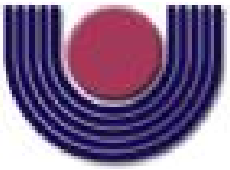


Unioeste - Universidade Estadual do Oeste do Paraná
CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS
Colegiado de Ciência da Computação
Curso de Bacharelado em Ciência da Computação

VCubeDHT - Uma DHT de Salto Único Baseada em um Hipercubo Virtual com Replicação

Alexandre Barreiro Neto

CASCAVEL
2016

Alexandre Barreiro Neto

VCubeDHT - UMA DHT DE SALTO ÚNICO BASEADA EM UM HIPERCUBO VIRTUAL COM REPLICAÇÃO

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação, do Centro de Ciências Exatas e Tecnológicas da Universidade Estadual do Oeste do Paraná - Campus de Cascavel

Orientador: Prof. Dr. Luiz A. Rodrigues

CASCAVEL
2016

Alexandre Barreiro Neto

VCUBEDHT - UMA DHT DE SALTO ÚNICO BASEADA EM UM HIPERCUBO VIRTUAL COM REPLICAÇÃO

Monografia apresentada como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação, pela Universidade Estadual do Oeste do Paraná, Campus de Cascavel, aprovada pela Comissão formada pelos professores:

Prof. Luiz Antonio Rodrigues (Orientador)
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Guilherme Galante
Colegiado de Ciência da Computação,
UNIOESTE

Prof. Clodis Boscarioli
Colegiado de Ciência da Computação,
UNIOESTE

Cascavel, 2 de março de 2017

AGRADECIMENTOS

Primeiramente a minha família por sempre me apoiarem nas decisões, não medindo esforços para que eu chegasse até esta etapa de minha vida.

Aos meus amigos e colegas do curso, que compartilharam as mesmas dificuldades e felicidades para chegar até onde estamos.

Quero também agradecer à Universidade Estadual do Oeste do Paraná, mais especificamente ao corpo docente do curso de Ciência da Computação e ao meu orientador, professor Luiz Antonio Rodrigues, pelos conselhos, apoio e disponibilidade.

De maneira geral, a todos que de alguma forma colaboraram para eu chegar até aqui.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

CAN	*Content-Addressable Network*
DHT	*Distributed Hash Table*
DiVHA	*Distributed Virtual Hypercube Algorithm*
DNS	*Domain Name Server*
EDRA	*Event Detection and Propagation Algorithm*
GRAS	*Grid Reality and Simulation*
Hi-ADSD	*Hierarchical Adaptive Distributed System-level Diagnosis*
HPC	*High-Performance Computing*
IP	*Internet Protocol*
MPI	*Message Passing Interface*
P2P	*Peer-to-Peer*
TI	Tecnologia da Informação
TTL	*Time-to-Live*
RAID	*Redundant Array of Independent Disks*
XBT	*eXtended Bundle of Tools*

Lista de Símbolos

θ	Classe assintótica
$C_{i,s}$	Função responsável por criar o hipercubo
d	Dimensão do sistema
m	Número de bits do identificador
N	Número de nodos pertencentes ao sistema
P	Ponto em um espaço de coordenadas

Sumário

Resumo

Um sistema *peer-to-peer* (P2P) permite o compartilhamento eficiente de dados entre uma grande quantidade de usuários sem a necessidade de coordenadores centralizados, como é encontrado em sistemas baseados na arquitetura cliente/servidor. No entanto, questões relacionadas a distribuição de dados em muitos computadores distintos e o seu consecutivo acesso, são dificuldades encontradas nesse tipo de sistema. Para solucionar esse problema, foram propostas as redes *peer-to-peer* baseadas em tabelas *hash*, denominadas DHTs (*Distributed Hash Table*). Essa abordagem busca oferecer um resultado eficiente e escalável para localização de informações sobre uma rede P2P. Além disso, pode-se construir sistemas distribuídos garantindo a descentralização, escalabilidade e tolerância a falhas. Mesmo assim, torna-se necessário garantir a disponibilidade dos dados armazenados, normalmente fazendo uso de técnicas de replicação. Este trabalho apresenta uma abordagem de DHT de salto único denominada VCubeDHT. Sua implementação baseia-se em um hipercubo virtual distribuído que utiliza o algoritmo de diagnóstico VCube (*Virtual Hypercube*) para criação da rede de sobreposição. Para prover disponibilidade, o VCubeDHT utiliza técnicas que permite a replicação dos dados fragmentados entre os vizinhos de cada nodo. Além dos algoritmos propostos, são apresentados resultados das simulações aplicadas sobre diferentes cenários, avaliando a solução proposta com o HyperDHT. Na análise do Protocolo de Entrada o VCubeDHT mostrou-se mais eficiente em todos os casos de testes. Na análise da Reorganização do Sistema e Disponibilidade, o VCubeDHT obteve, na maior parte dos casos de testes, um pior desempenho com arquivos menores que cem megabytes. Porém, mostrou-se mais eficiente com arquivos maiores que cem megabytes.

Palavras-chave: Sistemas Distribuídos, Redes Peer-to-Peer, Simulação, Replicação de Dados.

Capítulo 1

Introdução

Um sistema distribuído é uma coleção de dois ou mais processos que se comunicam e coordenam suas ações através de troca de mensagens. Um dos principais motivos para construir e utilizar sistemas distribuídos é o compartilhamento de recursos [Coulouris, Dollimore e Kindberg 2007]. Esse, por sua vez, pode ser utilizado sobre diferentes arquiteturas de rede através de aplicações existentes para esse fim. As arquiteturas de rede amplamente estudadas atualmente são: cliente/servidor e P2P (*Peer-to-Peer*).

Em um modelo cliente/servidor, o papel do cliente é realizar requisições de um serviço ao servidor. Este, por sua vez, ao receber as requisições, as processa e responde ao solicitante, fornecendo ou não o serviço requerido. Nota-se que nesse modelo o servidor se encontra centralizado na rede e os clientes estão ao seu redor realizando requisições. Uma vantagem desse modelo é o armazenamento de dados centralizado. Desse modo, as atualizações são fáceis de serem administradas. No entanto, sua principal desvantagem é a ocorrência de falhas de um servidor, comprometendo toda a rede.

Diferentemente do modelo anterior, as redes P2P são sistemas distribuídos constituídos de nós interconectados que possuem a característica de se auto-organizar, com o objetivo de realizar o compartilhamento de recursos. Além disso, as redes P2P surgiram originalmente como sendo uma alternativa ao modelo cliente/servidor [Androutsellis-Theotokis e Spinellis 2004].

Em um modelo de sistema P2P puro não há a noção de que um par (*peer*) seja apenas cliente ou servidor, ambos fazem papel tanto de cliente como de servidor. Além disso, a estrutura de uma rede P2P pode ser classificada em dois tipos, não-estruturada e estruturada [Androutsellis-Theotokis e Spinellis 2004]. Na primeira, a disposição dos arquivos não possui vínculo com a topologia sobreposta. Já na segunda, os arquivos são alocados em posições

específicas da rede e não de forma aleatória. Desse modo, as buscas são realizadas de forma eficiente e escalável. Neste trabalho será utilizada uma rede estruturada que utiliza tabelas *hash* distribuídas, mais conhecida como DHT (*Distributed Hash Table*).

As DHTs são redes P2P que mantém informações distribuídas sobre múltiplos nodos do sistema. Elas podem construir sistemas distribuídos garantindo a descentralização, escalabilidade e tolerância a falhas [Koppe 2013]. A primeira é atendida pois não há coordenação central no sistema, como ocorre no modelo cliente/servidor. Já a segunda é satisfeita devido a capacidade de auto-organização das DHTs. Por fim, a última é garantida de acordo com o modelo utilizado na construção da rede de sobreposição no sistema.

As DHTs armazenam dados em uma abordagem chave/valor e possuem operações de *get*, *put* e *lookup*. A operação de *get(chave)* recupera um valor associado a chave informada no parâmetro. O *put(chave, valor)* armazena a informação estipulada. E por último, a função *lookup(chave)* retorna a referência ao par responsável pela chave informada no parâmetro.

A fim de realizar as consultas, as DHTs fornecem uma função similar às encontradas em tabelas *hash* tradicionais, onde as chaves são distribuídas pelos nodos da rede. A chave define a porção da tabela *hash* que um nodo é responsável. Para utilizar a função comentada, ou seja, encaminhar uma consulta a partir do nodo solicitante até o responsável pela chave procurada, as DHTs utilizam estruturas conhecidas como tabelas de roteamento. A cada consulta encaminhada para um novo nodo é dito que ocorreu um salto.

Em relação ao número de saltos, as DHTs podem ser classificadas como sistemas de único ou múltiplos saltos. Nos sistemas de único salto (*Single-Hop*), cada nodo possui uma tabela de roteamento contendo todos os nodos do sistema. Já nos sistemas de múltiplos saltos (*Multi-Hop*), cada nodo utiliza um conjunto reduzido de tabelas de roteamento e necessitam que uma consulta seja roteada por vários nodos do sistema. Neste trabalho será implementado um sistema DHT de salto único, denominado VCubeDHT baseado no HyperDHT.

O HyperDHT [Koppe 2013] utiliza a rede de sobreposição construída pelo algoritmo DiVHA (*Distributed Virtual Hypercube*) [Bona et al. 1995]. O DiVHA é um algoritmo de diagnóstico distribuído hierárquico que cria uma topologia virtual baseada em um hipercubo virtual. Quando executado em um sistema composto por *N* nodos sua latência é de no máximo $\log_2^2 N$ rodadas de teste. Porém, o número de testes executados no pior caso é quadrático [Ruoso 2013].

Ao invés de utilizá-lo como algoritmo de diagnóstico distribuído, neste trabalho será implementado o VCube, que possui latência máxima de $\log_2^2 N$.

O VCube é uma solução de diagnóstico distribuído que constrói e mantém os processos do sistema com base em uma topologia virtual baseada no hipercubo. O hipercubo virtual é criado e mantido de acordo com as informações de diagnósticos obtidas por meio de um sistema de monitoramento de processos, descrito em [Ruoso 2013]. A cada execução do VCube, cada processo é capaz de testar outros processos no sistema para verificar se estão corretos ou falhos.

Uma característica dos sistemas distribuídos é a ocorrência de falhas, sejam elas de hardware ou de software. Neste contexto, é fundamental desenvolver soluções distribuídas tolerantes a falhas, isto é, aplicações que continuam sua execução corretamente mesmo na presença de falhas [Rodrigues 2014].

A tolerância a falhas pode ser obtida por redundância dos dados. Neste contexto, pode-se discutir sobre a replicação de dados em sistemas distribuídos. A replicação de dados permite que, caso a máquina onde uma das réplicas está situada eventualmente torne-se indisponível, o serviço se manterá disponível utilizando-se de outra réplica localizada em outra máquina. Portanto, afim de aplicar os conceitos relacionados a replicação dos dados, neste trabalho será implementado um módulo no qual gerencie as réplicas do sistema VCubeDHT, aplicando uma melhor técnica de replicação baseada no contexto do problema. Ela visa manter um nível de replicação, e, consequentemente, disponibilidade dos dados baseado na visão geral da rede.

Um dos principais problemas na replicação de dados está relacionado à complexidade em manter a consistência dos mesmos. No entanto, na implementação do módulo de replicação de dados, será mantido apenas o gerenciamento das réplicas no sistema, não sendo consideradas as atualizações feitas nos dados replicados.

O estabelecimento de uma estrutura de sistemas distribuídos para realizar testes em larga escala é custoso. A vantagem de utilizar simuladores é pelo fato de apresentarem resultados satisfatórios, o que muitas vezes seria inviável realizar no mundo real, além de permitir a repetição dos mesmos, evitando o consequente desperdício de recursos. Outra vantagem é o provisionamento dinâmico de recursos e sua escalabilidade. Conforme um estudo anterior [Neto e Rodrigues 2015], diversas ferramentas foram analisadas seguindo requisitos necessários para elaboração deste projeto. O SimGrid se destacou das demais principalmente por sua

simplicidade de uso, configuração e escrita de códigos.

O principal objetivo deste trabalho foi a implementação do VCubeDHT - uma DHT de salto único baseada na topologia criada e mantida pelo VCube. Além disso, buscou-se garantir a disponibilidade dos dados armazenados no sistema. Para atingir esses objetivos, foi feito um levantamento das técnicas de replicação de dados utilizadas em diferentes sistemas, buscando empregar, em uma nova abordagem, características daquelas que mais se adéquam as necessidades do trabalho. Após a implementação da proposta (utilizando o simulador SimGrid), foram definidos diferentes cenários para que, posteriormente, fosse feita a análise dos resultados. Nessa etapa, foi possível comparar o impacto das técnicas de replicação de dados implementadas pelo VCubeDHT e HyperDHT. De modo geral, o VCubeDHT se mostrou mais eficiente em todos casos de testes, quando comparado ao HyperDHT no Protocolo de Entrada. No entanto, nas análises de Reorganização do Sistema e Disponibilidade, o HyperDHT se mostrou mais eficiente na maior parte dos casos de testes, onde a replicação foi feita com arquivos com tamanhos menores que cem *megabytes*. Já com arquivos maiores de cem *megabytes*, o VCubeDHT se mostrou mais eficiente em todos os cenários. Além disso, a presente proposta garante uma maior disponibilidade dos dados quando os participantes da rede não falham instantaneamente.

O restante do texto está organizado da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica, destacando os conceitos fundamentais de sistemas distribuídos e tolerância a falhas. O Capítulo 3 trata sobre redes P2P, bem como o uso de tabelas *hash* distribuídas baseadas nesse tipo de sistema. O Capítulo 4 apresenta os principais conceitos relacionados a replicação de dados, exibindo técnicas existentes aplicadas nesse contexto. O Capítulo 5 aborda as principais características do VCubeDHT. O Capítulo 6 apresenta os resultados obtidos em relação às simulações realizadas. Por fim, o Capítulo 7 exibe as principais conclusões sobre o trabalho, bem como sugestões de trabalhos futuros.

Capítulo 2

Sistemas Distribuídos

Este Capítulo apresenta os principais conceitos relacionados aos sistemas distribuídos. A Seção 2.1 apresenta a definição, vantagens e desvantagens, bem como questões relacionadas ao sincronismo. Em seguida, na Seção 2.2 são apresentados os conceitos de tolerância à falhas aplicado a esse tipo de sistema.

2.1 Sistemas Distribuídos

Um sistema distribuído é aquele no qual o hardware e o software estão localizados em computadores conectados por uma rede, como a Internet, que se comunicam e coordenam suas ações enviando mensagens entre si a fim de realizar uma determinada tarefa [Coulouris, Dollimore e Kindberg 2007].

A comunicação entre esses computadores podem ocorrer basicamente de três formas: um-para-um (*unicast*), um-para-muitos (*multicast*) e um-para-todos (*broadcast*) [Hadzilacos e Toueg 1994]. Na comunicação um-para-um, cada enlace conecta um par de processos a fim de enviar uma mensagem para apenas um destinatário de cada vez. Na comunicação um-para-muitos, uma mensagem pode ser enviada para um grupo de processos. Em contrapartida, em redes *broadcast* tem-se a ideia de que a rede conecta todos os processos pertencentes ao sistema, onde cada processo poderá enviar mensagens simultaneamente para todos os outros participantes da rede.

Um dos principais motivos de construir e utilizar sistemas distribuídos é o compartilhamento de recursos. Com base nesse, a construção de sistemas distribuídos torna-se um ponto importante. Esses recursos compartilhados podem estar relacionados a unidades de armazenamento,

dados, arquivos, dentre outros.

Juntamente ao compartilhamento de recursos, a utilização de sistemas distribuídos visa combinar a escalabilidade à transparência do sistema ao usuário. Na escalabilidade podem ser consideradas a escalabilidade de tamanho, geográfica e administrativa [Tanenbaum e Steen 2007]. A primeira possibilita incrementar o número de recursos aos usuários do sistema. A segunda busca ampliar a área física de alcance do sistema. Já a terceira possibilita o gerenciamento, mesmo possuindo muitas organizações administrativas diferentes.

Para obter escalabilidade em um sistema distribuído há um conjunto de técnicas que podem ser aplicadas, sendo elas técnicas de distribuição e replicação [Tanenbaum e Steen 2007]. A primeira consiste em particionar um dado e espalhar suas partes pelo sistema. Já a segunda procura utilizar múltiplas cópias de um dado distribuído pela rede, a fim de manter sua disponibilidade caso o responsável por este dado venha a falhar.

A transparência em um sistema distribuído está relacionado à capacidade desse identificar-se aos usuários como sendo um único sistema computacional [Tanenbaum e Steen 2007]. Há alguns tipos de transparências que podem ser buscadas, como transparência de acesso, localização e replicação, por exemplo. A transparência de acesso procura ocultar diferenças na representação de dados e no modo que ocorre acesso a um determinado recurso. Já a transparência de localização busca omitir o lugar no qual o recurso está localizado. Por fim, a transparência de replicação procura ocultar do usuário a ideia de que um recurso é replicado.

Em um modelo de sistema distribuído são considerados dois fatores: a velocidade de processamento e o atraso relacionado a troca de mensagens nos canais de comunicação [Rodrigues 2014]. Em sistemas caracterizados como síncronos, há limites conhecidos para estes atributos, ou seja, dependendo da implementação pode-se definir como falho um processo que não respondeu a uma requisição em um determinado período de tempo. No entanto, em sistemas assíncronos esses limites não existem. Determinar um processo falho de um processo muito lento nesse tipo de sistema é uma tarefa impossível [Fischer, Lynch e Paterson 1985].

Os sistemas distribuídos são conhecidos também pela não-necessidade do controle global dos processos de forma centralizada [Coulouris et al. 2013]. Os recursos são utilizados de forma concorrente e a sincronização entre os processos pode ser feita pela troca de mensagens.

Dentre as vantagens na utilização de sistemas distribuídos, nota-se que a área possui uma

variedade de desafios. Alguns desses estão relacionados aos conceitos de falhas em sistemas distribuídos. Um exemplo disso é a ocorrência de atrasos na comunicação em uma rede. Há dificuldades em diferenciar o congestionamento na rede da ocorrência de falhas nos processos, pois, como comentado, em sistemas assíncronos os limites relacionados a troca de mensagens nos canais de comunicação não existem ou não são conhecidos.

O uso de sistemas distribuídos torna-se um ponto importante. No entanto, um sistema distribuído precisa necessariamente ser confiável, ou seja, deve possuir condições de acessar determinada informação que não esteja prejudicialmente modificada. Além disso, é necessário construir sistemas capazes de se recuperar automaticamente das falhas sem prejudicar sua execução.

2.2 Tolerância a Falhas

Uma falha em um sistema distribuído ocorre quando um integrante do sistema torna-se inacessível, afetando parcialmente a operação de outros componentes do sistema. Dessa forma, após sua detecção, o sistema deverá se recuperar de forma automática, evitando que o usuário perceba a ocorrência da mesma.

Um sistema tolerante a falhas está fortemente relacionado a um sistema confiável. A confiabilidade engloba vários requisitos úteis que um sistema distribuído deve seguir, os principais são disponibilidade, confiabilidade, segurança e capacidade de manutenção [Tanenbaum e Steen 2007]. A disponibilidade é a capacidade do sistema permanecer em correto funcionamento a todo momento. A confiabilidade está relacionada com o funcionamento do sistema (sem interrupções) durante um período de tempo. A segurança garantirá que após a ocorrência de falhas, nada catastrófico acontecerá. Já a capacidade de manutenção está relacionada a facilidade de manutenção caso um componente do sistema tenha se tornado inacessível.

2.2.1 Modelo de Falhas

O término inesperado de um programa muitas vezes não é imediatamente percebido por outros componentes com o qual ele se comunica. O modelo de falhas define o modo como uma falha pode se manifestar e se comportar em um sistema, de forma a entender seus efeitos e consequências [Coulouris et al. 2013]. Várias pesquisas tem sido realizadas sobre falhas em sistemas distribuídos, e de acordo com [Hadzilacos e Toueg 1994], essas falhas podem ser

classificadas como falhas por omissão, falhas arbitrárias e falhas de sincronização.

As falhas por omissão (*omission fault*) são aquelas onde um processo ou canal de comunicação deixa de exercer ações a que era designado. Nos processos, eles omitem-se de efetuar o envio e recebimento de mensagens. Nos canais de comunicação, a falha é detectada quando a entrega das mensagens entre os processos não é concretizada, causando a perda de mensagens.

As falhas arbitrárias ou bizantinas (*byzantines faults*) são aquelas em que um processo ou canal de comunicação pode realizar ações que não foram designados. No caso dos processos eles podem realizar um processamento indesejado. Sobre os canais de comunicação, o conteúdo da mensagem pode ser corrompido ou mensagens inexistentes podem ser enviadas, por exemplo.

Diferentemente das demais, as falhas de temporização (*timing fault*) são aquelas relacionadas aos sistemas distribuídos síncronos, em que a resposta do processo está fora de um intervalo de tempo pré-determinado, podendo estar adiantada ou atrasada.

2.2.2 Detecção de Falhas e Diagnóstico do Sistema

Um sistema tolerante a falhas deve possuir uma visão consistente dos componentes que fazem parte da sua rede. Para tanto, a detecção de falhas é fundamental. Em um grupo de processos, os membros devem ser capazes de detectar outros participantes falhos no sistema.

Uma abordagem bastante conhecida é a que realiza, de maneira ativa, a troca de mensagens entre os processos (para as quais se espera resposta) questionando-os se estão ativos no sistema. Essas mensagens se assemelham as requisições de *ping* realizadas entre os processos. Após um período de tempo, os processos que não responderam essas mensagens podem ser considerados como falhos. A fim de manter os processos informados em relação aos demais, de forma a garantir a consistência das informações e realizando tais operações de modo eficiente, serão utilizados algoritmos de diagnósticos distribuídos.

Diagnóstico Adaptativo e Distribuído

O algoritmo Adaptive-DSD proposto e implementado por [Bianchini e Buskens 1992] realiza diagnósticos de forma distribuída e adaptativa, sem a necessidade de uma unidade central de controle. Cada participante do sistema recebe um identificador numérico de forma a construir uma topologia de rede baseada em anel. Nesse algoritmo o número de nodos falhos é limitado

a *n-1*, sendo *n* o número de participantes do sistema. Com isso, é possível que um único nodo realize o diagnóstico dos outros *n-1* nodos do sistema.

Para a distribuição das informações do diagnóstico, os nodos devem testar seus sucessores até encontrar um nodo que não esteja falho para que esse continue testando os demais. As informações reportadas por nodos considerados falhos não são validadas. A Figura 2.1 representa um sistema de 8 nodos nos quais os nodos 1, 4 e 5 são considerados falhos.

Inicialmente o nodo 0 testa o nodo 1. Como não obteve resposta após um determinado período de tempo, considera o nodo 1 como falho e testa o nodo 2. Em seguida o nodo 2 testa o nodo 3. De forma similar o nodo 3 testa o nodo 4 e descobre a falha no mesmo. O mesmo ocorre ao testar o nodo 5. Com isso, o nodo 3 testa o nodo 6. Quando o nodo 7 testa o nodo 0, a informação de falha do nodo 1 encontrada anteriormente pelo nodo 0 é repassada ao nodo 7.

