:

Uma Tabela de Hash Distribuída (DHT) é um tipo de estrutura de dados armazenada em vários nós que visa resolver este problema, fornecendo escalabilidade ao dividir por cada nó que se junta à rede, a estrutura DHT, e fornecendo tolerância a falhas quando um nó abandona a rede, acedendo à informação que esse nó antes detinha noutros nós da rede. utiliza uma função hash consistente, atribuindo a cada nó um identificador m-bit (normalmente 160-bit) usando uma função hash como SHA-1, e que permite que os nós entrem e saiam do rede com disrupção mínima, através da actualização e replicação da DHT.

Existem algumas limitações fundamentais, como o abandono de uma só vez de todos os nós, pois não teremos onde armazenar informação. Pelo que supomos aqui que os nós abandonam a rede a uma taxa lenta o suficiente para existirem nós na rede.

O mapeamento das DHT são frequentemente representados de duas maneiras gerais: como uma tabela hash e como uma árvore de pesquisa. Por isso debruçamos sobre os méritos de usar uma tabela hash como o Chord, e uma árvore de pesquisa binária como o Kademlia.

3.2 Chord v Kademlia

Estes algoritmos usam a abordagem básica das DHT, em que as chaves são inteiros (não negativos), por exemplo todos os inteiros de 160 bits, em que os pares chave,valor são armazenados nos nós com a chave mais próxima da sua. O que permite a um algoritmo de procura eficientemente localizar nos servidores próximos a ligação ao servidor pretendido.

Uma das diferenças do Kademlia para o chord, é precisamente a caraterística de cálculo simétrica do Kademlia, que permite a cada participante na rede Kademlia receber o número de pesquisas igual aos nós existentes na DHT. Sem esta característica, sistemas como o Chord não aprendem informação de "routing"dos pedidos de pesquisa que recebe para actualizar a DHT. Mais ainda, a natureza assimétrica do Chord leva a tabelas de routing rígidas. Cada entrada na tabela do Chord deve armazenar essa entrada precisamente no intervalo do espaço das chaves que precedem esse nó. E desta forma, qualquer nó nesse intervalo poderá estar a maior distância do que outros no mesmo intervalo. O Kademlia em contraste pode enviar um pedido de procura a qualquer nó de um intervalo da rede, permitindo selecionar a rota baseada na latência ou ainda enviar pedidos assíncronos paralelos, de igual modo a vários nós.

O Kademlia usa pesquisas de nós paralelas e assíncronas, para evitar atrasos provocados pelos timeouts de nós que não respondem. Também apresenta resistência a certos ataques de DoS, dada a sua estrutura descentralizada, pois mesmo que todo um conjunto de nós seja inundado ("flooded"), isso terá um efeito limitado na disponibilidade da rede, pois a rede é recuperada ao formar-se contornando os nós que não respondem.

Em conclusão, embora no limite ambos permitem encontrar uma chave em O(log n) passos numa rede com n nós, o Kademlia permite enviar pedidos de "lookup"paralelos e trocar a informação da DHT durante o "lookup", em vez de usar comunicações separadas. E por outro lado, retorna IDs de nós mais perto porque as "routing tables"(árvores binárias) estão organizadas para terem maior densidade na vizinhança do "peer". Como um dos principais critérios para estes dois protocolos é localizar os nós pretendidos rapidamente, e apresentando o Kademlia as vantagens descritas anteriormente em relação ao Chord, tomamos a decisão de usar o protocolo e algoritmo Kademlia para a implementação da aplicação da "timeline"descentralizada.