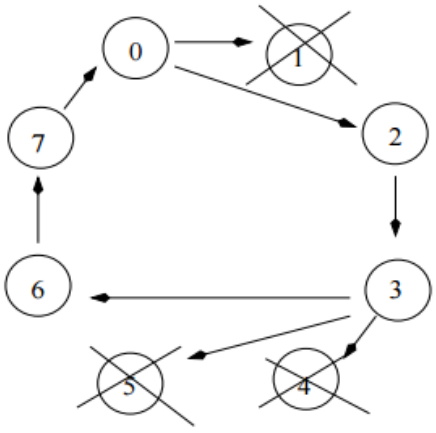


Figura 2.1: Grafo de testes do Algoritmo Adaptative-DSD para um sistema com 8 nodos
Fonte: [Weber 2008]

Em geral, os algoritmos de diagnósticos distribuídos são avaliados em relação ao número de testes necessários por rodadas de teste e também a latência de propagação de um evento. Uma rodada de teste é o período de tempo no qual os nodos não falhos executam seus testes. A latência de um sistema é definida como o tempo necessário para que todos os nodos não falhos identifiquem a ocorrência de um evento. Um evento é a mudança de um estado de um nodo, isto é, quando passa de um estado não falho a um estado falho ou vice-versa [Bona et al. 1995].

A latência de diagnóstico do algoritmo Adaptive-DSD é proporcional ao número de nodos que compõe o anel. Além disso, o desempenho desse algoritmo em relação a latência pode ser afetado caso o sistema apresente uma alta ocorrência de entradas/saídas de participantes ou

também caso a rede contenha um número muito grande de nodos. O problema ocorre pois é necessário percorrer todo o anel para realizar o diagnóstico de todos os participantes da rede. Portanto, em um momento qualquer pode-se identificar um componente como falho e assim notificar aos demais. Porém, o processo falho pode se restabelecer na rede logo em seguida, fazendo com que o diagnóstico apresente informações inconsistentes do sistema.

Algoritmos Hierárquicos de Diagnóstico Distribuído

O algoritmo Hi-ADSD (*Hierarchical Adaptive Distributed System-level Diagnosis*) [Duarte e Nanya 1998] diminui a latência de propagação de eventos do algoritmo Adaptive-ADSD. Nesta nova abordagem os nodos são organizados em *clusters* progressivamente maiores e os testes são feitos de maneira hierárquica baseados em uma estratégia de dividir e conquistar.

A topologia do sistema Hi-ADSD é baseada em um hipercubo. O hipercubo possui características topológicas importantes como simetria, diâmetro logarítmico e boas propriedades de tolerância a falhas [Krull, Wu e Molina 1992]. O número de nodos do sistema é definido por 2^d, sendo *d* a dimensão do hipercubo. A Figura 2.2 representa um hipercubo com dimensão $d = 3$.

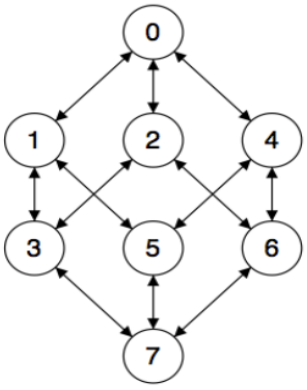


Figura 2.2: Grafo do Algoritmo Hi-ADSD para um sistema com 8 nodos
Fonte: [Ruoso 2013]

Os processos de testes são executados a cada intervalo no qual cada nodo testará os nodos de determinado *cluster* até encontrar um nodo sem falha. Após efetuar um teste sobre um nodo pertencente a um *cluster*, são obtidas informações de diagnóstico sobre todos os nodos daquele *cluster*. Os membros de cada *cluster s*, bem como a ordem em que os processos serão testados por um processo *i* são dados pela função $C_{i,s}$ [Duarte e Nanya 1998]. A Figura 2.3 lista os

resultados da função $C_{i,s}$ para um sistema de $d = 3$ dimensões.

s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Figura 2.3: Resultado da $C_{i,s}$ para um sistema de *d=3* dimensões
Fonte: [Duarte e Nanya 1998]

Considere um sistema de $d = 3$ dimensões. Na primeira rodada o nodo 0 teste o nodo 1 do *cluster* cujo tamanho é 1. Na segunda rodada de testes testa o *cluster* de tamanho 2 na qual contém os nodos 2 e 3. Na próxima rodada, testa o *cluster* de tamanho 4 e obtém diagnóstico sobre os nodos 4, 5, 6 e 7. Esse procedimento é repetido para os demais nodos. A latência do algoritmo no pior caso é $\log_2^2 N$ rodadas de teste, sendo *N* o número de nodos que compõem o sistema. No entanto, o número de testes executados no pior caso é quadrático.

O Algoritmo DiVHA

O Algoritmo DiVHA (*Distributed Virtual Hypercube Algorithm*) [Bona et al. 1995] é um algoritmo de diagnóstico distribuído e hierárquico que cria uma topologia virtual baseada em um hipercubo. Ele tem com objetivo determinar o conjunto de testes a ser realizado pelos nodos do sistema de acordo com o seu estado atual. Esse algoritmo calcula os enlaces do hipercubo de forma a permitir a auto-organização dos nodos mantendo o diâmetro logarítmico do hipercubo. Ele é considerado dinâmico pois um nodo pode falhar e se recuperar constantemente [Bona et al. 1995]. Do mesmo modo que o algoritmo Hi-ADSD, afim de organizar os nodos e posteriormente realizar os testes, o DiVHA utiliza o conceito de *clusters*, porém neste, é mantido apenas um único nodo testador em cada *cluster*.

No DiVHA a construção do hipercubo é baseada na função $C_{i,s}$ do algoritmo Hi-ADSD e é empregado o uso de *timestamps*. Um *timestamp* é um contador que é inicializado em 0 e posteriormente incrementado. Ele tem como objetivo apresentar o estado de um nodo através da ocorrência de eventos. Nesse algoritmo, um nodo é considerado como não-falho se seu *timestamp* for impar, caso contrário, é considerado como falho.

A topologia formada pelo DiVHA é baseada no grafo *H(S)*, de acordo com a Figura 2.2, na qual representa um hipercubo composto de 8 nodos sem falha. Os nodos são representados pelos

vértices e os enlaces de comunicação entre eles representados pelas arestas. Quando um nodo é diagnosticado como falho, o grafo *H(S)* deixa de ser um hipercubo e assim, são adicionadas novas arestas para preservar duas principais propriedades, sendo elas: o diâmetro logarítmico e o número total de arestas no grafo H(S) nunca é maior que $N \log_2 N$ [Ruoso 2013]. A adição de arestas é feita de maneira distribuída entre os nodos baseando-se nas distâncias do hipercubo.

O Algoritmo DiVHA determina o estado de todos os participantes do sistema em no máximo $\log_2 N$ rodadas de testes e o número máximo de testes por rodada é de $N * \log_2 N$. Um dos principais fatores que influencia negativamente a execução do algoritmo é a adição de arestas extras, onde cada nodo deve calcular quais testes os nodos devem realizar de acordo com o estado atual do sistema, sendo necessário efetuar várias buscas no grafo que representa o sistema.

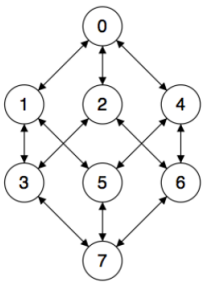
O Algoritmo VCube

O algoritmo VCube [Ruoso 2013] propõe uma nova estratégia de testes para o algoritmo Hi-ADSD que visa garantir uma quantidade máxima de testes logarítmica preservando o mesmo limite para a latência. Este algoritmo é uma solução de diagnóstico distribuído que constrói e mantém os processos do sistema com base em uma topologia virtual baseada em hipercubo. O hipercubo virtual é criado e mantido de acordo com as informações de diagnósticos obtidas por meio de um sistema de monitoramento de processos, descrito em [Ruoso 2013]. A cada execução do VCube, cada processo é capaz de testar outros processos no sistema para verificar se estão corretos ou falhos. Um processo é correto se a resposta ao teste realizado for recebida corretamente dentro de um intervalo de tempo.

Assim como no algoritmo DIVHA os processos são estabelecidos em *clusters*. Para cada rodada de testes um processo *i* testa o primeiro processo sem falha *j* na lista de processos de cada *cluster s* e obtém informação sobre os processos naquele *cluster*. Os membros de cada *cluster s*, bem como a ordem em que os processos serão testados por um processo *i* são dados pela função $C_{i,s}$. Ela determina que o processo *i* testa o processo $j \in C_{i,s}$, somente se o processo *i* é o primeiro correto em $C_{j,s}$. Portanto, todo processo é testado uma única vez em cada rodada. Isso garante uma latência de diagnóstico de $log_2 N$ rodadas na média e $\log_2^2 N$ no pior caso.

A Figura 2.4 ilustra a organização lógica de um hipercubo de três dimensões. Em $C_{(0,s)}$, o processo 0 testa o processo 1 quando *s* é 1, depois testa o processo 2 quando *s* é 2, e por fim, quando *s* é 3, testa o processo 4 e é notificado com informações atualizadas sobre o estado dos

demais processos.



s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Figura 2.4: Organização Hierárquica do VCube de *d=3* dimensões
Fonte: [Rodrigues 2014]

2.3 Considerações

Na Seção 2.1 foi possível analisar os principais conceitos relacionados aos sistemas distribuídos, bem como identificar o compartilhamento de recurso como sendo um motivo essencial para se construir e utilizar esse tipo de sistema. Além disso, foi citado a Internet como sendo um grande exemplo de sistema distribuído.

Já na Seção 2.2 introduziu-se o conceito de falha aplicado sobre um sistema distribuído, apresentando o modelo de falhas. Este modelo classifica os diversos tipos de falhas que podem se manifestar sobre qualquer sistema. Além disso, observou-se que um sistema tolerante a falhas é aquele que continua sua execução mesmo na presença de falhas e está, de certa forma, associado a um sistema confiável, que, por sua vez, atende os requisitos de disponibilidade, confiabilidade, segurança e capacidade de manutenção.

Por fim, para detecção e diagnóstico de falhas, foram visualizadas estratégias utilizadas pelos algoritmos de diagnóstico distribuído. Notou-se que o VCube se destaca dos demais algoritmos, apresentando uma nova estratégia de testes mais eficiente. Portanto, decidiu-se por utilizar este algoritmo para efetuar o diagnóstico dos participantes do sistema.

Capítulo 3

Redes Peer-to-Peer

Esse Capítulo tem como objetivo abordar uma visão geral das redes *peer-to-peer*, bem como o uso de tabelas *hash* distribuídas sobre esse tipo de sistema, tomadas como base para o desenvolvimento desse trabalho.

3.1 Introdução

As redes P2P (*Peer-to-Peer*) são sistemas distribuídos constituídos de nós interconectados que compartilham recursos entre si e possuem capacidade de auto-organização tolerante a falhas. Elas surgiram originalmente como sendo uma alternativa ao modelo cliente/servidor [Androutsellis-Theotokis e Spinellis 2004].

Ao contrário do modelo cliente/servidor, onde alguns computadores são dedicados a servirem outros, um sistema P2P é um tipo de rede onde cada estação possui capacidades e responsabilidades equivalentes. Nesse sistema, há duplas de hospedeiros, chamados de pares (*peers*), que são conectados por um enlace e comunicam-se diretamente entre si [Kurose e Ross 2012].

3.2 Modelo Cliente/Servidor

Em um modelo cliente/servidor, o papel do cliente é realizar requisições de um determinado serviço ao servidor. Esse, por sua vez, ao receber as requisições, processa as mesmas e responde ao solicitante, fornecendo ou não o serviço requerido. Nota-se, que nesse modelo o servidor está centralizado e os clientes estão ao seu redor realizando requisições, como mostra a Figura 3.1.

Uma vantagem da utilização desse modelo é o armazenamento de dados centralizado, onde as atualizações dos dados serão facilmente administradas. Porém, tomando como exemplo um

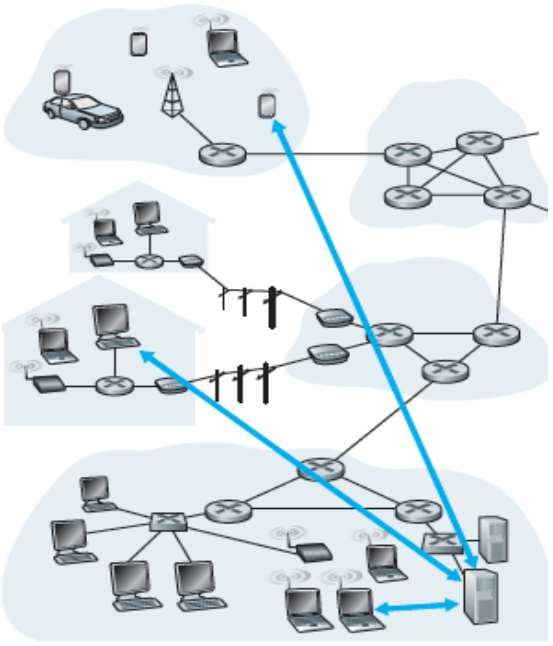


Figura 3.1: Modelo de rede Cliente/Servidor
Fonte: [Kurose e Ross 2012]

sistema de distribuição de arquivos, uma desvantagem é facilmente observada, pois o servidor deverá enviar uma cópia do arquivo para cada cliente da rede. Isso faz com que sua carga de trabalho cresça de acordo com o número de clientes. Outra desvantagem, sendo esta, a de maior relevância, é a ocorrência de falhas de um servidor, que pode comprometer toda a rede.

3.3 Modelo Peer-to-Peer

Baseado no mesmo exemplo de distribuição de arquivos, em um modelo de sistema P2P a carga de trabalho estará distribuída sobre cada participante da rede, e este pode redistribuir parte do arquivo recebido para os demais, fazendo com que o trabalho seja divido entre cada participante do sistema. A Figura 3.2 apresenta um exemplo de modelo de rede P2P.

A estrutura de um sistema P2P é baseada em uma rede de sobreposição (*overlay network*). Essa, é uma rede virtual localizada sobre a camada física de rede. Os *peers* que a compõem podem ser vizinhos entre si, porém, na camada física da rede isto nem sempre é verdade.

De acordo com [Androutsellis-Theotokis e Spinellis 2004], um sistema P2P pode ser classificado como não-estruturado ou estruturado. No primeiro, a disposição dos arquivos não possui vínculo com a topologia sobreposta, ou seja, há dificuldade para localizar um arquivo comparti-

lhado. Assim, para essa busca, pode-se tornar necessário a utilização de métodos de força bruta, como o de inundação da rede. Já no segundo, os arquivos são alocados em *peers* cuja posição seja específica na rede. Desse modo, as buscas são realizadas de forma eficiente e escalável. Exemplos comuns de redes P2P estruturadas são as que utilizam tabelas *hash* distribuídas.

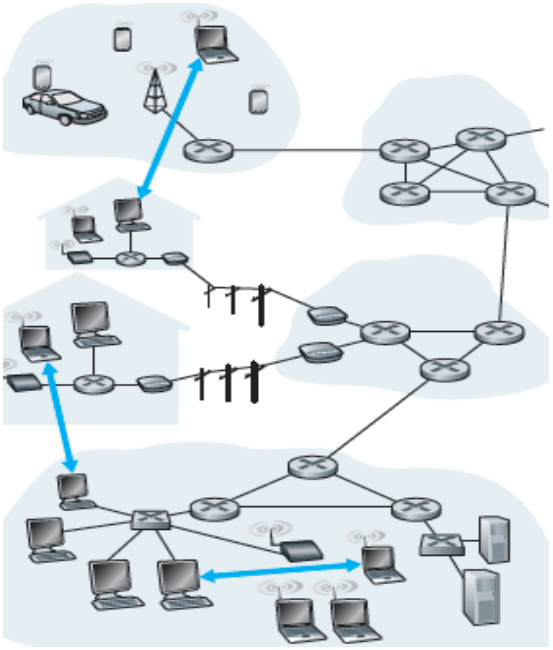


Figura 3.2: Modelo de rede P2P
Fonte: [Kurose e Ross 2012]

As principais características sobre o modelo de sistema P2P é a possibilidade de fornecer grande capacidade de armazenamento, ciclos de CPU, dentre outros recursos computacionais, impondo um custo baixo relacionado a escalabilidade e em relação a entrada e saída de *peers* da rede [Kshemkalyani e Singhal 2008].

Uma aplicação bastante conhecida e utilizada sobre sistemas P2P é o compartilhamento de arquivos. O arquivo compartilhado pode ser uma nova versão do sistema operacional, um arquivo de música, dentre outros exemplos. Uma das primeiras aplicações que se enquadra nessa categoria é o Napster [Napster 2016], e na época, se tornou o mais popular sistema que fornecia tecnologia P2P, oferecendo o *download* de músicas de forma gratuita. Após o Napster, outras tecnologias P2P surgiram, como o GNutella [GNutella 2016] e o atual BitTorrent [BitTorrent 2016]. No entanto, vários problemas surgiram com a ascensão dessas ferramentas, que causaram grande preocupação a várias empresas pelo motivo dessas alegarem a distribuição

ilegal de arquivos com direitos autorais por meio desse tipo de aplicação.

Como comentado, um sistema P2P possui como característica sua auto-organização. Portanto, sabe-se que os *peers* pertencentes a rede podem se conectarem ou desconectarem-se da mesma. Este evento é caracterizado como *churn* e é objeto de estudo por diversos pesquisadores. De acordo com [Melo, Vieira e Libório 2012], em um sistema que possui replicação de dados, o *churn* pode induzir a replicação descontrolada de dados do sistema, reduzindo a capacidade efetiva de armazenamento do mesmo. Esse tipo de evento será mais detalhado nos capítulos seguintes.

3.4 Redes P2P baseadas em Tabelas Hash Distribuídas

Atualmente uma variedade de empresas de tecnologia da informação (TI) buscam proporcionar serviços para milhares de usuários, como o fornecimento de novas versões de um *software* que pode ser baixado por muitos clientes. Com isso, a necessidade de desenvolver tecnologias que forneçam serviços em grande escala cresce constantemente. Para um sistema ser escalável, deseja-se que este não possua dependências de pontos únicos de acesso e que possua como característica sua auto-organização a medida em que novos servidores são adicionados a rede. Por fim, outro requisito importante é que este sistema seja tolerante a falhas, pois quanto maior o sistema, maior será a probabilidade da ocorrência de falhas. Esta seção apresenta as principais características de uma estrutura de dados que engloba grande parte das propriedades comentadas, esta estrutura é denominada de DHT (*Distributed Hash Table*).

As DHTs são estruturas de dados que armazenam informações distribuídas sobre múltiplos nodos do sistema. Para esse armazenamento, as DHTs utilizam o conceito chave/valor, mais conhecido por sua utilização em tabelas *hash* tradicionais. O principal serviço fornecido por uma DHT é a operação de pesquisa, que retorna um determinado valor associado a uma chave. Do mesmo modo, as DHTs possuem operações de inserção e exclusão de itens [Ghodsi 2006].

Para apresentar um exemplo de utilização das DHTs, considera-se um banco de dados distribuído onde o armazenamento das informações é baseada no conceito chave/valor. Para esse exemplo, a chave é considerada como um nome de uma tabela qualquer e o valor representa o endereço IP do nodo que possui essa informação (por exemplo, (Pessoas, 200.17.123.38)). Assim, nota-se que por meio de uma chave é possível obter a informação sobre qual nodo

possui esta informação armazenada. Após obter o endereço IP do nodo responsável pela informação, o nodo solicitante pode enviar mensagens de requisição ao nodo cujo endereço IP é 200.17.123.38.

As redes DHTs podem construir sistemas distribuídos garantindo a descentralização, escalabilidade e tolerância a falhas. A descentralização está relacionada a ausência de coordenação central, como ocorre no modelo cliente/servidor. A escalabilidade está relacionada a capacidade de auto-organização das DHTs. E por fim, a tolerância a falhas é garantida pelo modelo utilizado para construção da rede de sobreposição no sistema [Koppe 2013].

Diversos pesquisadores buscam estudar as DHTs, visto que as técnicas aplicadas a esse tipo de sistema são amplamente adotadas por aplicações P2P, em destaque para o compartilhamento de arquivos. No entanto, há uma grande quantidade de trabalhos que propõe diversas implementações de DHTs, cada um com suas características, o que dificulta a escolha de uma DHT ideal para um determinado sistema. Porém, as observações são feitas em relação a latência para propagação de eventos e o uso da rede para manutenção do sistema DHT. A latência está relacionada ao tempo necessário para que todos os nodos disponíveis descubram a ocorrência de um novo evento. Algumas implementações mais conhecidas de DHT são: Tapestry [Zhao, Kubiatowicz e Joseph 2001], Pastry [Rowstron e Druschel 2001], Chord [Stoica et al. 2001] e CAN [Ratnasamy et al. 2001].

As DHTs utilizam uma estrutura chamada tabela de roteamento para encaminhar uma consulta a partir do nodo solicitante até o responsável pela chave procurada. Cada nodo pertencente ao sistema possui uma tabela. Ela é composta por um subconjunto de entradas que servem de referência a outros nodos, de forma a associar um nodo a sua chave e a seu respectivo endereço na rede. A cada consulta encaminhada para um novo nodo é dito que ocorreu um salto. Algumas implementações de DHT utilizam tabelas de roteamento completas, ou seja, cada nodo possui conhecimento de todos os demais participantes do sistema. Redes DHTs que utilizam essa abordagem são definidas como sistemas de salto único, pois necessitam de um único salto para a consulta. Em outras implementações, cada nodo faz uso de tabelas de roteamento parciais e necessitam que uma consulta seja roteada por diversos nodos do sistema. As DHTs que fazem uso dessa abordagem são definidas como sistemas de múltiplos saltos. Geralmente, quanto maior a tabela de roteamento menor será o número de saltos necessários para realizar a

consulta, e vice-versa [Ghodsi 2006]. Com base nessas tabelas de roteamento são estabelecidas conexões lógicas na rede.

3.5 Tabelas Hash Distribuídas de Múltiplos Saltos

Enquanto algumas DHTs mantêm tabelas de roteamento com informações sobre todos os nodos do sistema, alguns sistemas optam pela utilização de tabelas de roteamento parciais que são divididas entre os nodos da rede. As soluções que usam essa abordagem visam minimizar o tráfego de manutenção das tabelas de roteamento, com prejuízo para a latência das consultas.

Esta seção define as principais características das DHTs de múltiplos saltos, apresentando duas implementações, sendo elas: CAN [Ratnasamy et al. 2001] e Chord [Stoica et al. 2001].

3.5.1 CAN

O CAN (*Content-Addressable Network*) [Ratnasamy et al. 2001] é um sistema completamente distribuído que não necessita de controle centralizado. Este sistema possui uma estrutura baseada em um espaço cartesiano virtual multi-dimensional de coordenadas. Esse espaço é completamente lógico, não possuindo qualquer relação com a estrutura física do sistema. Além disso, o espaço de coordenadas é particionado de forma dinâmica entre os nodos do sistema.

Os nodos pertencentes ao sistema CAN mantêm informações acerca de um pequeno número de nodos adjacentes, armazenando o endereço IP e as coordenadas desses nodos. O número de saltos necessários em uma consulta geralmente será maior que o número de saltos em um sistema que utiliza uma única tabela de roteamento para cada nodo.

O sistema CAN fornece uma performance de roteamento de $\theta\ (d.n^{(1/d)})$ saltos, onde *d* é a dimensão do espaço de coordenadas e *n* é o número de nodos no sistema. Considerando um espaço cartesiano particionado em *n* regiões equivalentes, a distância média entre dois nodos quaisquer é de $\theta((d/4)(n^{1/d}))$ [Ratnasamy et al. 2001].

Este sistema possui características no qual assemelham seu funcionamento ao funcionamento de uma tabela *hash*, ou seja, utiliza o conceito de chave/valor e foi projetado para ser escalável, tolerante a falhas e possuir a capacidade de se auto-organizar.

O espaço cartesiano é utilizado para o armazenamento dos pares chave/valor. Afim de armazenar um par (chave/valor) qualquer, a chave é armazenada em um ponto *P* no espaço de

coordenadas utilizando uma função *hash*. O nodo que irá conter essa informação (chave/valor) é aquele que possui a zona em que o ponto *P* se encontra. Para recuperar um determinado valor cuja entrada corresponda a esse, a operação *lookup* irá executar a mesma função *hash* para mapear a chave no ponto *P* e, em seguida, recuperar o valor correspondente a entrada informada. Caso o ponto *P* não seja de propriedade dos nodos vizinhos ao nodo solicitante, o pedido deverá ser encaminhado até atingir o nodo cujo local se encontra *P*.

Para melhorar a disponibilidade dos valores, o CAN permite mapear um par chave/valor em diversos pontos diferentes no espaço de coordenadas, ou seja, o par chave/valor é replicado sobre vários nodos do sistema.

A Figura 3.3, adaptada de [Ratnasamy et al. 2001], apresenta um exemplo de uma consulta. Esta, é inicializada no nodo 1 e o objetivo é encontrar o valor cujo ponto *P* está localizado no nodo 7. Inicialmente, o nodo 1 analisa sua própria tabela de roteamento, verificando qual vizinho é o mais próximo de *P*. Então, sendo o nodo 4 o mais próximo, o nodo 1 irá encaminhar uma consulta ao nodo 4. O processo irá se repetir até que a consulta atinja o nodo cujo local está presente o ponto *P*, nesse exemplo as consultas irão se repetir até que atinja o nodo 7.

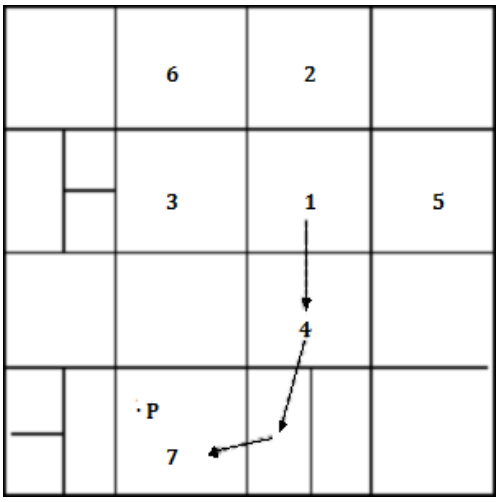


Figura 3.3: Exemplo de roteamento do nodo 1 para o nodo 7 no sistema CAN
Fonte: [Ratnasamy et al. 2001]

Como comentado, o espaço cartesiano de coordenadas é particionado de forma dinâmica no sistema. O CAN permite que a inserção de um novo nodo ocorra em uma mesma região previamente populada por outro nodo. No entanto, para participar do sistema CAN, o nodo deverá ter conhecimento do endereço IP de outro nodo participante da rede. Após esse processo, o

sistema irá dividir a região populada em duas partes, na qual irá alocar na outra metade o nodo que realizou a solicitação de entrada. Por fim, será feito a atualização das tabelas de roteamento dos nodos adjacentes ao novo participante. A Figura 3.4, adaptada de [Ratnasamy et al. 2001], apresenta um exemplo de entrada do nodo 8 na rede CAN. Após o ingressante possuir conhecimento sobre o nodo 1, o espaço cartesiano destinado ao nodo 1 é dividido e as tabelas de roteamento dos nodos vizinhos a esse são atualizadas.

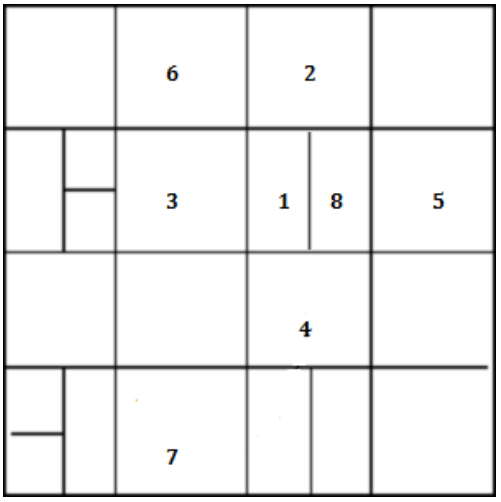


Figura 3.4: Exemplo de entrada do nodo 8 em um sistema CAN
Fonte: [Ratnasamy et al. 2001]

Para lidar com um nodo que irá deixar o sistema CAN, primeiramente é necessário identifica-lo. Após essa etapa, o espaço cartesiano deixado pelo nodo é rearranjado para um dos nodos adjacentes àquele que saiu e, por fim, é realizado a atualização das tabelas de roteamento dos demais nodos que ainda se encontram no sistema.

3.5.2 Chord

O Chord [Stoica et al. 2001] é um modelo de sistema que proporciona rápido mapeamento de chaves entre os nodos do sistema utilizando uma variante de *hashing* consistente. Uma *hash* consistente atribui a cada nodo e chave um identificador de *m* bits usando como base uma função *SHA-1* [Wang, Yin e Yu 2005]. O identificador do nodo é selecionado através do *hash* de seu endereço IP, e o identificador da chave é escolhido baseado no *hash* da mesma. Esse modelo de sistema busca resolver os problemas mais conhecidos em aplicações P2P, como o balanceamento de carga, descentralização, escalabilidade e disponibilidade dos dados.

A utilização de uma função *hash* distribuída espalha as chaves sobre os nodos de maneira ordenada e balanceada em um anel, conhecido como *Chord Ring* de tamanho 2^m, sendo *m* o número de bits dos identificadores. Além disso, mantém o equilíbrio do sistema de forma a resolver o problema relacionado ao balanceamento de carga.

Uma chave *K* é atribuída ao primeiro nodo cujo identificador é maior ou igual a *K*. A busca de um nodo sucessor é garantida através da função *sucessor(K)*. A função retorna o primeiro nodo sucessor (em sentido horário), em relação a aquele que possui a chave *K* no sistema. De forma a garantir a descentralização, o sistema é totalmente distribuído e nenhum nodo possui maior relevância em relação a outro.

A Figura 3.5, adaptada de [Stoica et al. 2001], apresenta um exemplo básico de requisição *lookup* em uma topologia Chord. O número de bits de cada identificador é de tamanho *m = 3*, sendo $2^3 = 8$ o número total de nodos. O sistema possui 3 chaves e o armazenamento das mesmas está sendo feito pelos os nodos 1, 3 e 6, respectivamente.

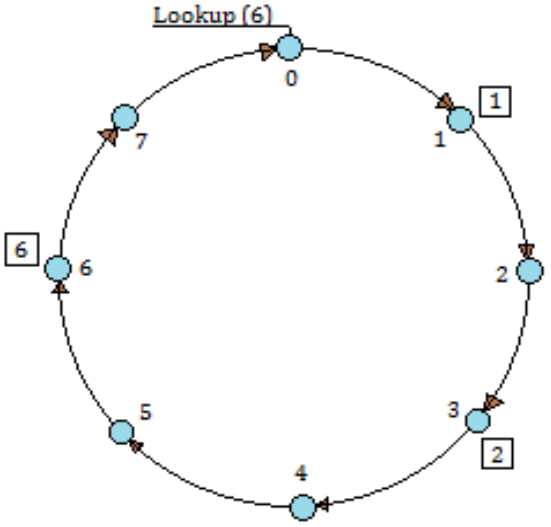


Figura 3.5: Operação básica de *lookup* em um sistema Chord com 8 nodos e 3 chaves
Fonte: [O Autor 2016]

A requisição *lookup(K)* busca o nodo que armazena a chave *K* previamente informada. A função compara *K* com a chave do nodo sucessor aquele que realizou a requisição. A cada solicitação, o nodo alvo a responde de modo a apresentar a chave armazenada por ele. O processo é repetido pelo encaminhamento da solicitação através do anel em sentido horário até encontrar o nodo responsável pela chave *K*. No exemplo, o nodo 1 realiza a requisição *lookup(6)*. Porém,

antes de realizar a função *lookup*, o nodo 1 primeiramente busca identificar qual nodo responsável pelo armazenamento da chave 6, para isso ele invoca a operação *findsucessor(6)*. Com base nisso, a consulta é roteada entre o nodo inicial e final, sendo os nodos 1 e 6 respectivamente. Quando a solicitação chega ao ultimo nodo, sendo esse o que possui a chave, a busca retorna de maneira recursiva a origem.

O sistema Chord utiliza uma tabela de roteamento conhecida como *finger table*. Ela é utilizada principalmente para reduzir a latência das consultas. Cada nodo possui uma tabela com no máximo *m* entradas, sendo *m* a quantidade de bits da chave obtida pela função *hash*. Cada entrada é formada pelo identificador Chord e pelo endereço IP associado ao nodo. Em uma rede com *N* nodos é necessário apenas $\log N$ requisições para a procura de uma chave *K* qualquer. Para a manutenção das tabelas de roteamento é executado periodicamente em segundo plano pelo sistema uma função de atualização, visto que as tabelas podem se tornar inconsistentes devido à entrada e saída de nodos da rede. De acordo com [Stoica et al. 2001], esse sistema provê que o nodo responsável por determinada chave *K* seja encontrado mesmo se o sistema esteja em constante mudança, garantindo a disponibilidade e veracidade dos dados.

Para participar de um sistema Chord é necessário que o nodo solicitante descubra o seu sucessor e remapeie algumas chaves associadas ao seu sucessor para o novo nodo. Já a saída de um nodo da rede indica que todas as suas chaves serão reassociadas ao seu nodo sucessor.

3.6 Tabelas Hash Distribuídas de Salto Único

A principal característica de um sistema DHT de salto único é a eficiência em retornar a referência do nodo responsável por uma chave, previamente informada na função *lookup(chave)*.

Mesmo realizando consultas rápidas, os sistemas DHTs de salto único precisam considerar o fato de que quanto maior for a quantidade de dados armazenados na tabela de roteamento em cada nodo, maior será a demanda de comunicação para sua manutenção de acordo com a entrada e saídas de nodos do sistema [Monnerat 2010].

Esta seção apresenta as principais características das implementações de DHTs que utilizam a abordagem de salto único, sendo elas: D1HT [Monnerat 2010] e OneHop DHT [Gupta, Liskov e Rodrigues 2004] e HyperDHT [Koppe 2013].

3.6.1 D1HT

O D1HT [Monnerat 2010] é um sistema que se propõe realizar consultas em um único salto. De modo semelhante ao sistema Chord, a associação das chaves aos nodos é feita de forma ordenada, no sentido horário, formando uma topologia baseada em anel.

Esse sistema utiliza o conceito *hash* consistente que associa uma chave a cada participante do sistema. A chave e o endereço IP do nodo são identificadores e ambos são criptografados pela função *SHA-1* [Wang, Yin e Yu 2005]. As chaves podem, opcionalmente, serem replicadas para outros nodos do sistema.

No sistema D1HT cada nodo possui uma tabela de roteamento com endereços IP de todos os participantes do sistema. Cada consulta pode ser resolvida com um único salto desde que a tabela de roteamento não esteja desatualizada.

Para atualização das tabelas de roteamento o sistema D1HT utiliza o EDRA (*Event Detection and Propagation Algorithm*). Diferente de algumas outras formas de propagação de informação (por exemplo, *broadcast*), o EDRA é capaz de notificar qualquer evento a todos participantes da rede em tempo logarítmico, garantindo balanceamento de carga e baixa demanda da rede. Além disso, se adapta as mudanças no comportamento do sistema, possuindo uma arquitetura puramente P2P e auto-organizável [Monnerat 2010].

Para a entrada de um novo participante no sistema, este deve conhecer ao menos um nodo pertencente a rede e logo em seguida, seguir o protocolo de adesão proposto no D1HT. Esse protocolo define quatro regras a serem seguidas. A primeira descreve que a etapa inicial é a transferência da tabela de roteamento (realizada pelo nodo sucessor) aquele novo participante. Somente após essa etapa se inicia sua disseminação na rede. A segunda define que a disseminação da adesão a um par deverá ser realizada segundo o algoritmo EDRA. A terceira define que o sucessor ou predecessor do novo participante deverá garantir que todas as notificações de eventos que estão sendo realizadas durante sua adesão sejam entregues para ele. A quarta e ultima regra define que o protocolo de adesão deverá ser apto de receber extensões.

Na disseminação de eventos no sistema D1HT, cada nodo deve armazenar (durante um intervalo de tempo, ajustado dinamicamente) uma quantidade de eventos, onde, em seguida, será iniciado o processo de disseminação desses eventos por meio de mensagens de manutenção. Nesse processo, o sucessor do nodo que sofreu o evento deverá reportá-lo em todas as mensa-

gens. Cada mensagem possui um contador denominado TTL (*Time-to-Live*) que determina o tempo de vida da mesma. Se por algum motivo um nodo qualquer não receber a mensagem de seu predecessor em um determinado período de tempo, esse nodo irá verificar se seu predecessor ainda compõe o sistema e, caso contrário, irá assumir que ele saiu do sistema. De acordo com [Monnerat 2010], esse procedimento busca minimizar as demandas de banda para manutenção das tabelas de roteamento.

3.6.2 OneHop DHT

O OneHop DHT [Gupta, Liskov e Rodrigues 2004] é um sistema que propõe realizar consultas em um único salto e com consumo de banda razoável. No entanto, devido a possíveis falhas que podem ocorrer na primeira consulta (após a entrada ou saída de nodos), o sistema compromete-se a resolver as consultas com no máximo dois saltos. Assim como o Chord e o D1HT, a associação das chaves é uniformemente distribuída e ordenada por seus identificadores, de forma a criar uma topologia baseada em anel. As chaves (identificadores) atribuídas aos nodos são mapeadas pela função *hash SHA-1* de 128 bits. Uma característica que a difere desses sistemas é o conhecimento que um nodo possui de seu predecessor e sucessor na rede.

Nesse sistema, cada nodo possui uma tabela de roteamento que mantém informações completas sobre todos participantes da rede. Dois fatores que são levados em consideração pelo sistema estão relacionados a atualização da tabela de roteamento e a disseminação da ocorrência de um evento (entrada ou saída de um nodo) na rede.

A fim de verificar se os nodos estão posicionados corretamente no sistema, cada nodo executa periodicamente uma rotina, chamada de rotina de estabilização, realizada de modo que cada nodo envie mensagens ao seu predecessor e sucessor. Após o envio, os nodos (predecessor e sucessor) verificarão se o nodo solicitante está posicionado corretamente, isto é, se ele é sucessor do nodo predecessor e predecessor do nodo sucessor. Se um nodo não responder as solicitações que são enviadas repetidamente a ele durante um período de tempo, este será julgado como inacessível ou falho.

A fim de manter baixo atraso de notificação de eventos e pouco consumo de banda, o OneHop DHT utiliza uma estrutura hierárquica bem definida no sistema. Essa hierarquia divide o espaço de identificadores circular de 128 bits em k intervalos contíguos balanceados,

chamados de *slices* [Gupta, Liskov e Rodrigues 2004]. Cada *slice* possui um líder (chamado de lider de *slice*) e é dividido internamente em partes com mesmo tamanho, chamados de *unidades*. Assim como os *slices*, cada *unidade* também possui um líder (chamado de líder de *unidade*). Em ambos casos, os líderes são escolhidos de forma dinâmica no sistema. Os líderes de *slice* possuem como função principal a organização e propagação de todas as informações sobre os eventos ocorridos em seu *slice* para outros líderes de *slices*.

O sistema OneHop DHT possui um mecanismo de tolerância a falhas. Se por algum motivo uma consulta falhar em seu primeiro salto, ela deverá ser reencaminhada de modo a ser resolvida em dois saltos. Os líderes de *slices* também podem falhar, portanto, o sistema se comporta de forma a selecionar o nodo sucessor ao líder falho para que o sucessor seja o novo líder.

Uma descrição mais detalhada a respeito de notificações de eventos, tolerância a falhas e escalabilidade no sistema OneHop DHT pode ser facilmente encontrada em [Gupta, Liskov e Rodrigues 2004].

3.6.3 HyperDHT

O HyperDHT [Koppe 2013] propõe uma solução na qual o procedimento de localização de uma informação ou um objeto distribuído na rede seja realizado de forma rápida, eficiente e escalável. Esse sistema visa utilizar uma infra-estrutura disponibilizada através do algoritmo DiVHA para prover serviços na qual caracterizam-na como sistema de salto único. Como comentado, o DiVHA é um algoritmo de diagnóstico distribuído hierárquico que cria uma topologia virtual baseada em um hipercubo.

Além das funções básicas fornecidas pelas DHTs (*lookup*, *put* e *get*), o HyperDHT implementa mecanismos para o posicionamento e busca dos objetos na rede P2P, em que associa identificadores (chaves) aos nodos e objetos da rede. Além disso, realiza o particionamento e mapeamento das chaves da tabela *hash* entre os nodos utilizando técnicas de *hash* consistente. O sistema também implementa protocolos de entrada de novos integrantes da rede, que determina, de acordo com a topologia atual da rede, uma posição para inserção do novo nodo. Por fim, o HyperDHT implementa funções para replicar as informações ou objetos dispostos na rede.

Sendo o HyperDHT uma DHT de salto único, os nodos desse sistema possuem tabelas de roteamento completa, ou seja, armazenam informações dos demais integrantes da rede. Cada

entrada da tabela faz referência a um identificador de um nodo ao seu endereço de rede.

O sistema é associado a um grafo cuja topologia formada é baseada em um hipercubo virtual criado pelo algoritmo DiVHA. Cada vértice do grafo representa uma posição na rede de sobreposição que pode ou não ser ocupada. Os vértices estão estabelecidos em *clusters* e o número de vértices pertencentes a um *cluster* é definida pela dimensão d do hipercubo que modela o HyperDHT e é determinado por 2^d. O sistema permite que os nodos podem tanto sair como participar da rede. Um vértice é considerado ocupado se ele contém um nodo, caso contrário é classificado como vértice vazio. Com a participação de vários nodos na rede, esses podem se comunicar diretamente.

Um exemplo de comunicação é a troca de mensagens, onde os nodos notificam uns aos outros sobre a ocorrência de um evento na rede. Como visto em [Koppe 2013], um evento é caracterizado pela mudança de estado, seja de um vértice ou de um nodo. O procedimento que verifica a ocorrência de um evento é caracterizado como teste. Os testes são realizados em rodadas por meio de mensagens trocadas entre os nodos adjacentes de forma a classificá-los como disponíveis ou indisponíveis, de acordo com um intervalo de tempo pré-determinado. Aqueles que respondem as requisições são considerados disponíveis. Os indisponíveis são aqueles que não as respondem. Visando apresentar a disponibilidade dos vértices, a Figura 3.6 apresenta um grafo do HyperDHT composto de 8 vértices. Aqueles em branco são considerados disponíveis, já aqueles representados em preto estão indisponíveis.

Uma das principais análises feitas em relação a eficiência de um algoritmo de diagnóstico distribuído utilizado nas DHTs está relacionado a sua latência. Esse termo é definido como o número de rodadas de testes necessárias para que todos os nodos disponíveis descubram a ocorrência de um evento. A latência do algoritmo DiVHA, quando executado em um sistema constituído de *N* nodos é de no máximo $\log_2^2 N$ rodadas de teste. Entretanto, o número de testes executados no pior caso é quadrático [Ruoso 2013].

A fim de manter o balanceamento de carga entre cada participante do sistema, o HyperDHT utiliza a função criptográfica *SHA-1* de 160 bits para calculo das chaves, formando um conjunto de 2^{160} possíveis chaves. A associação é feita de modo que cada vértice recebe uma faixa de valores determinada por $[i * 2^{160-d}, ..., (i+1) * 2^{160-d}]$, onde *i* representa o índice do vértice e *d* a dimensão do hipercubo. Quando todos os vértices estão sendo ocupados, a distribuição das

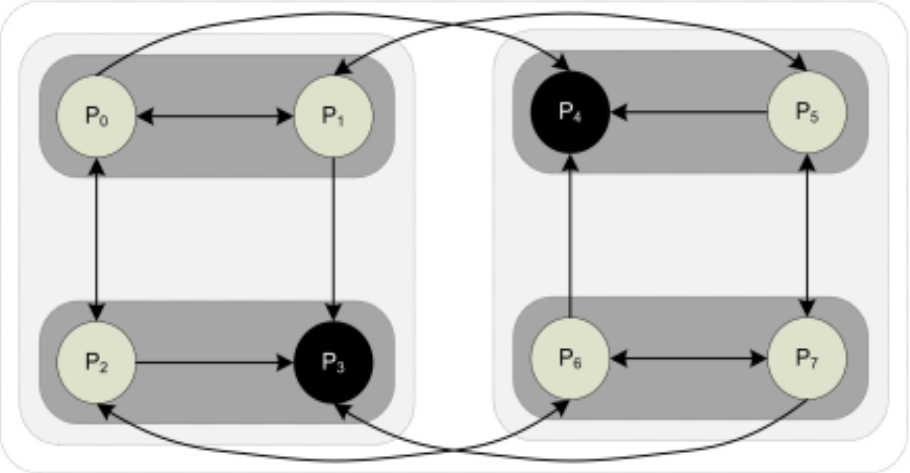


Figura 3.6: Grafo do HyperDHT para um sistema de 8 vértices ocupados
Fonte: [Koppe 2013]

chaves para cada nodo se dá de forma 1:1, de acordo com os índices *i* do vértice e do nodo, ou seja, o nodo n_i será responsável pelo espaço de chaves associado ao vértice v_i. [Koppe 2013].

O sistema HyperDHT utiliza abordagens para inserção e remoção de participantes na rede, são eles: protocolo de entrada e protocolo de saída. O protocolo de entrada garante posicionar o nodo solicitante em um local da rede onde ele é mais necessário. Essa é uma das características que o diferencia das demais DHTs. O local definido é baseado de acordo com a distribuição de chaves entre os nodos do sistema. A fim de manter a distribuição de carga entre os nodos, o local mais adequado para posiciona-lo é aquele em que haja uma grande distribuição de chaves a um único nodo. Além disso, após a entrada de um novo nodo, o algoritmo deve garantir que a propriedade de salto único ainda seja garantida.

Inicialmente, o sistema é possui dimensão *d = 1*, sendo composto de um único participante. Para participar da rede HyperDHT é necessário que o nodo requisitante tenha conhecimento de no mínimo um outro nodo pertencente ao sistema. Dessa forma, o nodo solicitante faz uma requisição ao nodo conhecido. A requisição feita baseia-se na procura de um identificador de um vértice, bem como o endereço do nodo que ele deve contactar para dividir o espaço de chaves. De acordo com sua visão completa do sistema, o nodo solicitado busca encontrar o local mais adequado para o posicionamento do nodo solicitante. Como visto, busca-se a posição de um nodo que possua um maior domínio de chaves. Se não existir um vértice vazio, é iniciado

o procedimento para aumento do sistema. Esse aumento consiste em criar novos vértices no sistema, isto é, o número de vértices da rede será duplicado [Koppe 2013].