4 Implementação da solução

4.1 Arquitetura

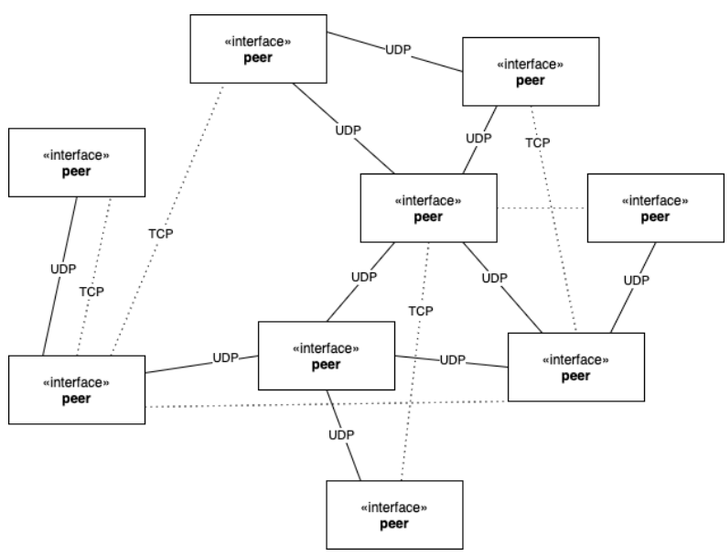

Figura 1: Arquitetura da rede da timeline descentralizada baseada no kademlia

A arquitetura utilizada foi uma arquitetura de rede descentralizada P2P entre os servidores da aplicação. Cada servidor mantém blocos de uma DHT Kademlia, que são usados para especificar a estrutura de uma rede virtual e efectuar procuras entre "peers", usando comunicação UDP. Este protocolo apesar de npermitir garantia de entrega de pacotes, também não necessita

estabelecer uma ligação permanente com o nó destino, realizando uma comunicação assíncrona, o que possibilita o rápido envio de pacotes para múltiplos destinos, sendo este um dos requisitos necessários para as aplicações P2P, nomeadamente o Kademlia. Já para a camada aplicacional de troca de mensagens da timeline, são usadas ligações TCP síncronas, para garantir entrega das mensagens entre os subscritores de uma timeline. Cada "peer"configura um "socket"TCP para actuar como servidor e escutar mensagens a receber, ou ligar-se às outras aplicações como cliente. Depois de dois sockets TCP se ligarem, a comunicação é bi-direcional.

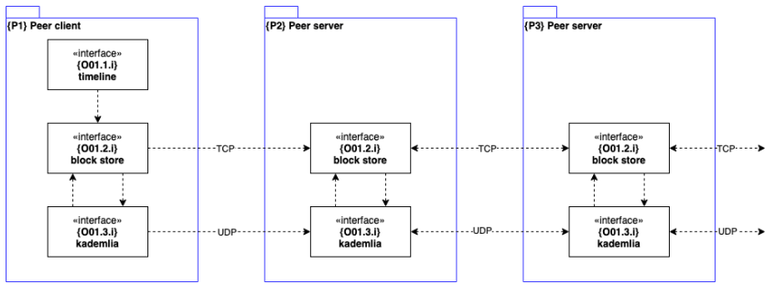

Figura 2: Modelo lógico da aplicação

Cada "peer"utiliza uma estrutura de software de camadas, que funciona tanto como cliente como servidor, permitindo partilha de serviços e dados sem a necessidade de um servidor central (figura 2). A camada mais alta fornece um interface para os utilizadores. Esta camada de apresentação implementa as funcionalidades de visualizar as mensagens da "timeline", subscrever a "timeline"doutros utilizadores, e publicar uma mensagem na "timeline", mapeando estas operações para operações de mais baixo nível. A próxima camada, é uma camada de persistência da "timeline"("block store") e de comunicação P2P, usando sockets TCP, para comunicar com os outros servidores desta camada, e processar o armazenamento, "cache"e replicação da timeline. Estes dados de persistência são guardados localmente num ficheiro JSON, que contém a timeline (lista com os pares ID, mensagem), os utilizadores subcritos pelo ID do nó (lista com as triplas ID, IP, porta TCP de cada utilizador subscrito, e o vector_clock do nó (dicionário com os pares ID, valor dos vector_clock desses utilizadores subscritos.

```
#Exemplo do objecto de dados da timeline, guardado num ficheiro JSON local:
"{\"messages\": [{\"id\": \"n11\", \"message\": \"m1\"}, {\"id\": \"n12\",
\"message\": \"m1\"}],
\"subscribed\": [{\"id\": \"n12\", \"ip\": \"192.168.1.69\", \"port\": 3012}],
\"vector_clock\": {\"n11\": 1, \"n12\": 1}}"
```

A camada de persistência local da "timeline"usa o kademlia da camada seguinte para identificar os nós responsáveis por guardar as mensagens da timeline de cada utilizador, para depois efetuar uma ligação P2P TCP para enviar ou ler mensagens da sua timeline.