O protocolo de saída oferece duas maneiras para saída de um nodo do sistema HyperDHT. A primeira define que o nodo deve permanecer no sistema durante d rodadas de teste após anunciar sua saída aos demais nodos do sistema. A permanência durante d rodadas é feita afim de garantir a disponibilidade dos valores em um único salto. Somente após esse período, o nodo poderá deixar o sistema. Já na segunda abordagem, o nodo não comunica sua saída e simplesmente deixa a rede. A replicação implementada nesse sistema define que, além das chaves, as solicitações que antes eram destinadas aquele nodo, também deverão serem redirecionadas ao nodo que armazena as réplicas do antigo participante da rede [Koppe 2013].

Além de oferecer protocolos de entrada e saída, o sistema fornece mecanismos de replicação de informações (ou objetos armazenados) entre os participantes da rede. Esse mecanismo busca manter a disponibilidade dos dados mesmo na ausência do nodo responsável [Koppe 2013].

O mecanismo implementado baseia-se na replicação de um determinado par chave/valor associado a um nodo para outros participantes da rede. No processo de replicação, é definido o número k de réplicas, sendo k o número de nodos que irão recebe-lá. A escolha desses é baseada na seleção dos k vizinhos do nodo solicitante. O conjunto formado pelos k nodos que receberam um par chave/valor replicado é chamado de cadeia de replicação [Koppe 2013].

Ao receber uma notificação de entrada de um novo nodo na rede, isto é, quando iniciado o protocolo de entrada, é necessário verificar se o nodo solicitante compõe a cadeia de replicação do nodo que o aceitou. Se essa afirmativa for correta, esse novo nodo receberá uma réplica dos valores armazenados pelo nodo que dividiu seu conjunto de chaves. De modo análogo, na saída de um nodo da rede, é designado um participante para armazenar as réplicas deixadas pelo nodo. Aquele que irá armazenar essas réplicas poderá, no mesmo instante, responder por elas.

3.7 Considerações

Nesse Capítulo foi possível identificar as principais características das redes P2P, de modo a explorar seu surgimento como sendo uma alternativa ao modelo cliente/servidor. Observou-se também a aplicabilidade das redes P2P no compartilhamento de arquivos, destacando sistemas como Napster, GNutella e o atual BitTorrent. Além disso, foram apresentados os conceitos

fundamentais relacionados as redes P2P baseadas em DHTs, apresentando exemplos de DHTs de único e múltiplos saltos. O principal motivo da apresentação desses modelos é a exposição de diferentes abordagens encontradas na literatura.

De acordo com as características das DHTs de salto único, principalmente a eficiência na localização de uma informação na rede, o desenvolvimento desse trabalho baseia-se na elaboração de uma DHT de salto único, denominada VCubeDHT, baseada no HyperDHT. A diferença entre ambos está no algoritmo de diagnóstico utilizado, bem como a técnica de replicação de dados abordada. Diferentes técnicas de replicação de dados serão abordadas no próximo Capítulo, de modo auxiliar na escolha daquela que mais se adéqua às necessidades deste trabalho.

Capítulo 4

Replicação de Dados

O presente Capítulo tem como objetivo inicial apresentar as características da replicação de dados, bem como os motivos para sua utilização. A Seção 4.2 exibe as características sobre o modelo geral de replicação de dados. Na Seção 4.3 é discutido sobre a necessidade de adicionar novos gerenciadores de réplicas caso um determinado gerenciador tenha se tornado falho. Nas Seções 4.4 e 4.5 são apresentadas duas técnicas, denominadas replicação passiva e ativa, respectivamente. A Seção 4.6 apresenta questões relacionadas ao gerenciamento de réplicas, ou seja, a decisão de quais são os melhores gerenciadores para receber determinadas réplicas. Por fim, a Seção 4.7 exibe abordagens utilizadas nos métodos de propagação de dados, bem como a escolha daquele que mais se adéqua as características e necessidades desse trabalho.

4.1 Introdução

Geralmente os dados são replicados a fim de garantir confiabilidade e melhora no desempenho em um sistema distribuído. Um grande desafio ainda muito discutido está relacionado a capacidade de manter as réplicas consistentes. A consistência é garantida no momento em que, após a atualização de uma determinada réplica, assegura-se que as demais réplicas apresentem as mesmas características em relação àquela atualizada. De acordo com [Coulouris, Dollimore e Kindberg 2007], as técnicas de replicação de dados buscam melhorar os serviços, impactando no desempenho, na disponibilidade ou torná-los tolerante a falhas.

O servidor de nomes DNS (*Domain Name Server*) [Mockapetris 2016] é um ótimo exemplo para apresentar conceitos sobre a melhoria de desempenho e disponibilidade dos serviços com a utilização de replicação de dados. Para qualquer serviço que armazene um banco de

dados muito extenso e com um grande número de usuários como o DNS, não seria vantajoso a organização de todas essas informações em um único servidor, pois este certamente seria um ponto de falha crítico. Nesse tipo de sistema, a carga de trabalho é compartilhada entre vários servidores de nomes hierárquicos através da vinculação de seus endereços IPs aos nomes. Uma consulta de DNS para um determinado endereço qualquer resulta no retorno de endereços IPs do destinatário ou no endereço de um servidor DNS intermediário.

A fim de fornecer a disponibilidade dos recursos fornecidos por um determinado serviço, dois fatores são relevantes para a replicação de dados: falhas de servidor ou operação desconectada [Coulouris, Dollimore e Kindberg 2007]. Se por algum motivo um servidor cujo arquivo solicitado falhe, um dos servidores alternativos tende a ser responsável por atender as solicitações do cliente. Portanto, a disponibilidade dos recursos é garantida.

A operação desconectada está relacionada a desconexão dos usuários devido a algum motivo. Tomando como exemplo o uso de um *smartphone* que usa uma aplicação de agenda compartilhada, ao desconectar-se da rede, os dados são armazenados neste dispositivo. Ao realizar alterações durante o período que não se está conectado, outra pessoa poderá ter efetuado mudanças sobre o mesmo campo. Assim, há uma grande chance de ocorrer inconsistências de dados. Portanto, cabe a aplicação escolher os melhores métodos para a correção desses problemas.

A consistência de dados, bem como a transparência de replicação são considerados requisitos gerais para a replicação de dados [Coulouris, Dollimore e Kindberg 2007]. A consistência de dados trata da apresentação correta dos dados ao cliente após uma sequência de operações terem sido realizadas sobre esses. A transparência de replicação defende que os clientes não devem saber sobre a existência das réplicas dos dados.

4.2 Modelo de Sistema

O modelo geral de um sistema de replicação de dados utiliza os chamados gerenciadores de réplicas para armazenar um conjunto de réplicas distintas. Esse modelo pode ser aplicado em um ambiente baseado na arquitetura cliente/servidor na qual um cliente realiza uma sequência de operações (leitura ou atualização dos dados) a uma estrutura intermediária (*front-end*). Essa, por sua vez, irá se comunicar através da troca de mensagem com os gerenciadores de réplicas. A Figura 4.1 apresenta o modelo de arquitetura para o gerenciamento de dados replicados.

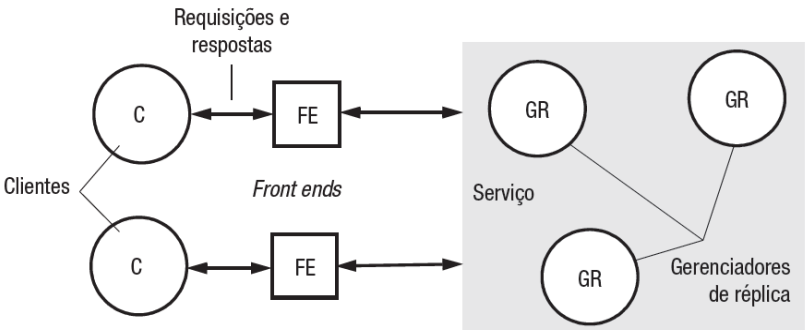


Figura 4.1: Modelo de arquitetura para o gerenciamento de dados replicados
Fonte: [Coulouris et al. 2013]

Réplicas de diferentes objetos podem estar distribuídas sobre vários gerenciadores de réplicas de modo a manter disponibilidade dos dados. O gerenciador de réplicas tem como objetivo executar operações diretamente sobre as réplicas e armazena-las de forma consistente. Por isso, muitas vezes um gerenciador de réplicas é visto como uma máquina de estados [Lamport 1978, Schneider 1990]. Nesse modelo de gerenciador, as operações são aplicadas sobre as réplicas de forma atômica, ou seja, operações que envolvem atualização de dados são não conflitantes e seguem uma sequência bem definida.

Baseada nessas operações de requisição, foram definidas por [Wiesmann et al. 2000], cinco fases que afetam-nas diretamente em relação ao seu desempenho:

- Requisição: está relacionada à estrutura intermediária (*front-end*), que pode emitir requisições para um ou mais gerenciadores de réplicas. No primeiro caso, o *front-end* se comunica com um gerenciador e este se comunica com os demais gerenciadores;

- Coordenação: para atingir mais de um gerenciador, o *front-end* emite um *multicast* de requisição para os demais gerenciadores. A Coordenação define o modo com que os gerenciadores fazem sua coordenação de forma a executar as requisições de modo consistente. Um exemplo de coordenação é baseada na ordem FIFO (*First-In, First-Out*), na qual a requisição é processada conforme sua ordem de chegada;

- Execução: descreve o modo no qual os gerenciadores executam uma requisição;

- Acordo: define a concordância coletiva dos gerenciadores sobre efeito de uma requisição;

- Resposta: está relacionada as respostas enviadas por meio dos gerenciadores. Elas podem ser dadas por um único gerenciador ou por um conjunto desses. Se a segunda for válida, o *front-end* determinará um método para selecionar uma única resposta e envia-lá ao cliente;

4.3 Serviços Tolerantes a Falhas

Na replicação de dados, o conjunto de gerenciadores de réplicas pode ser dinâmico, isto é, novos gerenciadores de réplicas podem ser adicionados ao sistema para fornecer algum tipo de suporte. Um exemplo é a adição de um novo gerenciador após algum outro ter se tornado falho. Portanto, a comunicação entre os gerenciadores se torna um ponto importante.

A ideia de grupos entre os processos também pode ser aplicada aos gerenciadores de réplicas, no qual cada gerenciador pertencente a um grupo poderá ser responsável por armazenar uma determinada réplica. Serviços que dispõem da manutenção do sistema a medida em que os processos entram e saem são considerados sistemas tolerante a falhas, discutido na Seção 2.2. Essa ideia também pode ser aplicada aos serviços baseados em replicação de dados.

Um serviço baseado em replicação de dados é considerado correto se responde corretamente a presença de falhas e se os clientes não identificam a diferença entre um serviço obtido no qual os dados são replicados, daquele fornecido por um único gerenciador de réplica correto [Coulouris, Dollimore e Kindberg 2007]. Porém, sabe-se que um serviço pode apresentar anomalias se não for gerenciado corretamente. Nas Seções 4.4 e 4.5 são feitas análises sobre os modelos de replicação de dados tolerantes a falhas existentes, para que, posteriormente, seja feita a escolha daquele que mais se enquadra no desenvolvimento deste trabalho.

4.4 Replicação Passiva

No modelo denominado replicação passiva (*primary backup*), além da presença de um único gerenciador de réplicas denominado gerenciador primário, há também um (ou vários) gerenciadores de réplicas secundários. A Figura 4.2 representa o modelo baseado em replicação passiva.

Neste modelo a comunicação entre as estruturas intermediárias (*front-ends*) com os gerenciadores ocorre de modo divergente se comparado ao modelo geral, apresentado na Seção 4.2. O *front-end* se comunicará apenas com o gerenciador denominado primário. Este, por sua vez, irá executar as operações e enviar as cópias dos dados atualizados aos gerenciadores secundários.

No momento em que o gerenciador primário tornar-se falho, um gerenciador secundário será promovido como primário.

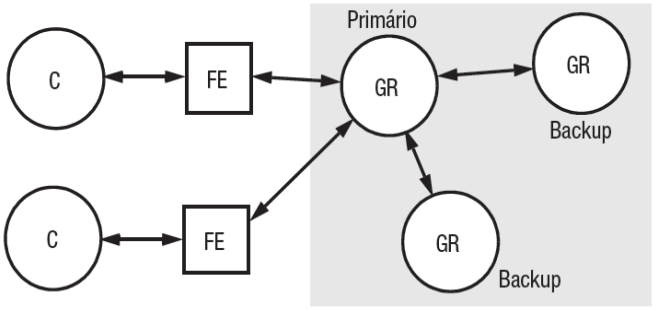


Figura 4.2: Modelo de replicação passiva (*primary backup*)
Fonte: [Coulouris et al. 2013]

A sequência de eventos é iniciada pelo cliente, no qual enviará a solicitação da operação ao *front-end* e seguirá as demais etapas [Coulouris et al. 2013]:

- Requisição: o *front-end* emite uma requisição ao gerenciador primário contendo um identificador exclusivo da operação desejada;
- Coordenação: o gerenciador primário processa cada requisição baseada na ordem de chegada, tratando-as atomicamente. Inicialmente ele verifica se a operação já foi executada, de modo a analisar o identificador recebido. Caso já tenha recebido uma operação com o mesmo identificador, ele apenas reenviará a resposta;
- Execução: se na etapa anterior for reconhecido que é uma nova operação a ser executada, o gerenciador as executa e armazena a resposta;
- Acordo: é verificado se a operação recebida foi apenas requisição de leitura, com isso o gerenciador primário enviará os dados ao *front-end*. Este, por sua vez irá repassá-los ao cliente. Porém, se a operação for de atualização, o gerenciador primário atualizará as informações e enviará os dados atualizados aos gerenciadores secundários. Para confirmação, esses devem responder através de mensagens de confirmação;
- Resposta: o gerenciador primário responderá ao *front-end*. Este, por sua vez, enviará a resposta ao cliente;

Esse modelo de replicação requer que a comunicação entre os gerenciadores de réplicas (primário e secundários) sejam feitas de forma síncrona, de modo a manter a consistência das informações [Coulouris et al. 2013]. Caso o gerenciador primário tornar-se falho, o sistema deverá manter atualizadas as operações feitas até o momento e continuar as operações a partir da última ação executada. Para isso, todos os gerenciadores secundários mantém informações referentes as operações realizadas, bem como aquelas relacionadas aos demais gerenciadores. Eles devem concordar a respeito das operações feitas pelos clientes até o momento da falha do gerenciador primário. Baseando-se hipótese que o gerenciador primário tornou-se falho, o *front-end* reenviará a solicitação ao novo gerenciador primário. Este, por sua vez, não precisará saber em qual etapa do processo de requisição o antigo gerenciador primário estava, ele apenas irá realizar a etapa de coordenação normalmente.

Para funcionar corretamente até *n* falhas de processo, o sistema de replicação passiva necessita de *n+1* gerenciadores de réplicas. Uma desvantagem desse sistema está relacionado ao número de rodadas necessárias (via *multicast*) que os gerenciadores secundários devem efetuar para manter um estado consistente entre as réplicas [Coulouris et al. 2013].

4.5 Replicação Ativa

O modelo caracterizado como replicação ativa funciona de modo que os gerenciadores de réplicas são máquinas de estado, organizados em grupos, e, diferentemente do modelo anterior, não há um gerenciador de réplicas primário. A Figura 4.3 representa esse tipo de modelo.

As requisições emitidas pelo *front-end* são entregues a um grupo de gerenciadores, que processam-nas de forma idêntica. Geralmente, a falha de um gerenciador não gera nenhum impacto no funcionamento no serviço, pois os demais gerenciadores tendem a responder a futuras solicitações.

Na replicação ativa, a sequência de eventos é iniciada pelo cliente no qual enviará a solicitação da operação ao *front-end* e seguirá nas seguintes etapas [Coulouris et al. 2013]:

- Requisição: assim como na replicação passiva, o *front-end* associará um identificador à requisição e a enviará para um grupo de gerenciadores de réplicas (via *multicast* ordenado), de forma que todos a recebam-na;

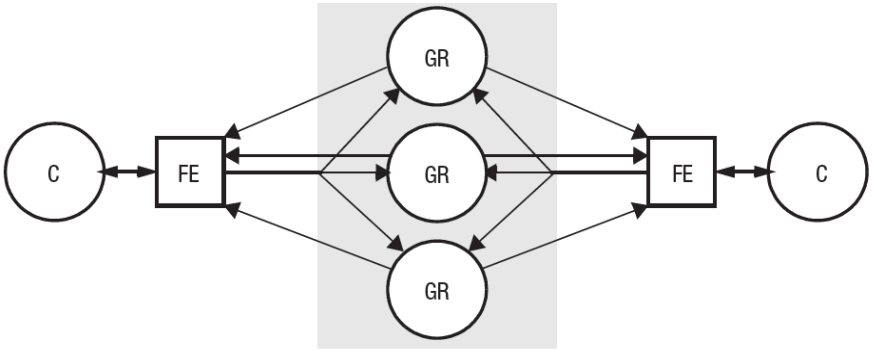


Figura 4.3: Modelo de replicação ativa
Fonte: [Coulouris et al. 2013]

- Coordenação: o sistema de comunicação em grupo envia a requisição a cada gerenciador de réplica correto na mesma ordem total, ou seja, cada requisição é processada sequencialmente na mesma ordem em cada réplica;
- Execução: os gerenciadores de réplicas que receberam as requisições executam-nas. Assim como na primeira etapa, a resposta possuirá o identificador exclusivo do cliente;
- Acordo: essa fase não existirá nesse modelo de replicação, visto que é adotado à distribuição via *multicast*;
- Resposta: os gerenciadores de réplicas enviam suas respostas para o *front-end*. Este, por sua vez, decidirá (de acordo com o algoritmo implementado) a forma com que irá repassar a resposta ao cliente. Um exemplo de tomada de decisão pode ser aquela em que o *front-end* repassará a primeira resposta recebida;

Nesse tipo de modelo é necessário que os gerenciadores mantenham a consistência sequencial, ou seja, todas as operações devem ser executadas na mesma ordem. Para o processamento *multi-threads* essa abordagem muitas vezes pode se tornar um problema [Coulouris et al. 2013].

4.6 Gerenciamento de Réplica

Nas seções anteriores foi possível observar o armazenamento das réplicas por meio de gerenciadores de réplicas (primários ou secundários). No entanto, uma questão básica em um sistema distribuído que suporta replicação de dados, é a decisão de onde, quando e quais

serão os gerenciadores que estarão melhores localizados para o armazenamento dos dados [Tanenbaum e Steen 2007]. A questão relacionada ao posicionamento abrange duas vertentes, sendo elas: o posicionamento dos gerenciadores e o posicionamento dos dados.

O posicionamento dos gerenciadores refere-se em encontrar a melhor localização na rede para posicioná-los. Já o posicionamento dos dados está relacionado a escolha do melhor gerenciador para o armazenamento desses.

Diversos fatores podem influenciar diretamente o cálculo para escolha do melhores gerenciadores de réplicas, podendo eles não estarem relacionados apenas a distância, mas por exemplo a largura de banda disponível na comunicação entre os clientes e os gerenciadores de réplicas. Geralmente, a medida utilizada para realizar esse cálculo é a distância.

Usualmente, as implementações se baseiam na escolha do gerenciador cujas médias das distâncias entre ele e os clientes são mínimas. No entanto, como visto em [Tanenbaum e Steen 2007], há trabalhos que propõe considerar apenas a topologia de rede.

Há uma variedade de métodos que buscam identificar uma região para o posicionamento de réplicas. Uma que pode ser destacada é a vista em [Szymaniak, Pierre e Steen 2005], que define um conjunto de nodos posicionados em um espaço m-dimensional acessando o mesmo conteúdo. A ideia geral do algoritmo é identificar os k maiores *clusters* e designar apenas um nodo de cada *cluster* para armazenar o conteúdo replicado. Para esse processo de identificação, o espaço inteiro é particionado em células. O processo de escolha das células é baseado na densidade das mesmas, onde as mais densas são escolhidas para posicionar um gerenciador de réplicas. Uma célula densa é aquela que possui um grande número de nodos vizinhos.

Tendo definido a melhor posição ocupada por um gerenciador de réplicas, outro assunto bastante discutido na literatura está relacionado a determinação do posicionamento do conteúdo. Há três métodos para este fim, mais conhecidos como: réplicas permanentes, réplicas iniciadas por servidor e réplicas iniciadas pelo cliente [Tanenbaum e Steen 2007].

No primeiro método, o número de réplicas permanentes armazenam um depósito de dados distribuído, sendo que, em geral, o número de réplicas é relativamente pequeno. Um exemplo de uso desse método é encontrado na distribuição de páginas Web que utiliza, por exemplo, um conceito denominado espelhamento. Este, consiste em copiar uma determinada página para um conjunto limitado de servidores que, após o acesso a essa página por um cliente, é determinado

qual será o servidor espelhado que irá atende-lo. Um outro exemplo da utilização desse conceito é na replicação de bancos de dados distribuídos sobre um conjunto de servidores.

Já no segundo método, os proprietários dos dados criam réplicas visando, diferentemente do método anterior, o aprimoramento no desempenho. Esse aprimoramento pode ser feito, por exemplo, com o objetivo de reduzir a carga de um servidor que está constantemente sendo requisitado por um grande número de clientes. Desse modo, armazenar temporariamente as réplicas dos dados mais solicitados em um servidor de réplicas que se encontra mais próximo as solicitações seria uma solução interessante. Nesse caso, as solicitações que antes eram feitas ao servidor passado, serão redirecionadas aquele servidor mais próximo do cliente.

Como visto no segundo método, as iniciativas da criação de réplicas eram feitas pelos proprietários dos dados. No entanto, no terceiro e último método essa iniciativa é tomada pelos clientes. O funcionamento desse método pode ser assimilado ao funcionamento da cache. Sabe-se que as caches são utilizadas para melhorar o tempo no acesso aos dados. Portanto, quando um cliente deseja acessar um determinado dado, ele irá busca-lo em um servidor de réplicas mais próximo. Esse, por sua vez, pode estar localizado na máquina do cliente ou até mesmo em uma máquina separada que esteja participando da mesma rede local do cliente.

Geralmente os dados são mantidos na cache do cliente por um limitado período de tempo. No entanto, assim que os dados são modificados o cliente deve fazer uma nova requisição ao servidor de réplicas principal e, posteriormente, atualizar os dados modificados ou até mesmo realizar a inserção de novos dados na cache [Tanenbaum e Steen 2007].

4.7 Propagação de Dados

A replicação de dados em um sistema distribuído é de suma importância, bem como algumas possíveis formas de realizar o gerenciamento de réplicas. No entanto, além desses aspectos, é necessário decidir o modo no qual o conteúdo será distribuído entre os gerenciadores de réplicas. Esta seção apresenta algumas ideias de distribuição de conteúdo encontradas na literatura.

Ao tratar-se de distribuição de dados entre gerenciadores de réplicas, um ponto importante, de acordo com [Tanenbaum e Steen 2007], refere-se aquilo que deve ser propagado entre eles. Através disso, há três possibilidades que são levadas em consideração: (i) propagar somente uma notificação de uma atualização, (ii) propagar a operação de atualização para outras cópias,

(iii) transferir dados de uma cópia para outra.

A primeira, conhecida como protocolo de invalidação, tem como objetivo propagar apenas uma notificação de atualização aos demais gerenciadores de réplicas. Através disso, ao receberem essa notificação, os gerenciadores assumem que houve uma atualização, e, através da notificação é possível descobrir quais dados não estão consistentes em relação ao original. Após a solicitação do cliente por uma réplica inválida, há a necessidade da atualização dessa para que, posteriormente, possa ser repassada ao cliente. A principal vantagem da utilização desse método é o uso de pouca largura de banda, visto que a única informação a ser transferida é a notificação dos dados desatualizados [Tanenbaum e Steen 2007].

A segunda abordagem também busca realizar apenas a transferência de notificações, porém, além de informar a necessidade de atualização, noticia também, através dos parâmetros, qual operação cada gerenciador de réplica deve realizar para manter as réplicas consistentes.

Já a terceira abordagem, diferente das demais, busca realizar a transferência dos dados entre os gerenciadores de réplicas. Essa abordagem foi a escolhida para ser utilizada no desenvolvimento deste trabalho, pois em sistemas de arquivos P2P, a replicação de dados busca principalmente a aceleração de requisições de busca e consulta, bem como o balanceamento de carga entre os nodos. Nesse tipo de sistema, praticamente todos os arquivos são somente leitura e as atualizações são tratadas apenas como inserção ou remoção de arquivos ao sistema.

Além de ser necessário realizar buscas e consultas de modo eficiente, um outro requisito que deve ser atendido é a aplicação de um método ágil na transferência de arquivos entre os gerenciadores de réplicas. Desse modo, nas subseções a seguir serão apresentadas diferentes abordagens que permitem a propagação desses dados.

4.7.1 Replicação baseada no Protocolo BitTorrent

A primeira abordagem utilizada na propagação de dados é aquela presente em um protocolo bastante conhecido para distribuição de arquivos, denominado BitTorrent [Zhang et al. 2011]. Antes de apresentar a técnica usada pelo protocolo, deve-se explicar seu funcionamento.

O funcionamento desse protocolo se baseia na cooperação entre pares para distribuição de um arquivo na internet. Esse conjunto de pares é descrito como *torrent* [Kurose e Ross 2012]. A principal característica de projeto no protocolo é o fracionamento do arquivo em partes menores,

bem como sua disponibilidade entre os pares sobre uma rede P2P [Coulouris et al. 2013].

Após a inserção de um novo par na rede, a quantidade de blocos de dados armazenados por ele será nulo. A medida na qual o tempo passa, ele acumulará blocos através do *download* dos mesmos. Esse procedimento funciona de modo que o rastreador selecione um subconjunto de pares e, posteriormente, envie o endereço de cada par do subconjunto a este novo ingressante. Esse, por sua vez, irá tentar estabelecer conexões TCP com cada par pertencente ao conjunto, de modo a solicitar seus blocos de dados. Os *n* pares que obtiveram sucesso no estabelecimento da conexão com o novo ingressante serão denominados vizinhos do mesmo. Desse modo, o novo par receberá como resposta *n* listas de blocos de dados.

Assim como a solicitação por blocos de dados feitas no *download*, o par ingressante também poderá responder as solicitações de modo a disponibilizar seu conjunto de dados a seus vizinhos, através do *upload* dos mesmos.

Um rastreador no protocolo BitTorrent é um servidor que está limitado apenas ao armazenamento de informações referente aos *downloads* em andamento, bem como no auxilio aos pares a fim de encontrarem os demais participantes da rede. Todas as demais questões são tratadas na interação dos pares, não havendo um coordenador central [Cohen 2003].

Numa comunicação de um par com seus vizinhos, é possível identificar quais blocos de dados seus vizinhos possuem e também quais ele necessita para completar parte de um arquivo. Dessa forma, duas decisões são bastantes importantes a serem tomadas pelo par em questão. A primeira se refere a decisão sobre quais blocos de dados solicitar inicialmente. Já a segunda deve ser a escolha dos pares nos quais deverá realizar tal solicitação.

A técnica utilizada pelo protocolo BitTorrent para determinar os blocos a serem solicitados é denominada *rarest-first* (o mais raro primeiro). A principal característica dessa técnica é identificar quais são os blocos mais raros armazenados por seus vizinhos. Um bloco raro é aquele com menor número de cópias. Desse modo, os blocos mais raros são disseminados, aumentando seu número de cópias na rede [Kurose e Ross 2012].

Assim como na decisão sobre quais blocos serão solicitados, o protocolo utiliza uma técnica a fim de determinar quais solicitações serão priorizadas. A principal característica dessa técnica é priorizar os vizinhos que possuem uma maior taxa de transmissão de dados. Desse modo, cada par mede sua taxa de recebimento de bits e determina, por padrão do protocolo, quatro pares

cujo critério anterior seja atendido [Kurose e Ross 2012]. Além disso, a cada trinta segundos, o par emissor deverá escolher um outro par aleatório para se tornar um novo vizinho, de modo a enviar seus blocos de dados a ele.

Numa transmissão de dados, cada participante sempre procurará aquele vizinho que possuir melhor taxa de transmissão compatível. Se os dois pares se mostrarem satisfeitos com a troca de dados, eles a continuarão até encontrarem um parceiro melhor. A técnica comentada é interessante por identificar pares com taxas de *upload* compatíveis.

4.7.2 Replicação baseada no Caminho de Busca

A segunda abordagem é denominada Caminho de Busca [Yamamoto, Maruta e Oie 2005], bastante discutida na literatura devido a sua utilização em vários serviços de armazenamento de arquivos, como OceanStore [Kubiatowicz et al. 2000] e Freenet [Clarke et al. 2001].

Esse método é aplicado após um procedimento de busca ter sido concluído corretamente. As réplicas serão armazenadas em todos os nodos que participaram da busca. O procedimento inicia-se com a propagação dos dados no caminho inverso percorrido durante o processo de busca, até chegar ao nodo que a originou.

Há uma variante desse método, denominada Caminho de Busca com Probabilidade [Yamamoto, Maruta e Oie 2005]. O procedimento realizado para distribuição das réplicas é o mesmo do anterior. No entanto, neste, cada nodo possuirá uma probabilidade para recebimento da réplica. Desse modo, as réplicas não serão atribuídas a todos os nodos do caminho de busca.

De acordo com [Yamamoto, Maruta e Oie 2005], essa variante foi desenvolvida baseada em dois fatores. O primeiro está relacionado ao número de réplicas criadas, sendo que o número dessas pode ser mais que o necessário para atingir o desempenho da busca. O segundo destaca o desperdício em relação a capacidade de processamento e de armazenamento dos nodos.

Notou-se que o método caminho de busca tem por objetivo aumentar a disponibilidade dos dados, bem como facilitar a busca das réplicas por meio da localização das mesmas.

4.7.3 Replicação utilizada no sistema PAST

Assim como a anterior, essa abordagem de replicação de dados também é encontrada em sistemas de armazenamento de arquivos, como o PAST [Rowstron e Druschel 2001].

O PAST é um serviço P2P de armazenamento de arquivos (imutáveis) que objetiva, através do uso de técnicas de replicação de dados e *caching*, prover alta disponibilidade dos dados, bem como escalabilidade e segurança dos mesmos. Esse serviço é composto por nodos que estão conectados via internet onde cada um é capaz de atender as solicitações realizadas por clientes. As solicitações a serem atendidas podem ser tanto de inserção como remoção de arquivos.

Diferentemente da abordagem utilizada pelo BitTorrent, no sistema PAST o número de réplicas k é definido no momento da inserção do arquivo por meio do cliente. Além disso, outra diferença é que nesse, as k réplicas serão transferidas por completo nos k nodos mais próximos.

A técnica de *caching* comentada inicialmente faz com que haja mais de k cópias presentes no sistema. Ela é utilizada afim de reduzir a latência de buscas das réplicas, de modo a armazenar, temporariamente, um determinado número de réplicas mais próximo do cliente.

4.7.4 Replicação em Banco de Dados Distribuídos

Um banco de dados distribuídos é definido como um agrupamento de múltiplos bancos de dados que estão logicamente inter-relacionados e distribuídos por uma rede de computadores [Elmasri e Navathe 2010]. A replicação de dados também pode ser aplicada a esse tipo de sistema, de forma a garantir melhorias da disponibilidade dos dados. Caso seja diagnosticado com falha um determinado site que armazena uma tabela de dados replicada, a disponibilidade é garantida pois a mesma tabela poderá ser encontrada em outro site que armazena essa réplica.

A replicação em um banco de dados distribuído pode ser feita de algumas formas. As duas abordagens discutidas nesse trabalho serão: a replicação total e replicação parcial de dados. O primeiro método consiste em replicar todo o banco de dados em todos os sites do sistema distribuído. Além de aumentar a disponibilidade dos dados, essa abordagem melhora o desempenho para consultas globais. No entanto, a velocidade das operações de atualizações de dados pode ser afetada. Já na segunda, apenas alguns fragmentos são replicados. Essa abordagem é conhecida como replicação parcial de dados.

Uma técnica que também pode ser aplicada juntamente a replicação é a fragmentação de dados [Elmasri e Navathe 2010]. A fragmentação de dados nesse contexto consiste em dividir uma tabela em vários fragmentos menores para distribuição desses. Há dois métodos conhecidos para fragmentação, são eles: fragmentação horizontal e vertical [Elmasri e Navathe 2010].

A fragmentação horizontal consiste na divisão horizontal da relação, de modo a separar tuplas em dois ou mais fragmentos. A ideia pode ser comparada a recuperação de tuplas utilizando uma clausula *where* para seleção de dados em um comando SQL (*Structured Query Language*). A Tabela 4.1 representa o conjunto de dados original na qual serão aplicadas as fragmentações.

Tabela 4.1: Tabela pessoa

id	nome	cidade	cpf
1	José	São Paulo	000.111.222-01
2	Eduardo	São Paulo	111.000.333-01
3	Carlos	Curitiba	222.333.111-02
4	Maria	Curitiba	123.321.111-03
5	Joseph	Curitiba	100.001.212-04

A fragmentação horizontal da tabela foi aplicada em relação a coluna cidade. Desse modo, obtém-se como resultado as Tabelas 4.2 e 4.3. É possível notar a separação, bem como a união das tuplas que possuem o mesmo valor da coluna cidade. A Tabela 4.2 representa as tuplas cujo valor da cidade corresponde a São Paulo e a Tabela 4.3 representa aquelas que possuem Curitiba como valor da cidade. Desse modo, as tabelas estarão localizadas em locais diferentes, de modo que a busca e a replicação das mesmas seja feita de modo mais eficiente.

Tabela 4.2: Aplicação da fragmentação horizontal para coluna cuja cidade é São Paulo

id	nome	cidade	cpf
1	José	São Paulo	000.111.222-01
2	Eduardo	São Paulo	111.000.333-01

Tabela 4.3: Aplicação da fragmentação horizontal para coluna cuja cidade é Curitiba

id	nome	cidade	cpf
3	Carlos	Curitiba	222.333.111-02
4	Maria	Curitiba	123.321.111-03
5	Joseph	Curitiba	100.001.212-04

Diferente da anterior, a fragmentação vertical divide a tabela verticalmente, decompondo suas colunas. Esse método é feito de modo a possibilitar a reconstrução da tabela a partir dos fragmentos. Para isso, ao fragmentar verticalmente uma tabela, será inserido uma nova coluna como chave primária, para posteriormente relacionar os dados que foram particionados.

A aplicação da fragmentação vertical sobre a Tabela 4.1 resultou em duas outras (4.4 e 4.5). Na Tabela 4.4 é possível notar apenas a presença de duas colunas, sendo elas a do identificador e a do nome. Já na Tabela 4.5 há a presença de três colunas, isso por que a fragmentação vertical foi aplicada de modo a separar em uma tabela apenas os valores de nome e em outra tabela as instâncias dos atributos cidade e cpf. Nota-se a presença da primeira coluna *id* em ambas as tabelas. Ela serve para associar as tabelas nas quais foram aplicadas a fragmentação vertical.

Tabela 4.4: Aplicação da fragmentação vertical para coluna nome

id	nome
1	José
2	Eduardo
3	Carlos
4	Maria
5	Joseph

Tabela 4.5: Aplicação da fragmentação vertical para as coluna cidade e cpf

id	cidade	cpf
1	São Paulo	000.111.222-01
2	São Paulo	111.000.333-01
3	Curitiba	222.333.111-02
4	Curitiba	123.321.111-03
5	Curitiba	100.001.212-04

4.7.5 Replicação utilizada no RAID 5

O RAID (*Redundant Array of Independent Disks*) [Patterson, Gibson e Katz 1988] é uma solução que visa combinar vários discos rígidos físicos, formando uma única unidade lógica de armazenamento para obter desempenho e segurança sobre um conjunto de dados armazenados.

Há diferentes variantes de RAID, cada um com suas técnicas para manipulação dos dados. A abordagem utilizada no RAID 5 se destaca por utilizar técnicas que permitem espalhar tanto os dados como também a paridade entre todos os $N + 1$ discos.

De modo geral, um bit de paridade é adicionado para evitar erros na transmissão de dados. Esse método diverge daqueles que armazenam dados em N discos e a paridade somente em um. Além disso, a fragmentação dos dados entre os discos é feita a nível de *byte*.

A partir da divisão dos dados e da paridade comentada anteriormente, o processo de busca do primeiro é iniciado com a junção das informações de paridade distribuída aos discos com os dados. Nesse processo um disco pode tornar-se falho. Porém, devido a replicação de dados, um disco com a informação necessária poderá se encontrar disponível. Portanto, a busca é feita sobre esse disco, de modo a evitar a perda de dados e mantendo a disponibilidade dos mesmos.

4.8 Considerações

Nesse Capítulo foi possível identificar que a replicação de dados visa garantir a disponibilidade dos dados, tornando os sistemas que o implementam confiáveis e consequentemente aumentando seu desempenho. Além disso, foi possível entender o funcionamento do modelo geral de replicação de dados, bem como as técnicas de replicação ativa e passiva. Ainda sobre essas técnicas, foi possível identificar a dificuldade na garantia da consistência dos dados com a utilização do processamento *multi-thread*. Por fim, foi destacada a importância da propagação de dados na replicação, bem como a apresentação das técnicas para esse fim.

Os métodos no protocolo BitTorrent, em bancos de dados distribuídos e também no RAID 5 podem ser utilizados nesse trabalho, isto é, aproveitando suas características para a implementação de uma nova metodologia de replicação.

A principal motivação para apresentação desses métodos de propagação de dados é a exposição de diferentes abordagens para realizar a propagação de dados de modo eficiente, visto que o método tradicional de propagação de um arquivo inteiro pode não ser uma solução eficiente, por exemplo, em termos de largura de banda ou latência. O Capítulo seguinte aborda as principais metodologias utilizadas para o desenvolvimento da proposta.

Capítulo 5

VCubeDHT - Uma DHT de Salto Único Baseada em um Hipercubo Virtual com Replicação

Esse Capítulo apresenta as principais características do sistema proposto, denominado VCubeDHT. Inicialmente são apresentadas as motivações para seu desenvolvimento, bem como as semelhanças entre o HyperDHT e o VCubeDHT. Na Seção 5.1 é apresentada a abordagem que realiza a identificação dos vértices vizinhos para replicação de dados. O Capítulo segue com a Seção 5.2, que expõe a metodologia de replicação utilizada neste trabalho. A Seção 5.3 explora a reorganização do sistema após o diagnóstico de participantes falhos. O Capítulo é finalizado com a Seção 5.4, expondo as características de disponibilidade do sistema.

O sistema utilizado como base para o desenvolvimento do VCubeDHT foi o HyperDHT, proposto e implementado por [Koppe 2013] e descrito no Capítulo 3 deste trabalho.

A principal motivação para o desenvolvimento deste sistema foi a implementação de uma estratégia de replicação de dados mais eficiente que a empregada pelo HyperDHT. Além disso, uma nova solução de diagnóstico foi utilizada.

O HyperDHT utiliza o algoritmo DiVHA para realizar o diagnóstico dos participantes da rede e, como modelo de replicação utiliza uma abordagem de replicação ativa, na qual os *peers* replicam seus dados por completo entre seus k vizinhos no hipercubo. Já o VCubeDHT, para efetuar o diagnóstico dos participantes, utiliza o VCube, proposto e implementado por [Ruoso 2013]. O VCube foi utilizado devido a suas vantagens em relação ao algoritmo DiVHA, descritas no Capítulo 2.

As principais estratégias utilizadas no HyperDHT, como mecanismos para posicionamento

e busca de objetos na rede, particionamento e distribuição consistente das chaves entre os *peers* e também protocolos de entrada e saída, foram mantidas na implementação do VCubeDHT. Por fim, para garantir a confiabilidade, disponibilidade e tolerância a falhas em relação aos arquivos armazenados na rede, foi implementado no sistema proposto uma abordagem de replicação ativa, explorando uma alternativa na propagação das réplicas entre os participantes do sistema.

5.1 Determinação dos Vizinhos

A fim de determinar os vizinhos que irão receber as réplicas de um determinado participante, a presente proposta utiliza a mesma estratégia de cálculo adotada pelo HyperDHT. Cada participante do sistema realiza um cálculo baseado em três parâmetros, sendo eles, um identificador (*i*) do vértice no qual deseja-se verificar seus vizinhos, a dimensão (*d*) na qual determina que os vértices a serem testados são aqueles pertencentes ao *cluster d*, bem como um contador (z) que representa a distância mínima entre dois vértices.

O cálculo é feito com o uso da operação de $\oplus$ (*ou-exclusivo*) entre o identificador *i* e o contador *z*, identificando os vértices mais próximos daquele representado pelo identificador (*i*). A função utilizada é definida em [Koppe 2013] e representada por:

$$C_{i,d} = \ i \ \oplus z, \ \forall z \in \mathbb{Z}_{\geqslant 0} \ : \ z \leqslant 2^d - 1.$$

Na Tabela 5.1 são apresentados os resultados após o término da execução da função $C_{i,d}$ realizada por cada *peer* em um sistema de dimensão $d = 3$. Nela é possível observar, por exemplo, que o método identifica o primeiro vértice mais próximo ao vértice 0 sendo o vértice 1, o segundo mais próximo sendo o vértice 2, e o terceiro, sendo o vértice 3, e assim sucessivamente. No entanto, se o vértice analisado não estiver ocupado, ou seja, não possuir nenhum participante que o compõe, ele não poderá fazer parte do grupo de replicação, fazendo com que o próximo da lista seja analisado. A Figura 5.1 exibe a identificação do cenário comentado.

Tabela 5.1: Identificação dos vértices vizinhos em um sistema de dimensão d = 3

d	$c_{0,d}$	$c_{1,d}$	$c_{2,d}$	$c_{3,d}$	$c_{4,d}$	$c_{5,d}$	$c_{6,d}$	$c_{7,d}$
1	1	0	3	2	5	4	7	6
2	2, 3	3, 2	0, 1	1, 0	6, 7	7, 6	4, 5	5, 4
3	4, 5, 6, 7	5, 4, 7, 6	6, 7, 4, 5	7, 6, 5, 4	0, 1, 2, 3	1, 0, 3, 2	2, 3, 0, 1	3, 2, 1, 0

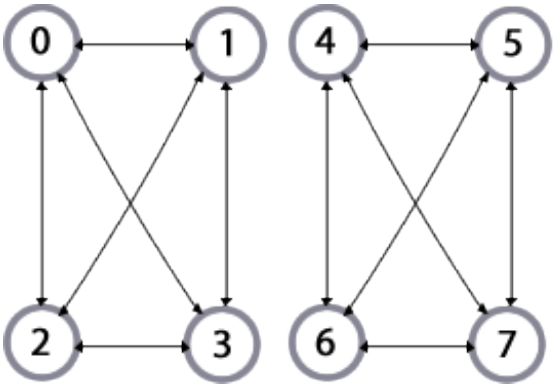


Figura 5.1: Exemplo da identificação dos vizinhos em um sistema de dimensão $d = 3$
Fonte: [O Autor 2016]

5.2 Método de Propagação das Réplicas

Para garantir a confiabilidade, melhora no desempenho e tolerância a falhas em um sistema distribuído, é necessário o uso de uma abordagem que garanta a redundância dos dados, pois sabe-se que o sistema pode apresentar algum tipo de falha. Sendo assim, o VCubeDHT apresenta uma nova proposta de propagação de dados.

A abordagem implementada segue conceitos de algoritmos como Protocolo BitTorrent e Replicação Baseada em Raid 5, ambos comentados no Capítulo 4, que utilizam abordagens de replicação de fragmentos a nível de *byte*. O pseudo-código responsável por esse processo é apresentado no Algoritmo 1. Vale ressaltar que o VCubeDHT determina que as informações armazenadas nos arquivos não são alteradas, não havendo a necessidade da implementação de procedimentos de atualização e verificação de consistências dos dados.

Na primeira linha do pseudo-código é possível observar que a função recebe como entrada três parâmetros. O primeiro, identificado como *idPeer*, representa o identificador do *peer* que iniciou o processo de replicação de seus dados. O segundo, denotado por *k*, caracteriza o fator de replicação escolhido, que pode ser ajustado de acordo com o nível de replicação desejado. Já o terceiro, identificado como *frag*, representa o fator de fragmentação que também pode ser ajustado. De acordo com [Koppe 2013], redes P2P maiores que possuem a característica de serem instáveis, isto é, com uma maior ocorrência de *churn*, dependem de um nível de replicação maior, se comparado com sistemas menores e mais estáveis.

Nas linhas 2 e 3 são inicializadas três variáveis auxiliares, definidas por *z*, *cPiecesSender* e

i, respectivamente. Nota-se que até a linha 8 (com exceção da linha 4 que é abordada posteriormente), a função *make_replicas* segue os mesmos princípios da função original implementada pelo HyperDHT. Posteriormente, caso o vértice analisado esteja ocupado (verificação feita na linha 8), o *peer* responsável é identificado como pertencente a cadeia de replicação (linha 9). A partir da linha 10, cada arquivo será fragmentado em *frag* fragmentos e, para identificação futura desses, cada bloco é composto pelo nome do arquivo original, juntamente com um índice que indica o bloco do arquivo original que ele representa. Após isso, os blocos serão enviados para os *peers* que foram identificados como pertencentes a cadeia de replicação (linha 16).

Algoritmo 1 Make Replicas

```
1:  procedure MAKE_REPLICAS(idPeer, k, frag)
2:      z, cPiecesSender = 0
3:      i = idPeer
4:      listBlocks = configureBlocks(f)
5:      while k > 0 do
6:          z++
7:          i = idPeer XOR z
8:          if vertices[i] is occupied then
9:              groupReplication[i] = true
10:             for p = 0 TO files do
11:                 numBytes = files[p].length
12:                 fragment = numBytes/f
13:                 for n = 0 TO listBlocks do
14:                     blockName = files[p] concat with _R_ and n
15:                     blocks.create(blockName, fragment)
16:                     peer_id send block to peer_i
17:                 end for
18:             end for
19:             k--
20:             cPiecesSender++
21:         end if
22:     end while
23: end procedure
```

Ainda no Algoritmo 1, mais especificamente na linha 4, é possível observar a chamada de um método denominado *configureBlocks*. Este método tem por objetivo fazer com que cada *peer*, antes de dar início ao processo de replicação, crie uma lista com a divisão dos blocos que devem ser enviados de acordo com o fator de fragmentação *frag* informado pelo parâmetro.