A camada Kademlia implementa as operações do algoritmo, efectuando o armazenamento, cache e replicação de uma tabela hash distribuída (DHT). Os nós Kademlia formam uma rede virtual ou rede sobreposta formada pelos nós participantes, em cima da topologia da rede. Essa sobreposição é utilizada para descobrir e indexar os pares DHT da rede tornando o sistema P2P funcional independente da topologia da rede física. Esta rede armazena as informações das conexões dos utilizadores, viabilizando o serviço de comunicação P2P. Cada entrada valor da DHT contém um objecto de dados com os atributos do IP e porta TCP de cada nó, um dicionário "subscribers"com o par ID, IP port_UDP, identificando os utilizadores que subscreveram a "timeline"pertencente a cada nó, e um dicionário vector_clock com os pares ID, valor_VC de cada um desses utilizadores subscritores.

```
#Exemplo de uma entrada do par {key,value} da DHT Kademlia:
{ key: '179a5ca64acc2846dc863a49213e06545517ab22'
value: '{"ip": "192.168.1.69", "port": 3012,
"subscribers": {"n10": "192.168.1.69 3010", "n11": "192.168.1.69 3011"},
"vector_clock": {"n12": 1, "n10": 0, "n11": 0}}' }
```

O conteúdo da DHT é trocado diretamente sobre o protocolo IP usando sockets UDP. As operações Kademlia servem também para actualizar dinâmicamente a DHT, para a manter resiliente a falhas ou ataques (DoS). As operações de protocolo incluem ping() para verificar se o nó está ativo, store() para armazenar uma chave num nó, find_node() para retornar os nós mais próximos da chave pedida e find_value() para retornar o valor da chave pedida.

4.2 Funcionalidades

Cada peer é instalado de forma distribuída, implementando uma tabela hash distribuída Kademlia, e serviços para uma aplicação de "timeline". A linguagem selecionada para o desen-

volvimento do projeto foi python, em modo consola para permitir um desenvolvimento rápido de sistemas distribuídos, executado numa máquina local, utilizando as portas dos endereços de rede para diferenciar os serviços em cada "peer".

Inicialmente é colocado a correr o nó bootstrap, que automaticamente define e abre a porta 2000 UDP, onde todos os outros nós se irão ligar inicialmente. Por defeito, a versão desenvolvida toma por defeito o endereço local (0.0.0.0), partilhado por todos os peers, sendo diferenciados pelo par de portas UDP, TCP de cada um.

```
#Comando para iniciar o peer bootstrap:
python3 Bootstrap.py
```

De seguida são criados vários peers, abrindo vários processos consola, executando em cada um deles o comando de criação do "peer", definindo uma porta UDP para a rede Kademlia, e uma porta TCP para as ligações P2P "timeline"entre nós:

```
#Comando para iniciar um peer:
python3 peer.py <-pDHT port_dht> <-pP2P port_p2p> [<-ipB bootstrap_ip> <-pB bootstrap_port>]
#Exemplo para iniciar um peer com porta UDP 2010 e TCP 3010:
python3 peer.py -pB 2000 -pDHT 2010 -pP2P 3010
```

Cada serviço "peer"começa por pedir ao utilizador para inserir o seu ID, que vai ser usado como chave na DHT à qual vai associar o objecto de dados com os valores do IP e porta, e dicionários "subscribers"e "vector_clock"para o nó criado. E de seguida efectua a ligação ao nó "bootstrap"para entrar na rede, usando a sua ligação UDP à porta pré-conhecida do bootstrap.