O método consiste na criação de uma tabela verdade de tamanho 2^{frag} e na seleção de de-

terminados valores com base nela. A Figura 5.2 apresenta a execução do método *make_replicas* feita pelo *peer* 0, cujo fator de replicação (*k*) e fragmentação (*frag*) são iguais a 3.

O processo de seleção das colunas da tabela verdade feitas por este *peer*, tem por objetivo selecionar apenas aquelas que possuem linhas da tabela verdade com *frag - 1* valores 1. Após essa identificação, o *peer* 0 armazenará em sua lista tais valores.

No exemplo é possível observar que o *peer* 0 armazena em sua lista os valores BC, AC e AB respectivamente. Dessa forma, para um fator de fragmentação previamente definido como $f = 3$, o arquivo original, representado por *R.txt*, será dividido em 3 blocos. De acordo com o algoritmo, o vizinho mais próximo do *peer* 0 irá receber os blocos B e C, o segundo mais próximo receberá os blocos A e C e por fim, o terceiro mais próximo receberá os blocos A e B. Esse procedimento é ilustrado na Figura 5.2.

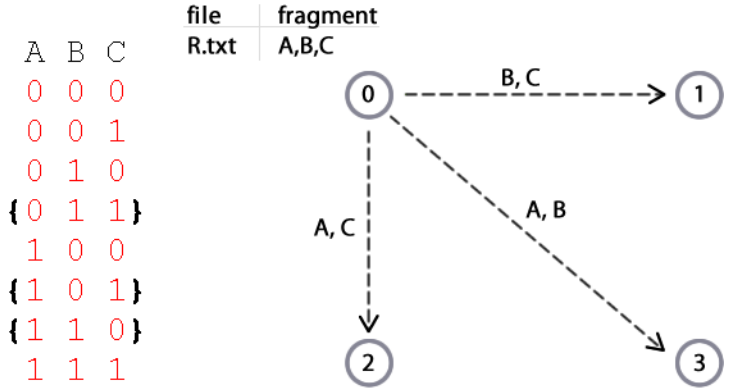


Figura 5.2: Exemplo de execução da função *make_replicas* feita pelo *peer* 0
Fonte: [O Autor 2016]

A entrada de novos *peers* na rede faz com que os demais participantes verifiquem se o *peer* solicitante faz parte da sua cadeia de replicação, de modo a replicar seus blocos de dados a ele.

5.3 Balanceamento de Réplicas

Com o intuito de realizar o diagnóstico dos participantes da rede, após um período de tempo que pode ser ajustado pela necessidade da aplicação, o VCubeDHT faz com que todos os *peers* executem o algoritmo de diagnóstico distribuído VCube.

Caso diagnosticado um *peer* falho, além de identificar o novo *peer* responsável pelo vértice deixado pelo *peer* falho, há a necessidade da redistribuição dos blocos de dados entre os

peers que possuem o *peer* falho como pertencente a seu grupo de replicação. O procedimento necessário para redistribuição dos blocos de dados é apresentado no Algoritmo 2.

Após o término do diagnóstico, cada participante da rede analisa, através de seu vetor de *timestamp* (que indica o seu conhecimento sobre o estado de cada *peer* do sistema), a ocorrência de um *peer* falho (linha 3). Nota-se que no *timestamp*, o *status* de um *peer* é definido por um inteiro *impar* caso o *peer* seja falho e par caso não-falho. Depois dessa identificação, é verificado se o *peer* que está executando o algoritmo faz parte da cadeia de replicação daquele *peer* diagnosticado como falho (linha 4). Na sequência, o mesmo *peer* replicará os blocos de dados referentes a seus arquivos ao seu novo vizinho, substituindo o *peer* falho em sua cadeia de replicação (linha 5). Após esse procedimento, é identificado o novo *peer* responsável pelo vértice em que se encontrava o *peer* falho (linha 6) e, por fim, caso o *peer* que esteja executando o algoritmo não for responsável pelo vértice deixado pelo *peer* falho, este, deverá enviar todos os blocos referentes ao *peer* falho ao novo responsável pelo vértice desocupado (linhas 7 e 8).

Algoritmo 2 Redistribution blocks

```
1:  procedure ORGANIZE_REPLICAS
2:      for each i in timestamp do
3:          if i % 2 == 1 then
4:              if peer_id is replication member of peer_i then
5:                  peer_id send your blocks to new member of group replication
6:                  respIdFail = find_vertice_owner(id peer of vertice_i)
7:                  if id != idFail then
8:                      peer_id send blocks of peer_i to peer_respIdFail
9:                  end if
10:             end if
11:         end if
12:     end for
13: end procedure
```

A Figura 5.3 tem como objetivo apresentar o processo comentado, retratando um cenário no qual o *peer* 7 é caracterizado como falho. Nela são identificados os *peers* disponíveis, através dos círculos com a cor branca, e, o indisponível, representado pelo círculo com maior destaque. Cada *peer* armazena os blocos de dados referentes a cada réplica dos arquivos que possui.

Após o diagnóstico do sistema, todos os participantes possuem conhecimento do estado dos demais, porém, os que iniciam o procedimento de reorganização do sistema são aqueles que possuíam o *peer* falho como membro de seu grupo de replicação. Desse modo, o *peer* 6 busca

um novo vizinho, substituindo o *peer* 7 e enviando a réplica de seu arquivo (em blocos) a ele. Nota-se que nesse processo, os blocos de dados do *peer* 6 (G e H) referentes ao arquivo R.txt são enviados ao *peer* 2. Os *peers* 4 e 5 são responsáveis por enviar todos os blocos de dados referentes ao *peer* falho. Portanto, o *peer* 4 envia os blocos A e B e o *peer* 5 os blocos A e C ao novo responsável, identificado pelo *peer* 6. Como o *peer* 6 já possui os blocos B e C, irá descartar os blocos repetidos. Esse procedimento está representado pelas conexões tracejadas. Após esse processo, o *peer* 6 possuirá todos os blocos de dados do arquivo referente ao *peer* 7.

Da mesma forma que o *peer* 6 identificou seu novo vizinho (*peer* 2), os demais *peers* também terão de faze-lo. Tal ação é representada pelas conexões pontilhadas.

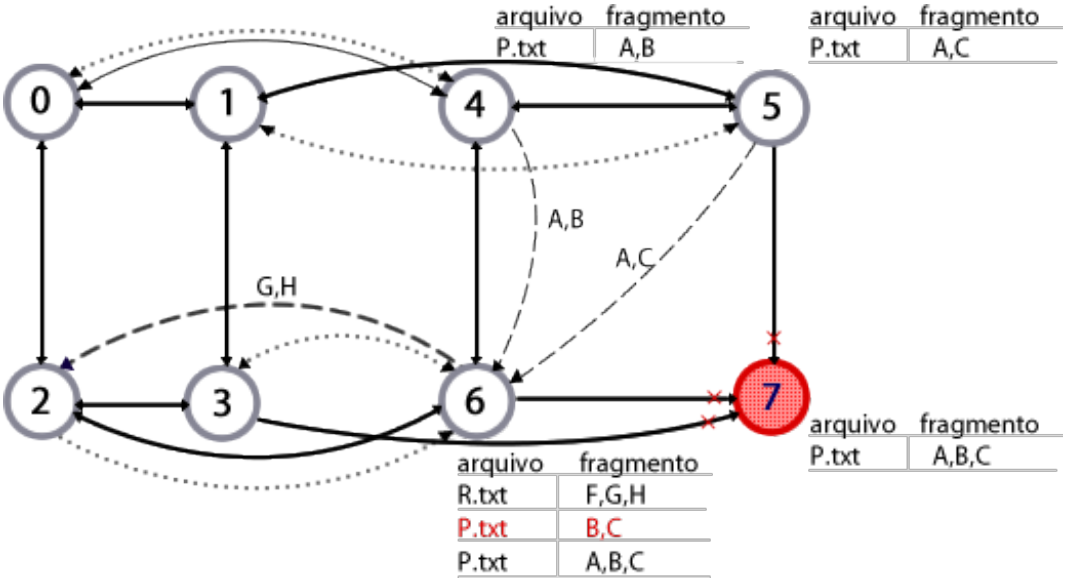


Figura 5.3: Exemplo da redistribuição dos blocos pelos *peers* do sistema
Fonte: [O Autor 2016]

5.4 Tolerância a Falhas

A replicação de dados é uma técnica de tolerância a falhas que garante a disponibilidade dos dados mesmo em um sistema que apresente componentes falhos. Com base na metodologia de replicação empregada neste trabalho, é importante analisar o nível de tolerância a falhas que o sistema garante. Essa análise pode ser feita com base no sistema exibido na Figura 5.4. Ela apresenta os *peers* que compõe o sistema e as réplicas dos blocos referentes aos arquivos armazenados por eles após a distribuição feita no Protocolo de Entrada. As conexões pontilhadas

representam a troca de arquivos entre os *peers* que compõe o mesmo grupo de replicação.

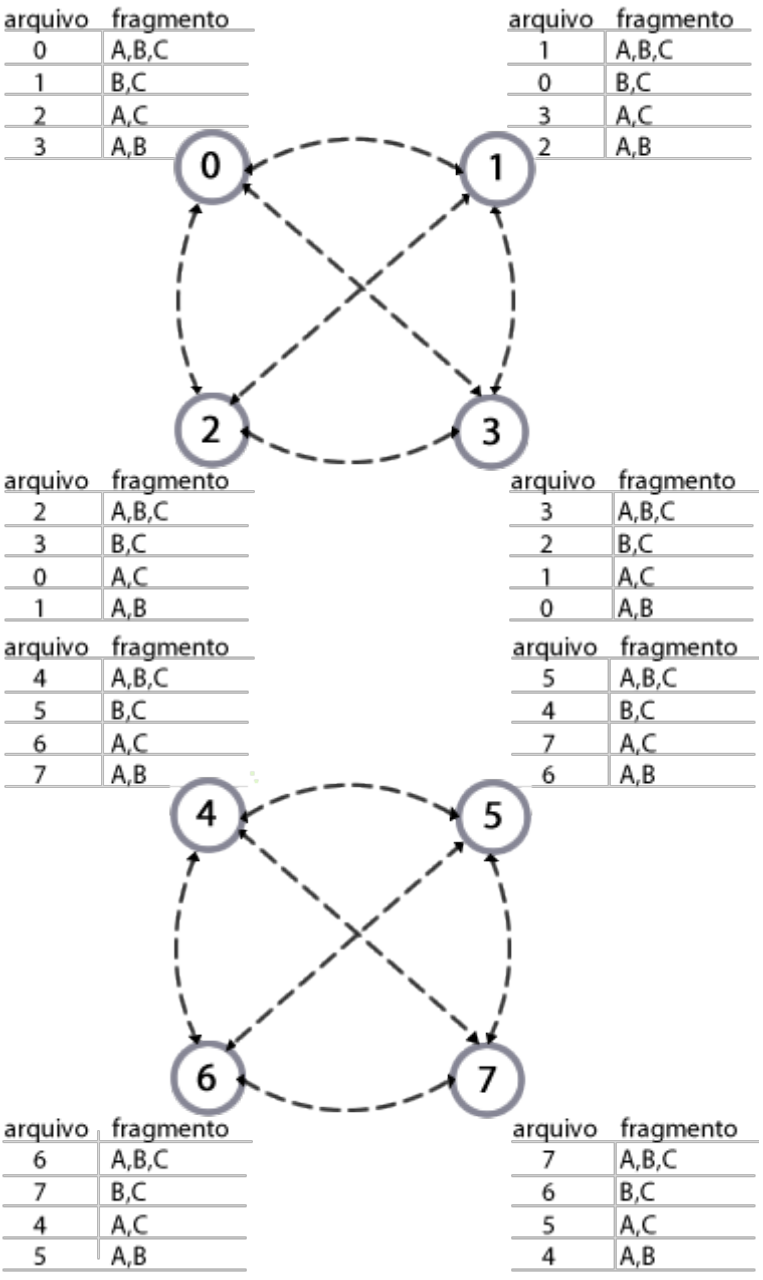


Figura 5.4: Exemplo da redistribuição dos blocos pelos *peers* do sistema
Fonte: [O Autor 2016]

Nota-se que o sistema é composto por 8 *peers* e o fator de replicação (*r*) e fragmentação (*frag*) foram definidos por $log_2 N$, sendo N o número de participantes no sistema. De acordo com o algoritmo explanado na Seção 5.2, tais parâmetros são configurados para que cada *peer* tenha 3 vizinhos, onde cada um receberá 2 blocos de dados referentes ao arquivo replicado.

Considera-se o cenário no qual o arquivo 2_{ABC} é buscado por um *peer* aleatório que não compõe o grupo de replicação do *peer* 2. Não havendo participantes falhos (que armazenam blocos de dados referentes ao arquivo 2), é possível obter o arquivo completo através do contato com o *peer* 2, utilizando o método *lookup(2)* ou contatando dois dos *peers* caracterizados como gerenciadores de réplica do arquivo 2, sendo eles os *peers* 0, 1 e 3, respectivamente.

Em um segundo cenário, no qual o gerenciador de réplica que possui o arquivo original (*peer* 2) torna-se falho, ainda é possível obter o arquivo completo, através do contato com dois dos *peers* 0, 1 e 3. Em um terceiro cenário, no qual os gerenciadores de réplicas 0, 1 e 3 tornam-se instantaneamente falhos, ainda é possível obter o arquivo completo através do contato com o *peer* 2. No entanto, caso o *peer* 2, juntamente com dois outros *peers* gerenciadores tornem-se instantaneamente falhos, não é possível obter a réplica completa do arquivo 2, pois o único gerenciador não-falho (*peer* 3) possui apenas dois blocos de dados referentes ao arquivo 2.

Com base no que foi explanado anteriormente, é possível definir que o sistema VCubeDHT tolera $f = log_2 N$ falhas quando o gerenciador que armazena o arquivo original não é falho, $frag - 1$ falhas quando a falha é identificada no gerenciador que possui o arquivo original, juntamente com outro(s) gerenciador(es) de réplica e, por fim $N - frag$ falhas quando os *peers* falhos são identificados como sendo não-gerenciadores de réplica do arquivo a ser buscado.

Vale ressaltar que para afetar a disponibilidade do sistema, os *peers* devem falhar instantaneamente, pois, caso contrário, o VCubeDHT se encarregará de manter a disponibilidade dos dados por meio da redistribuição dos blocos de dados.

Capítulo 6

Avaliação Experimental

O presente Capítulo tem como objetivo apresentar os principais resultados das simulações efetuadas utilizando o sistema VCubeDHT. Para implementação do sistema foi utilizado a linguagem de programação Java sobre a ferramenta de simulação SimGrid [Legrand, Marchal e Casanova 2016], utilizada para visualização das simulações e obtenção dos resultados. Mais informações sobre o SimGrid podem ser encontradas no Apêndice A.

Este capítulo concentra-se em questões referentes ao impacto do método de propagação de dados proposto por esse sistema, realizando algumas comparações com a metodologia de propagação utilizada pelo sistema HyperDHT.

A Seção 6.1 apresenta as considerações preliminares em relação ao desenvolvimento da estrutura do sistema para realização dos testes. A Seção 6.2 apresenta o método usado pelo SimGrid para calcular o tempo de simulação na troca de mensagens entre os membros da rede. A Seção 6.3 analisa e discute os resultados obtidos durante a replicação de dados no protocolo de entrada. Por fim, a Seção 6.4 tem como objetivo investigar e discutir os resultados adquiridos em relação a redistribuição dos arquivos após o diagnóstico de um *peer* falho na rede.

6.1 Considerações Preliminares

Com o intuito de simular um cenário próximo a uma rede de computadores baseada na Internet, foram construídas diferentes redes constituídas de *peers*, *roteadores* e *links* de comunicação para troca de mensagens entre os participante da rede. Cada cenário de teste é composto por 8, 16, 32, 64, 128, 256 e 512 participantes.

O número de roteadores foi definido por um valor logarítmico, representado por $\log_2 N$,

sendo N o número de participantes da rede. Consequentemente, o número de roteadores presentes em uma rede com 8 participantes foi limitado a 3. Em um cenário cuja rede foi composta por 16 participantes foram definidos 4 roteadores, e assim sucessivamente.

O número de conexões entre os roteadores foi aleatorizada de modo que um roteador não pode se conectar a todos os demais, mas sim com $R - 2$ vizinhos, sendo R o número de roteadores. Além disso, foi definido que um roteador terá no mínimo 1 e no máximo $N - (R - 1)$ participante(s) da rede conectado(s) a ele.

A largura de banda de cada *link* de conexão, seja de um *peer* ou roteador também foi aleatorizada, possuindo valores entre 20 a 40 megabits por segundo. Além disso, a latência de cada *link* de conexão também foi aleatorizada com valores entre 10 a 40 milissegundos.

Para todas as análises feitas em relação ao algoritmo de propagação de dados, o tamanho do arquivo, configurado previamente no arquivo de configuração do SimGrid (*deployment.xml*), variou entre 1, 10, 100 megabytes e 1 gigabyte. Além disso, o roteamento dos pacotes de dados entre os participantes da rede foi feito pelo algoritmo Dijkstra [Dijkstra 1959].

6.2 Cálculo do Tempo de Comunicação no SimGrid

De acordo com [Velho et al. 2011], o cálculo do tempo de comunicação para uma mensagem é representado pela formula $TF = \frac{SF}{F+LF}$, sendo SF o tamanho da mensagem, F a largura de banda e, por fim, LF a soma das latências dos *links* de comunicação utilizados. Além disso, o autor comenta a necessidade de se contabilizar a sobrecarga do protocolo utilizado, tornando-se necessário multiplicar as latências por um fator $\alpha = 10,2$ e as larguras de banda por $\beta = 0,92$.

6.3 Análise do Protocolo de Entrada

A etapa de análise do protocolo de entrada tem como objetivo apresentar os resultados obtidos em relação ao método de propagação de dados utilizado pelos algoritmos HyperDHT e VCubeDHT, destacando o resultado obtido em ambos sistemas durante o Protocolo de Entrada. Em todos os testes, não foram geradas falhas durante a entrada dos participantes na rede.

O critério escolhido para comparação entre os sistemas foi o tempo de simulação total para que os participantes da rede repliquem seus dados. Vale ressaltar que os parâmetros fator de replicação e fragmentação foram ajustados em $log_2 N$, sendo N o número de *peers* do sistema.

Portanto, em um sistema com 8 participantes, um determinado *peer* terá 3 vizinhos que compõem seu grupo de replicação e, no VCubeDHT cada arquivo será fragmentado em 3 blocos.

Como comentado, o Protocolo de Entrada para ambos algoritmos segue o mesmo conceito, porém, ao contrário do HyperDHT, o algoritmo proposto fragmenta suas réplicas e em seguida as propaga em blocos, como mostrado no Capítulo 5.

As Tabelas 6.1 a 6.7, apresentam os resultados dos testes realizados para diferentes redes compostas por 8 a 512 *peers* que replicam um arquivo de 1 *megabyte* a 1 *gigabyte* de tamanho. Em todos os testes realizados o VCubeDHT obteve melhor tempo em relação ao HyperDHT, devido ao fato de que o algoritmo proposto replica um conjunto menor de dados.

No cenário da Tabela 6.1, considerando que cada participante da rede possui um arquivo de 1 *megabyte*, no HyperDHT, cada *peer* transfere *3 megabytes*, totalizando 25.165.824 *bytes* propagados em 11,67 segundos. Já no sistema VCubeDHT, cada *peer* transfere dois blocos de 349.525,33 *bytes*, totalizando 5.592.405,33 *bytes* transferidos em 9,39 segundos.

Considerando o cenário da Tabela 6.7, composto por 512 *peers*, onde cada um possui um arquivo de 1 *megabyte*, no HyperDHT, cada *peer* transfere *9 megabytes*, totalizando 4.831.838.208 *bytes* transferidos em 108,67 segundos. Nesse mesmo cenário, cada *peer* do VCubeDHT transfere oito blocos de 116.508,44 *bytes*, totalizando 4.294.967.296 *bytes* propagados em 101,72 segundos.

É possível notar, em ambos cenários, um número menor de *bytes* a ser transferido pelo VCubeDHT, sendo 19.573.418,67 *bytes* para o primeiro cenário e 536.870.912 *bytes* no segundo. Nas Tabelas 6.1 e 6.7, nota-se um resultado próximo para ambos sistemas. Isso ocorre pois a latência impactada na transferência do arquivo completo pelo HyperDHT é bem próxima a latência impactada no envio dos blocos feitos pelo VCubeDHT. Em razão disso, o ganho de desempenho do VCubeDHT não é muito significativo, mesmo com um número menor de *bytes* enviados. Além disso, é possível observar que o mesmo ocorre para os diferentes cenários.

Nota-se que o desempenho médio obtido pelo VCubeDHT nos diferentes cenários foi de 5,68 segundos para replicação de arquivos com 1 *megabyte*, 58,33 segundos para arquivos de 10 *megabytes*, 567,70 segundos para 100 *megabytes* e por fim, 5946,26 segundos para replicação de arquivos com 1 *gigabyte*.

Tabela 6.1: Tempo (s) para Replicar os Dados no Protocolo de Entrada com 8 Peers

Tamanho do Arquivo	HyperDHT	VCubeDHT
1 MB	11,67	9,39
10 MB	61,41	39,15
100 MB	581,18	366,83
1 GB	5.917,56	3.731,54

Tabela 6.2: Tempo (s) para Replicar os Dados no Protocolo de Entrada com 16 Peers

Tamanho do Arquivo	HyperDHT	VCubeDHT
1 MB	20,29	17,63
10 MB	108,22	81,76
100 MB	1.047,35	769,15
1 GB	10.656,76	7.815,15

Tabela 6.3: Tempo (s) para Replicar os Dados no Protocolo de Entrada com 32 Peers

Tamanho do Arquivo	HyperDHT	VCubeDHT
1 MB	28,64	25,47
10 MB	179,39	140,84
100 MB	1.783,46	1.406,43
1 GB	18.233,48	14.374,41

Tabela 6.4: Tempo (s) para Replicar os Dados no Protocolo de Entrada com 64 Peers

Tamanho do Arquivo	HyperDHT	VCubeDHT
1 MB	47,96	40,53
10 MB	345,92	288,01
100 MB	3.303,90	2.727,09
1 GB	34.475,60	28.479,64

Tabela 6.5: Tempo (s) para Replicar os Dados no Protocolo de Entrada com 128 Peers

Tamanho do Arquivo	HyperDHT	VCubeDHT
1 MB	47,35	42,42
10 MB	341,96	289,03
100 MB	3.211,84	2.823,17
1 GB	33.355,79	28.379,51

Tabela 6.6: Tempo (s) para Replicar os Dados no Protocolo de Entrada com 256 Peers

Tamanho do Arquivo	HyperDHT	VCubeDHT
1 MB	83,93	71,56
10 MB	743,39	653,22
100 MB	7.475,21	6.465,41
1 GB	75.886,69	66.317,94

Tabela 6.7: Tempo (s) para Replicar os Dados no Protocolo de Entrada com 512 Peers

Tamanho do Arquivo	**HyperDHT**	**VCubeDHT**
1 MB	108,67	101,72
10 MB	1.055,13	935,04
100 MB	10.468,69	9.339,63
1 GB	107.317,68	95.121,49

Para melhor visualização dos resultados vistos nas Tabelas 6.1 a 6.7, as Figuras 6.1 a 6.4 apresentam quatro gráficos de comparação entre os sistemas VCubeDHT e HyperDHT, exibindo o tempo total no processo de replicação de dados no Protocolo de Entrada.