Procede à leitura do objecto de dados do seu armazenamento local, gravado num ficheiro JSON com o nome "database+ID", (ID do utilizador). E constroi o menu com as seguintes funcionalidades disponiveis ao utilizador:

```
1 - Timeline
2 - Subscribe username
3 - Post message
4 - Show subscribers
0 - Exit
```

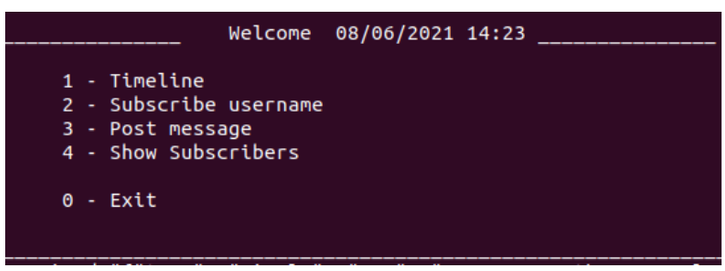

Figura 3: Menu

A funcionalidade "Timeline", mostra o conteúdo da lista "mensagens"em memória, que irá reunir o conteúdo da lista mensagens do objecto armazenado localmente, lido no arranque do nó, e as mensagens trocadas durante a sessão activa do nó (ou peer).

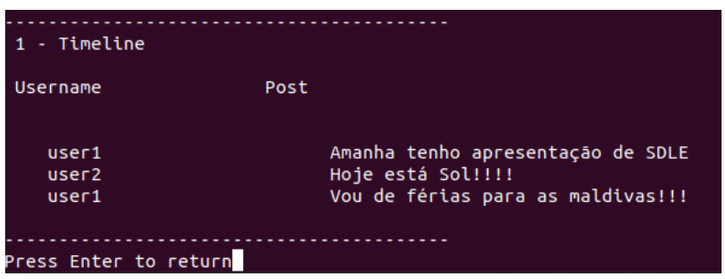

Figura 4: Exemplo da timeline do user3

A funcionalidade "Subscribe username", (subscrever utilizador), pede para indicar o ID do utilizador a subscrever. É feito um pedido à DHT para obter o objecto de dados associado a essa chave (valor do par chave, valor da DHT), pelo que o algoritmo Kademlia entra em funcionamento para procurar na rede o nó que contém a chave desse utilizador. Quando encontra, obtém o objecto de dados associado a essa chave, e caso o dicionário de subscritores desse objecto (dicionário "subcribers") não contenha o seu ID, então adiciona o seu par ID, IP porta_tcp.

A funcionalidade "Post message", pede ao utilizador para preencher a mensagem a publicar, e de seguida pesquisa a sua chave ID na DHT kademlia para obter o objecto de dados associado ao seu ID. Nesse objecto de dados incrementa uma unidade ao seu vector clock, e publica essa actualização na DHT. E para cada par ID, IP porta_TCP de cada subcritor existente no dicionário "subscribers", é estabelecida uma conexão TCP e enviada a mensagem para a timeline desse nó.

A funcionalidade "Show subscribers"consulta a rede Kademlia com o ID do utilizador, para obter o correspondente objecto de dados com o dicionário "subscribers", e mostrar o seu conteúdo.

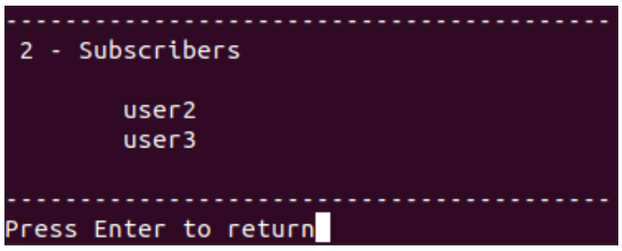

Figura 5: Lista de subscritores do user1

Terminando a execução do "peer"("Exit"), o programa procede à gravação num ficheiro JSON do estado das mensagens da timeline, lista de utilizadores que subscreveu, e os vector_clock de cada um dos utilizadores que subscreveu, que estavam em memória de execução.

A parte que necessitou mais esforço de desenvolvimento e que envolveu uma maior complexidade, foi a de recuperar a timeline de utilizador, depois de estar "offline"e perder mensagens de utilizadores que subscreveu. Esta recuperação da timeline ocorre sempre que o utilizador fica de novo "online"cada vez que inicia a execução do "peer". Por exemplo, quando um utilizador que estava "online", user2, e subscreveu alguns utilizadores, user1 e user3, e depois fica "offline", caso o user1 envie uma mensagem, o user2 não recebe a respetiva mensagem. Neste caso, como o user2 subscreveu o user1, e guardou localmente e de forma persistente os utilizadores que subscreveu, quando fica de novo "online"percorre os utilizadores que subscreveu, e verifica se o seu "vetor clock"está atualizado com o "vetor clock"do utilizador que subscreveu. Se o "vetor clock"se encontrar atualizado, significa que enquanto o user2 se encontrava "offline", o user1 não publicou nenhuma mensagem. Se o vetor clock se encontrar desatualizado, o user2 faz um pedido ao user1 para lhe enviar as mensagens que não recebeu enquanto estava "offline".