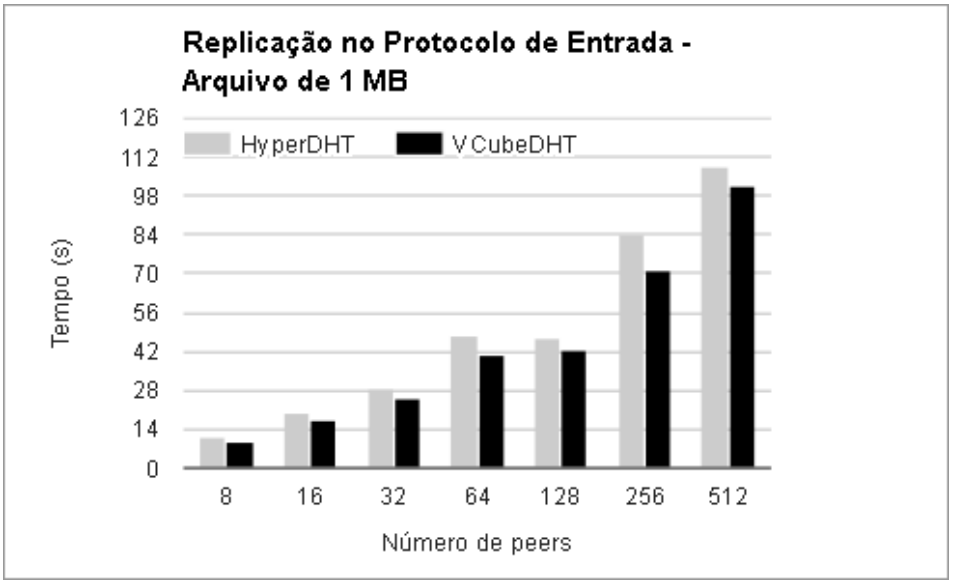


Figura 6.1: Replicação de dados no Protocolo de Entrada com arquivo de 1 megabyte
Fonte: [O Autor 2016]

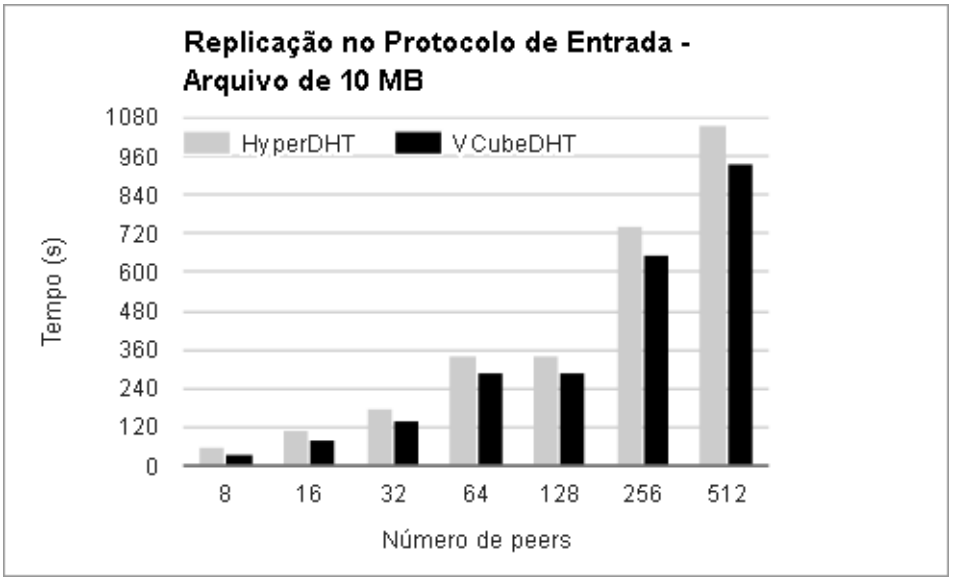


Figura 6.2: Replicação de dados no Protocolo de Entrada com arquivo de 10 megabytes
Fonte: [O Autor 2016]

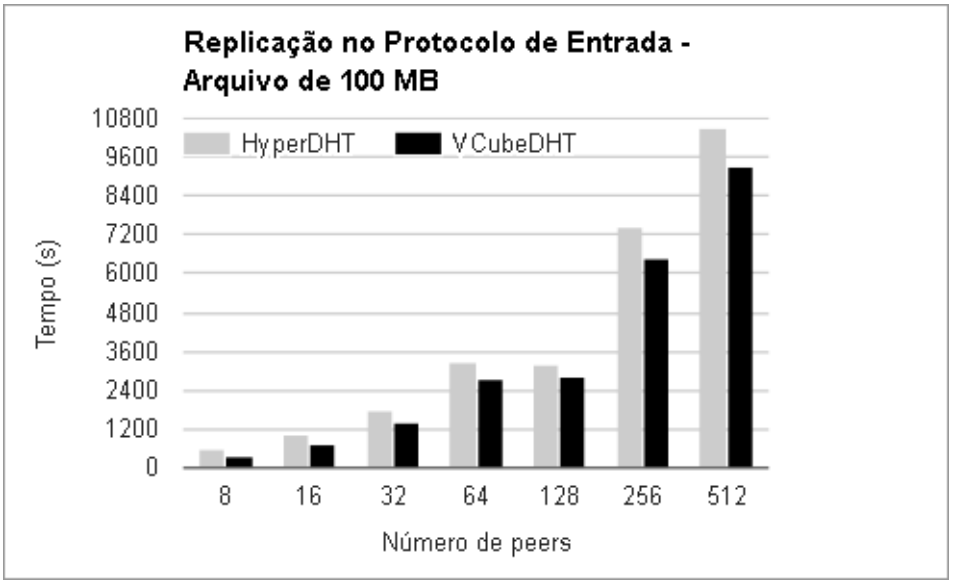


Figura 6.3: Replicação de dados no Protocolo de Entrada com arquivo de 100 megabytes
Fonte: [O Autor 2016]

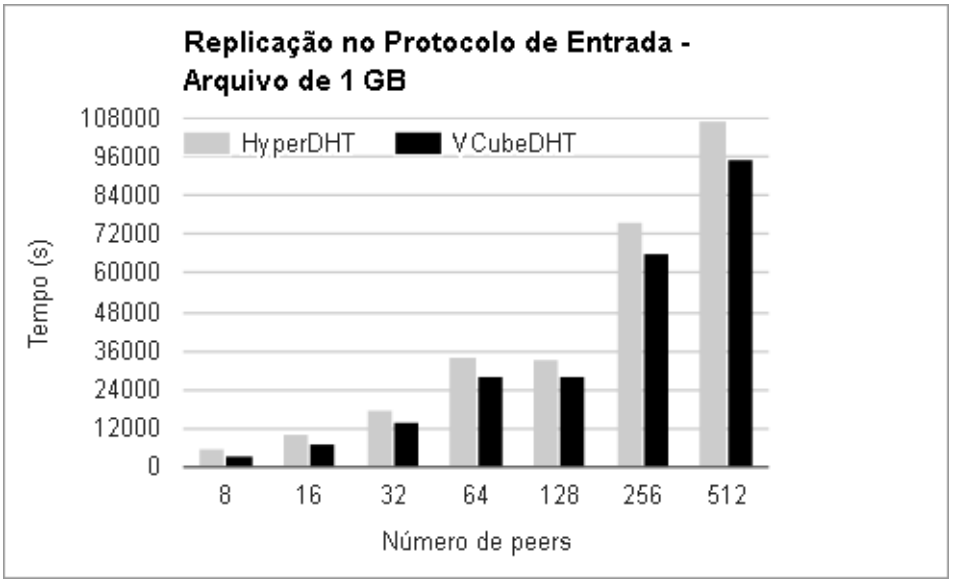


Figura 6.4: Replicação de dados no Protocolo de Entrada com arquivo de 1 gigabyte
Fonte: [O Autor 2016]

Além de analisar o Protocolo de Entrada, torna-se necessário identificar o desempenho dos sistemas na redistribuição dos blocos quando diagnosticado um participante falho na rede.

6.4 Análise da Reorganização do Sistema

A etapa de análise da Reorganização do Sistema busca verificar o comportamento do sistema VCubeDHT diante do diagnóstico de *peers* falhos na rede.

Como comentado, após efetuar o diagnóstico e detectar um participante falho na rede, tanto o HyperDHT como o VCubeDHT tem como objetivo inicial identificar o novo *peer* responsável pelo vértice onde estava o participante falho, de modo a responder pelos dados armazenados por esse. Portanto, após essa etapa, inicia-se o processo de Reorganização do Sistema.

Durante o procedimento de Reorganização do Sistema, foi considerado que nenhum novo *peer* solicite a entrada na rede. A escolha inicial do *peer* falho ocorreu de forma aleatória, porém, posteriormente foi pré-configurada a falha desse para coleta dos resultados em um sistema com maior número de participantes e com arquivos de tamanhos maiores.

O fator de replicação, bem como o de fragmentação, foram definidos por um valor logarítmico, representados por $\log_2 N$, onde N identifica o número de participantes da rede. Portanto, em um sistema com um número de participantes igual a 128, cada *peer* possuirá 7 vizinhos e no

VCubeDHT, cada arquivo será fragmentado em 7 blocos, por exemplo.

Baseado nesse cenário, é possível observar nas Tabelas 6.8 e 6.9 os resultados dos testes realizados para diferentes redes compostas por 8 a 512 *peers*, sendo que em cada uma, cada *peer* possui um arquivo de 1 *megabyte* a 1 *gigabyte* de tamanho. Assim como na análise anterior, o tempo de simulação foi escolhido como critério de desempenho.

Vale ressaltar que não se torna possível, de acordo com o cenário estipulado, comparar as diferentes redes, isto é, a rede composta por 128 participantes com a de 64 participantes. Isso ocorre pois mesmo que o número de réplicas a serem enviadas num sistema composto por 128 participantes seja maior que em um sistema composto por 64 participantes, as características da rede (*links* de comunicação, roteadores, entre outros), serão diferentes para ambos sistemas e impactarão de diferentes formas. Além disso, de acordo com o algoritmo de Reorganização do Sistema, os novos vizinhos a serem identificados também serão diferentes para ambas as redes.

Nota-se que o sistema HyperDHT obteve melhor desempenho em todos os testes executados. Esse resultado pode ser justificado pois, na reorganização do sistema HyperDHT, cada *peer* identifica seu novo vizinho, transmitindo seu arquivo completo para ele. Já no VCubeDHT, haverá a mesma transmissão feita pelo HyperDHT, porém em blocos e em menor quantidade. No entanto, há a necessidade de garantir que o novo *peer* responsável pelo vértice possua todos os blocos de dados relacionados ao participante falho. Dessa forma, todos os *peers* que faziam parte do grupo de replicação do *peer* falho, enviam todos os blocos de dados referentes ao arquivo armazenado pelo *peer* falho ao novo responsável pelo vértice.

Baseado no cenário composto por 8 *peers*, onde o *peer* 7 encontra-se falho e o arquivo a ser replicado possui 1 *megabyte*, no sistema HyperDHT os *peers* 4, 5 e 6 (pertencentes ao grupo de replicação do *peer* 7), enviam uma réplica de seu arquivo aos *peers* 0, 1 e 2 nesta ordem, transferindo um total 3.145.728 *bytes* em 1,50 segundos. No VCubeDHT os *peers* 4, 5 e 6 enviam parte de seu arquivo, em dois blocos de 349.525,33 *bytes*, aos *peers* 0, 1 e 2, respectivamente. Na sequência, os *peers* 4 e 5 enviam 2 blocos de dados de 349.525,33 *bytes* para o *peer* 6 referentes ao *peer* 7, transferindo no total, 3.495.253,33 *bytes* em 2,98 segundos.

Nota-se que neste processo o VCubeDHT transferiu 349.525,33 *bytes* a mais, obtendo um pior desempenho. Como é possível observar nas Tabelas 6.8 e 6.9, a medida que o número de vizinhos, bem como o tamanho do arquivo aumenta, o HyperDHT tem um desempenho

proporcional em relação ao VCubeDHT.

A afirmação acima é justificada após observar os resultados obtidos no cenário composto por 512 *peers*, onde cada um possui um arquivo de 1 *gigabyte*. No VCubeDHT, após a detecção do *peer* falho e a identificação dos novos vizinhos por parte dos 9 *peers* que possuem o *peer* falho como membro do seu grupo de replicação, cada um destes deverá enviar os 8 blocos referentes ao *peer* falho para o novo responsável pelo vértice desocupado. Nota-se que nesse procedimento o VCubeDHT leva um tempo muito superior ao HyperDHT.

Tabela 6.8: Tempo (s) para reorganização do sistema HyperDHT

Número de Peers	**Tamanho do Arquivo**			
	1 MB	**10 MB**	**100 MB**	**1 GB**
8	1,50	8,25	76,91	781,81
16	1,89	6,70	54,77	548,32
32	3,12	9,26	77,36	776,56
64	3,83	9,67	72,14	717,55
128	1,64	6,67	58,27	588,07
256	2,88	9,69	77,79	776,99
512	2,94	10,72	88,55	887,63

Tabela 6.9: Tempo (s) para reorganização do sistema VCubeDHT

Número de Peers	**Tamanho do Arquivo**			
	1 MB	**10 MB**	**100 MB**	**1 GB**
8	2,98	12,74	110,41	1.113,06
16	4,85	15,31	119,96	1.194,30
32	6,92	19,04	155,62	1.547,32
64	11,37	28,73	202,32	1.984,56
128	11,08	33,31	256,20	2.544,47
256	11,67	36,10	280,40	2.788,52
512	11,46	44,15	401,30	4.038,55

Para melhor visualização dos resultados vistos nas Tabelas 6.8 e 6.9, as Figuras 6.5 a 6.8 apresentam quatro gráficos de comparação entre os sistemas VCubeDHT e HyperDHT, exibindo

o tempo total na Reorganização do Sistema.

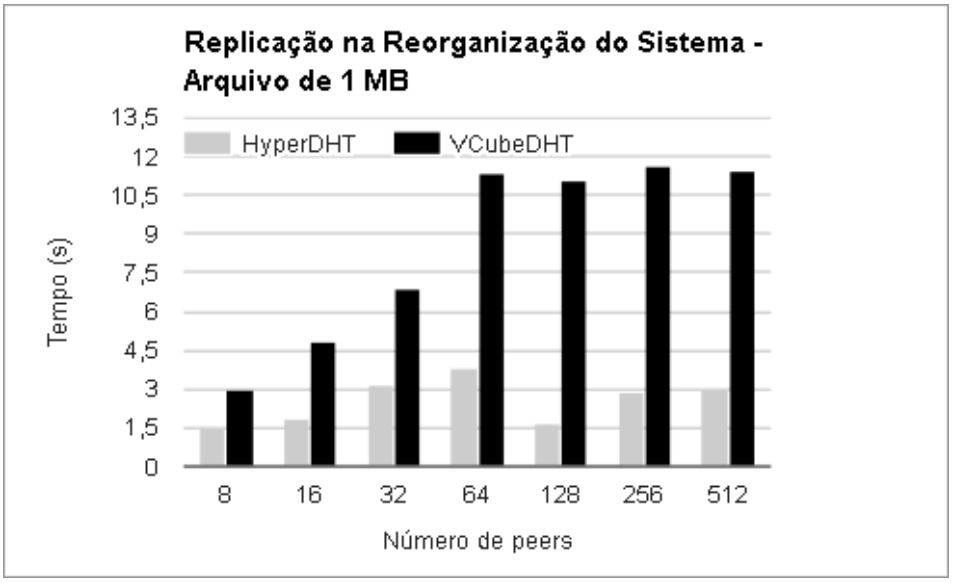


Figura 6.5: Replicação de dados na Reorganização do Sistema com arquivo de 1 megabyte
Fonte: [O Autor 2016]

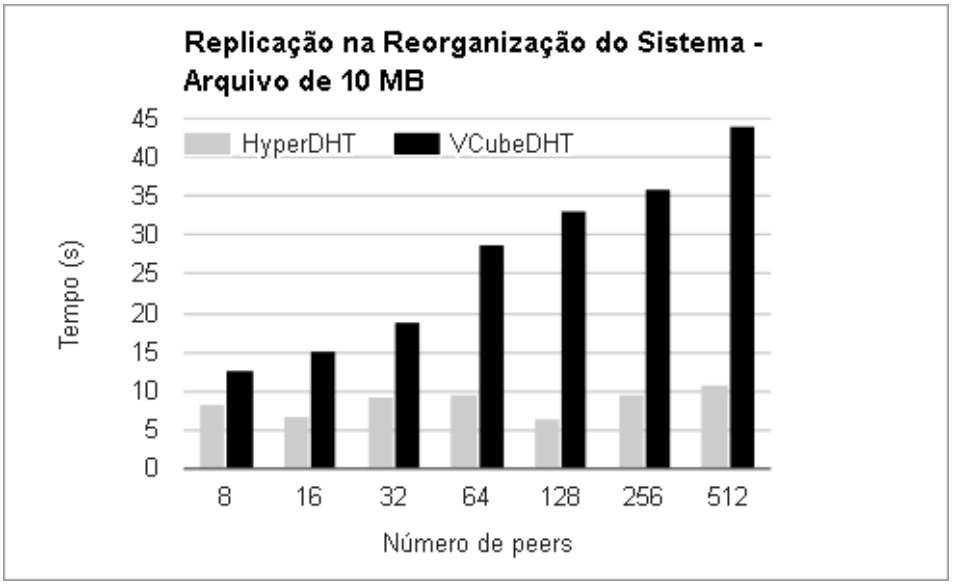


Figura 6.6: Replicação de dados na Reorganização do Sistema com arquivo de 10 megabytes
Fonte: [O Autor 2016]

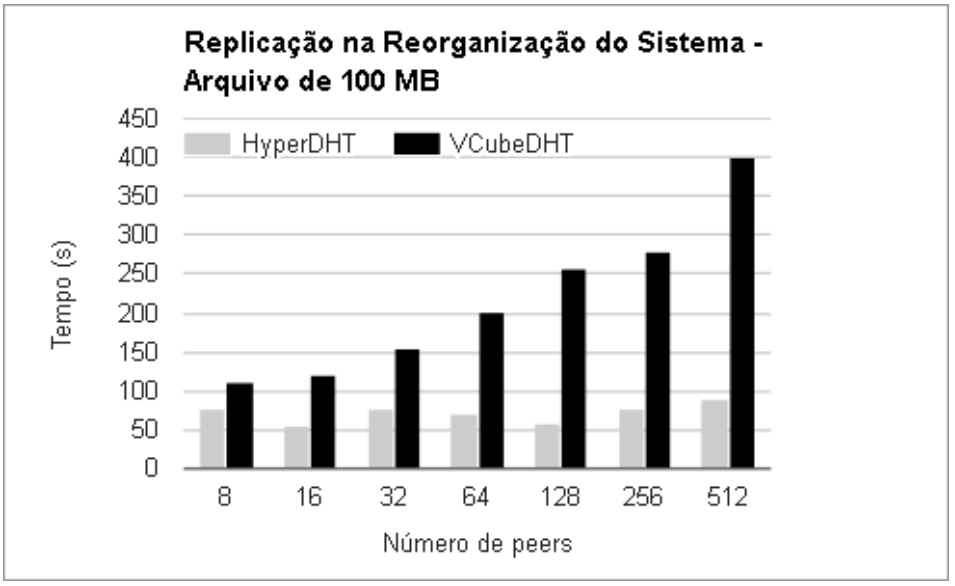


Figura 6.7: Replicação de dados na Reorganização do Sistema com arquivo de 100 megabytes
Fonte: [O Autor 2016]

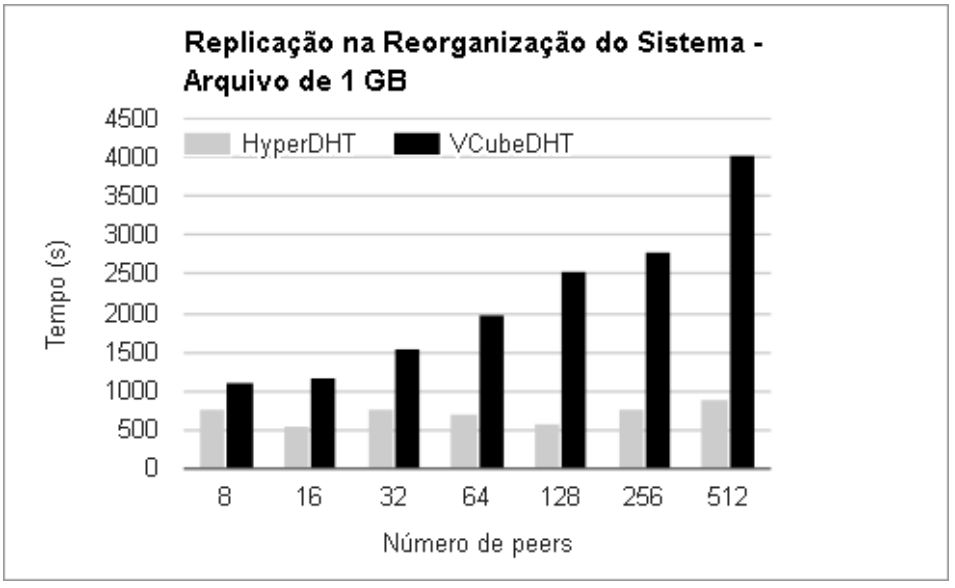


Figura 6.8: Replicação de dados na Reorganização do Sistema com arquivo de 1 gigabyte
Fonte: [O Autor 2016]

A fim de melhorar a implementação do método utilizado para Reorganização do Sistema, foi desenvolvido um método que, além de identificar o bloco de dado que o novo responsável pelo vértice desocupado necessita, identifica o *peer* válido para transferência deste. A Tabela 6.10 exibe os resultados do VCubeDHT utilizando os mesmos parâmetros analisados anteriormente.

No novo método do VCubeDHT, cujo cenário é análogo ao usado na análise anterior, a primeira etapa se manteve, logo, os *peers* 4, 5 e 6, replicam 2 blocos de 349.525,33 *bytes* do arquivo armazenado por eles, aos *peers* 0, 1 e 2, nesta ordem. Em seguida, é identificado pelo *peer* 5, o único bloco de dado necessário para que o *peer* 6 possua o arquivo completo referente ao *peer* falho. Na sequência o *peer* 5 transfere apenas um bloco de dado cujo tamanho é 349.525,33 *bytes* ao *peer* 6. Portanto, o número de *bytes* transferidos pelo VCubeDHT foi de 2.446.677,33 *bytes* contra 3.145.728 *bytes* no HyperDHT, ou seja, 699.050,66 *bytes* a menos.

Nesse cenário, mesmo transferindo um número menor de *bytes*, nota-se, através das Tabelas 6.10 e 6.8, que o VCubeDHT obteve pior desempenho em relação ao HyperDHT, transferindo as réplicas em 2,12 segundos, sendo que o segundo as transferiu em 1,50 segundos.