4.3 Framework de programação

Na procura por uma biblioteca Kademlia que estivesse bem documentada e permitisse reutilizar as suas funções para implementar a nossa aplicação, optamos por uma biblioteca que tem como referência para a sua implementação o paper Kademlia. Esta biblioteca [4], é uma implementação assíncrona em python que usa RPC sobre UDP (usa a biblioteca asyncio em Python3 para as comunicações assíncronas). Desta forma optamos pela linguagem python, para desenvolver a aplicação com as funcionalidades descritas na secção anterior, assente nestas bibliotecas que resumimos aqui e que fazem parte dos requisitos de instalação da aplicação desenvolvida, mas também utilizando outras relevantes, como a biblioteca "Threading"para correr as várias camadas da aplicação de forma concorrente, e a biblioteca "TCP.connection"para implementar as comunicações por sockets TCP para as ligações P2P de troca das mensagens da "Timeline".

```
kademlia==2.2.2
asyncio==3.4.3
rpcudp>=4.0.1
```

5 Discussão dos resultados

Inicialmente consideramos comparar os dois protocolos DHT, Chord e Kademlia, em termos de custo de largura de banda e latência na procura de um nó na rede. Para isso identificamos uma biblioteca Chord [5], que segue uma implementação simplificada do modelo Chord DHT e correspondente paper [6], neste caso na framework Dotnetcore em c. Chegou a ser implementado a camada do protocolo DHT (neste caso Chord), e uma parte da camada ("Timeline") de criação de nós para construir uma rede Chord. Testamos a entrada e saída de nós da rede, que nos pareceu bastante eficiente. No entanto por falta de tempo para construir a camada "Block store"e "Timeline"não concluimos esta aplicação. Alguma publicação não é conclusiva quanto à comparação destes protocolos de DHT, como por exemplo [7], que envolve experiências de apenas medir a procura de chaves na DHT, e que concluem obter performance semelhante, se parametros do algoritmo forem bem afinados, o que torna este processo mais complexo de comparar. Por isso centramo-nos na discussão das vantagens conceptuais dos algoritmos, que pendem para o Kademlia.

A implementação de "vector clocks"apenas garante a ordenação causal das mensagens da timeline. No entanto não foi considerada uma estratégia para associar as mensagens com um tempo físico.

Adotamos ainda uma estratégia de recuperação de mensagens da timeline enquanto o nó estivesse "offline", fazendo com que em cada arranque do nó, este iterasse a lista de utilizadores que o nó subscreveu (lista "subscribed"obtida do ficheiro JSON local), para comparar e actualizar a sua lista "vector_clock"e "messages"em memória (lidas também inicialmente do ficheiro JSON local).

6 Referências

(1) Maymounkov, Petar Eres, David. (2002). Kademlia: A Peer-to-peer Information System Based on the XOR Metric. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. 2429. 10.1007/3-540-45748-8_5.

(2) Distributed Hash Tables with Kademlia.
https://codethechange.stanford.edu/guides/guide_kademlia.html, Last accessed on 2021-06-07

(3) Yang, B. Garcia-Molina, Hector. (2003). Designing a super-peer network. Proceedings - International Conference on Data Engineering. 49 - 60. 10.1109/ICDE.2003.1260781.

(4) Python Kademlia. https://github.com/bmuller/kademlia, Last accessed on 2021-06-07

(5) Chordette. https://github.com/hexafluoride/Chordette, Last accessed on 2021-06-07

(6) Stoica, Ion Morris, Robert Liben-nowell, David Karger, David Frans, M. Dabek, Frank Balakrishnan, Hari. (2002). Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications.

(7) Li, Jinyang Stribling, Jeremy Gil, Thomer Morris, Robert Kaashoek, M.. (2004). Comparing the Performance of Distributed Hash Tables Under Churn. Lecture Notes in Computer Science. 10.1007/978-3-540-30183-7_9.