Tabela 6.10: Tempo (s) para reorganização do sistema VCubeDHT

Número de Peers	**Tamanho do Arquivo**			
	1 MB	**10 MB**	**100 MB**	**1 GB**
8	2,12	7,73	66,48	669,59
16	2,56	7,10	52,44	517,94
32	2,99	9,05	70,34	699,62
64	4,81	10,19	67,35	658,98
128	2,91	6,48	50,08	504,19
256	3,79	10,13	73,50	724,14
512	3,64	11,02	86,45	861,12

As Figuras 6.9 e 6.10 representam os arquivos de *log* da execução dos sistemas VCubeDHT e HyperDHT, respectivamente, representando o cenário comentado acima.

É possível observar que para cada dado enviado, o VCubeDHT obteve um pequeno ganho, porém, o tempo para transferência de cada dado em ambos sistemas é próximo, mesmo que o número de *bytes* enviado pelo HyperDHT seja superior ao VCubeDHT. Além disso, é possível notar que o *peer* 5, após enviar os blocos de dados 1 e 2 ao *peer* 1, aguarda até o mesmo recebe-lo (devido ao envio bloqueante) e, posteriormente realiza o envio do bloco de dado 1 ao *peer* 6. Nota-se que até o momento o VCubeDHT possui uma pequena vantagem, porém, ao emitir o último bloco, o tempo total se torna maior em relação ao obtido pelo HyperDHT.

```
[peer-5: 160000.004] send: peer-4 | rec: peer-0 | size block: 699050.666 | data: 730750_R_1_2
[peer-6: 160000.005] send: peer-5 | rec: peer-1 | size block: 699050.666 | data: 913438_R_1_2
[peer-7: 160000.006] send: peer-6 | rec: peer-2 | size block: 699050.666 | data: 109612_R_1_2
[peer-1: 160001.043]    peer-0 receive piece of peer-4  | data: 730750_R_1_2
[peer-1: 160001.043]    size block: 699050.66 | time to transfer: 1.039

[peer-3: 160001.130]    peer-2 receive piece of peer-6 | data: 109612_R_1_2
[peer-3: 160001.130]    size block: 699050.66 | time to transfer: 1.124

[peer-2: 160001.397]    peer-1 receive piece of peer-5 | replicas: 3 | data: 913438_R_1_2
[peer-2: 160001.397]    size block: 699050.66 | time to transfer: 1.392

[peer-6: 160001.397] send: peer-5 | rec: peer-6 | size block: 349525.333 | data: 127881_R_1

[peer-7: 160002.127]    peer-6 receive piece of peer-5 | data: 127881_R_1
[peer-7: 160002.127]    size block: 349525.33 | time to transfer: 0.729

[peer-7: 165002.127] time send: 160000.004 | time receive: 160002.127 | time total = 2,123
```

Figura 6.9: Arquivo do *log* de execução do VCubeDHT composto por 8 *peers*
Fonte: [O Autor 2016]

```
[peer-5: 160000.004] send: peer-4 | rec: peer-0 | size block: 1048576.0 | data: 730750
[peer-6: 160000.005] send: peer-5 | rec: peer-1 | size block: 1048576.0 | data: 913438
[peer-7: 160000.006] send: peer-6 | rec: peer-2 | size block: 1048576.0 | data: 109612

[peer-1: 160001.281] peer-0 receive piece of peer-4 | replicas: 4 | data: 730750
[peer-1: 160001.281] size file: 1048576.0 | time to transfer: 1.277

[peer-3: 160001.385] peer-2 receive piece of peer-6 | replicas: 4 | data: 109612
[peer-3: 160001.385] size file: 1048576.0 | time to transfer: 1.379

[peer-2: 160001.502] peer-1 receive piece of peer-5 | replicas: 4 | data: 913438
[peer-2: 160001.502] size file: 1048576.0 | time to transfer: 1.497

[peer-2: 165001.502] time send: 160000.004 | time receive: 160001.502 | time total = 1,498
```

Figura 6.10: Arquivo do *log* de execução do HyperDHT composto por 8 *peers*
Fonte: [O Autor 2016]

Mesmo com grande parte dos resultados apresentando um melhor desempenho do sistema HyperDHT, onde os tamanhos do arquivos replicados por cada *peer* são de 1 a 10 *megabyte(s)* respectivamente, nota-se que o VCubeDHT obteve um melhor desempenho em uma rede composta de 32 *peers*, onde os arquivos a serem replicados possuem 10 *megabytes* de tamanho. Tal desempenho se deve ao fato de que a rota percorrida pelo *peer* responsável por replicar o último bloco de dado é impactada por uma menor latência em relação as demais, logo, o resultado na replicação de um arquivo pequeno pelo VCubeDHT se mostrará mais eficiente em relação ao HyperDHT. Caso contrário, possuirá um pior desempenho.

A partir dos dados das Tabelas 6.8 e 6.10, referentes ao HyperDHT e VCubeDHT, respectivamente, notou-se que a diferença entre o tempo de transferência de ambos sistemas diminui

constantemente a medida com que o tamanho do arquivo é maximizado. A partir do momento em que ambos sistemas trabalham com arquivos de 100 *megabytes* de tamanho, é possível observar que o VCubeDHT começa a obter um melhor desempenho em relação ao HyperDHT.

Através do método utilizado pelo simulador para efetuar o cálculo do tempo de comunicação entre os participantes do sistema, notou-se que a latência impacta fortemente no envio de dados com tamanhos menores, como no caso da replicação de arquivos com 1 *megabyte* de tamanho, onde o VCubeDHT obteve pior desempenho. Como apresentado, a medida com que o tamanho do arquivo aumenta, se torna visível a eficiência do VCubeDHT em relação ao HyperDHT.

Para melhor visualização dos resultados presentes nas Tabelas 6.8 e 6.10, as Figuras 6.11 a 6.14 apresentam quatro gráficos de comparação entre os sistemas VCubeDHT e HyperDHT com a nova metodologia de redistribuição, exibindo o tempo total na Reorganização do Sistema.

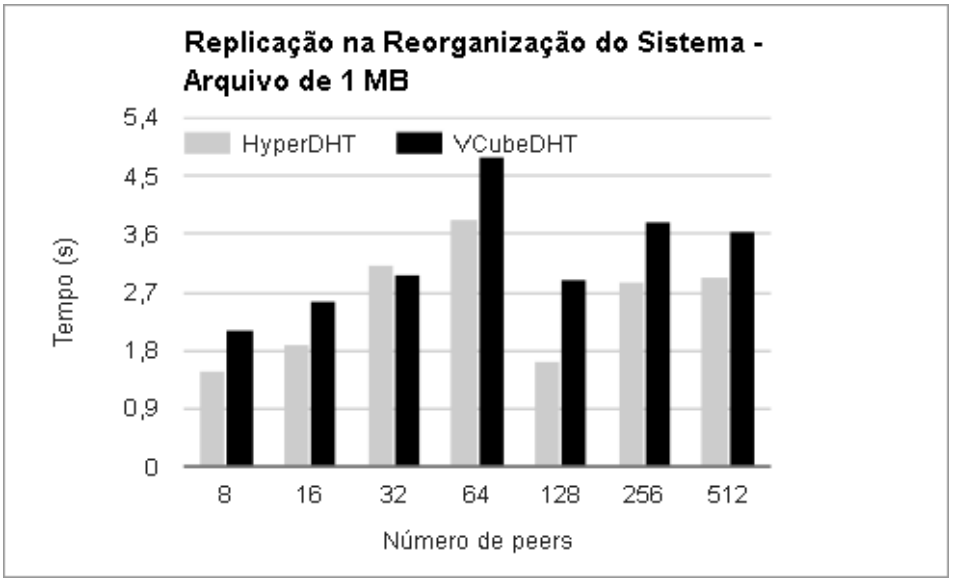


Figura 6.11: Replicação de dados na Reorganização do Sistema com arquivo de 1 megabyte
Fonte: [O Autor 2016]

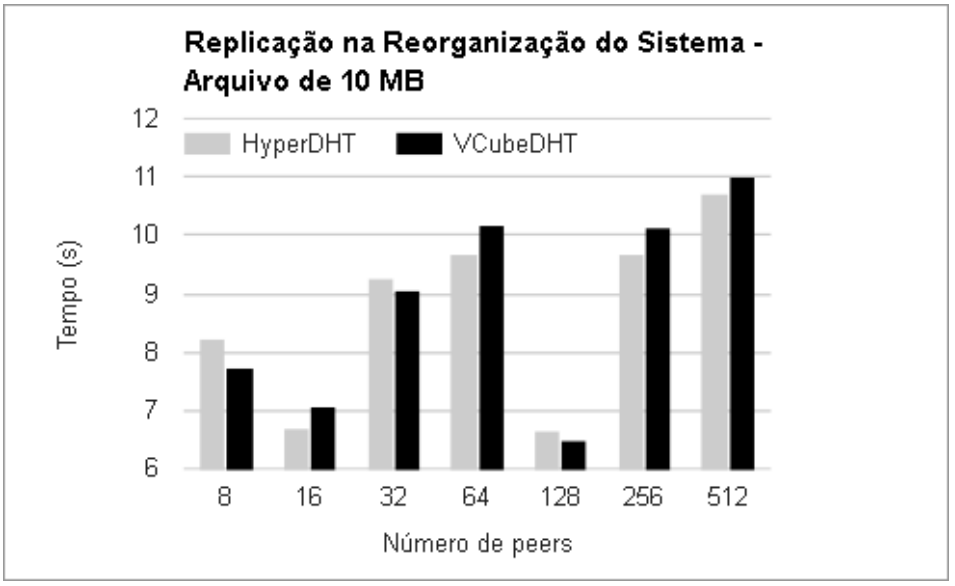


Figura 6.12: Replicação de dados na Reorganização do Sistema com arquivo de 10 megabytes
Fonte: [O Autor 2016]

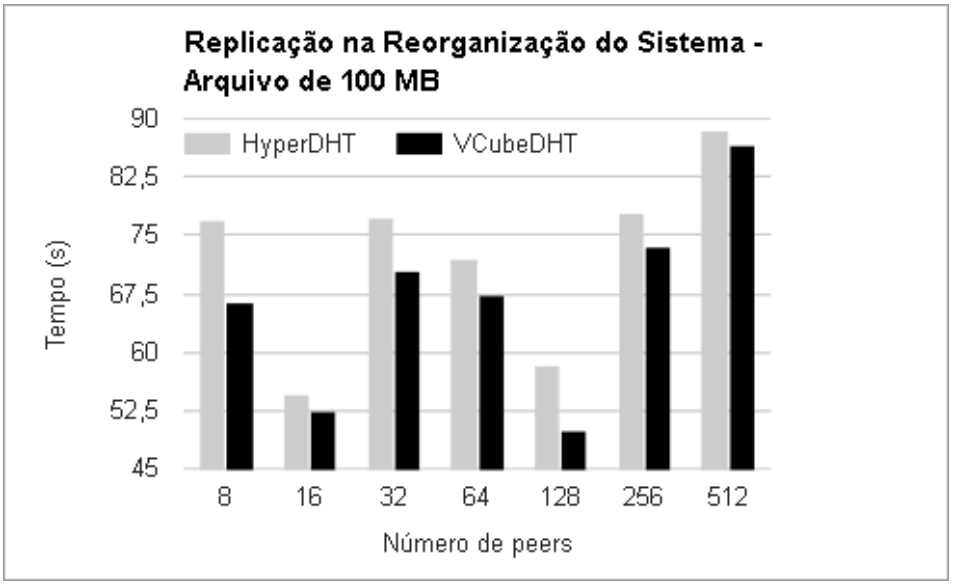


Figura 6.13: Replicação de dados na Reorganização do Sistema com arquivo de 100 megabytes
Fonte: [O Autor 2016]

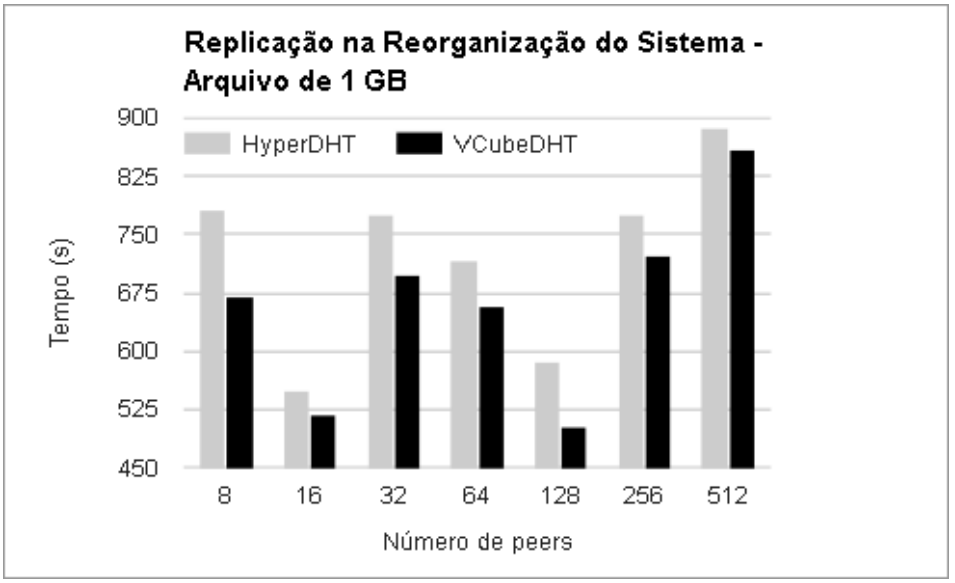


Figura 6.14: Replicação de dados na Reorganização do Sistema com arquivo de 1 gigabyte
Fonte: [O Autor 2016]

6.5 Disponibilidade dos Dados

Além de analisar o desempenho do algoritmo proposto na Reorganização do Sistema, há a necessidade de investigar a disponibilidade dos dados garantida por ele. Sendo assim, a partir de outro modo de visualização dos resultados da Reorganização do Sistema, é possível definir cenários que destacam a disponibilidade dos sistemas HyperDHT e VCubeDHT.

A Tabela 6.15 representa os resultados obtidos na Reorganização do Sistema com arquivos de 100 *megabytes* de tamanho. Analisando a primeira coluna, que representa a rede composta por 8 *peers*, nota-se que se após 66,48 segundos do início do processo de Reorganização do Sistema, os participantes que estavam replicando seus dados se tornem falhos, o sistema HyperDHT não garantiria a disponibilidade total dos dados armazenados pelos *peers*. Isso ocorre pois os *peers* deste sistema não teriam finalizado o processo de reorganização do sistema. Já no VCubeDHT, é possível observar que após esse instante de tempo, os participantes do sistema já haveriam finalizado o procedimento de Reorganização do Sistema, garantindo uma maior disponibilidade dos arquivos armazenados na rede.

Nos cenários onde o VCubeDHT replica arquivos de até 10 *megabytes* de tamanho, como visto nas Figuras 6.11 e 6.12 respectivamente, este não garante uma melhor disponibilidade dos

dados sobre o HyperDHT, visto que o tempo para Reorganização do Sistema no primeiro, em grande parte dos casos, se torna superior.

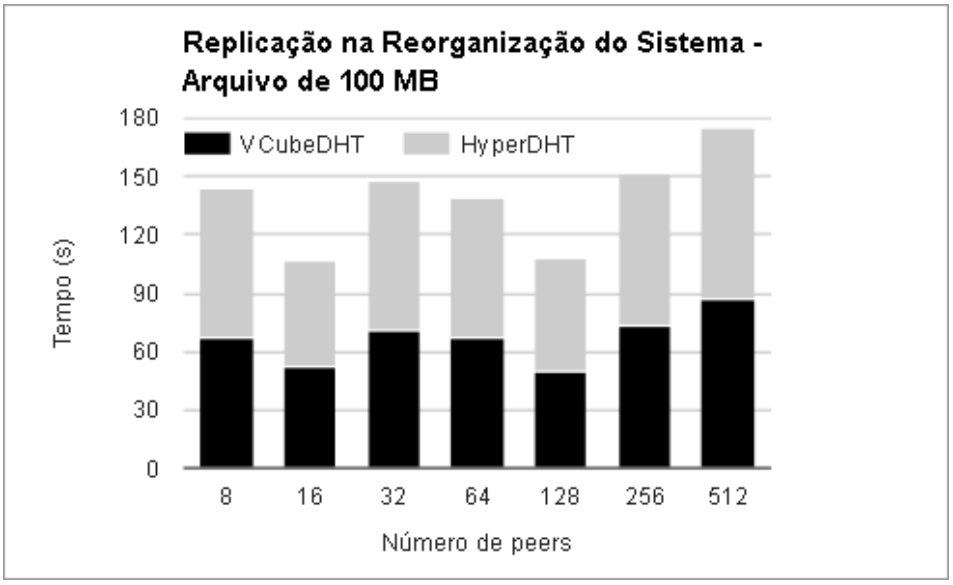


Figura 6.15: Replicação de dados na Reorganização do Sistema com arquivo de 100 *megabyte*
Fonte: [O Autor 2016]

É possível notar que na implementação do algoritmo de replicação de dados do sistema HyperDHT, não há a redistribuição dos dados referentes ao *peer* falho para o novo responsável pelo vértice deixado por esse. Isso ocorre pois o novo responsável possuirá a réplica dos arquivos em questão. Porém, considerando um cenário no qual os *peers* não falhem instantaneamente, após a falha de todos os *peers* pertencentes a um único grupo de replicação, nota-se que as réplicas do primeiro *peer* a se tornar falho não existirão mais no sistema. No entanto, no VCubeDHT, como há a segunda etapa de redistribuição dos blocos referentes ao *peer* falho ao novo responsável, a réplica do primeiro *peer* falho, assim como as outras, estarão disponíveis no sistema.

Com base nas análises feitas até o momento, foi possível identificar que o VCubeDHT preserva as réplicas dos dados armazenados pelos participantes do sistema. Sendo assim, a presente proposta implementa uma abordagem que garante uma maior disponibilidade dos arquivos armazenados na rede em relação a abordagem implementada pelo HyperDHT.

Capítulo 7

Conclusões e Trabalhos Futuros

As DHTs são estruturas de dados distribuídas que utilizam o conceito de chave/valor para o armazenamento de dados em sistemas distribuídos baseados no modelo *Peer-to-Peer*. Elas possuem como características a eficiência na busca de uma determinada informação na rede, bem como a escalabilidade e tolerância a falhas.

A replicação de dados consiste em manter várias cópias de dados, conhecidas como réplicas, a fim de garantir disponibilidade dos mesmos em casos de falhas.

A principal contribuição deste trabalho foi o desenvolvimento de uma solução de DHT que implementa uma alternativa ao método tradicional de replicação de dados. A implementação proposta utiliza o algoritmo VCube a fim de realizar diagnósticos dos participantes da rede, mantendo os limites máximos para a latência.

Na abordagem empregada pelo VCubeDHT, os participantes do sistema replicam seus dados de forma parcial, buscando obter melhorias em relação ao tempo de transmissão, disponibilidade, bem como tolerância a falhas caso um participante da rede responsável por determinado arquivo tenha se tornado inacessível. Para atender esses requisitos, foram efetuados testes comparativos com outro sistema DHT, utilizado como base para o desenvolvimento deste trabalho.

A primeira etapa dos testes buscou analisar o tempo levado pelo HyperDHT e VCubeDHT para conclusão do processo de replicação de dados durante o Protocolo de Entrada. Na segunda etapa, buscou-se avaliar o tempo levado pelos sistemas para replicarem totalmente seus dados após a ocorrência de um participante falho na rede.

Com base nos resultados obtidos, pode-se concluir que a alternativa de replicação baseada em fragmentação implementada pelo VCubeDHT se mostra mais eficiente que a alternativa de replicação total do HyperDHT onde os arquivos a serem replicados possuem tamanhos iguais

ou superiores a 100 *megabytes*. Na emissão de arquivos cujos tamanhos são inferiores a 100 *megabytes*, o algoritmo proposto obtém melhores resultados se a latência impactada na emissão do último bloco de dado referente ao *peer* falho for significantemente menor que as demais.

É importante ressaltar que a alternativa abordada pelo VCubeDHT é interessante pois garante a disponibilidade total dos dados caso os participantes do sistema não se tornem falhos instantaneamente. Porém, no pior caso, o sistema mantém a disponibilidade dos arquivos, tolerando $frag - 1$ falhas, onde $frag$ representa o fator de fragmentação configurado no sistema.

Neste trabalho, as informações armazenadas pelos participantes do sistema não sofreram alterações. Dessa forma, propõe-se como trabalho futuro, a exploração do VCubeDHT para que este suporte trabalhar com arquivos modificados, de modo a garantir a consistência dos dados. Além disso, propõe-se a implementação e execução do VCubeDHT em um ambiente real, ou mesmo simulando diferentes cenários em um ambiente de testes como o PlanetLab.

Apêndice A

Simulador SimGrid

O desenvolvimento de algoritmos distribuídos em um ambiente real requer uma grande quantidade de recursos, muitas vezes não disponíveis nas etapas iniciais de um projeto. Com isso, a utilização de simulares é fundamental, permitindo modelar novas aplicações distribuídas através da criação de máquinas virtuais, *data centers*, entre outros módulos que podem ser adicionados, facilitando a análise das aplicações desenvolvidas.

A ferramenta científica SimGrid [Casanova et al. 2014] é utilizada para simulação de ambientes de programação distribuída, como sistemas P2P, Grades, Nuvens e HPC (*High-Performance Computing*). Ela fornece múltiplos ambientes de programação, como pode ser visto na Figura A.1. Cada um objetiva uma aplicação específica constituída de paradigmas distintos. Dentre esses estão ambientes para simulação de aplicações MPI (*Message Passing Interface*), aplicações genéricas e distribuídas. Além disso, a ferramenta também possui um ambiente que fornece todas funcionalidades de simulação.

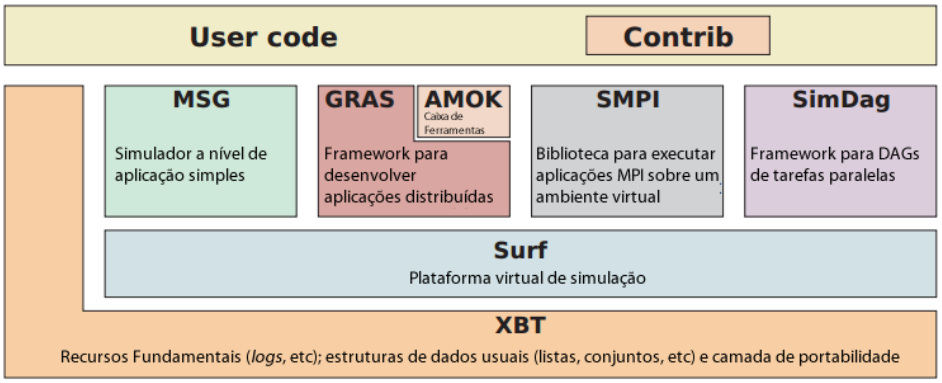


Figura A.1: Arquitetura da ferramenta SimGrid
Fonte: [Oliveira 2007]

A.1 Ambientes da Ferramenta

O ambiente MSG foi o primeiro a ser disponibilizado [Legrand, Marchal e Casanova 2016], sendo utilizado para modelar aplicações, como processos sequenciais concorrentes. Além disso, é útil para modelar problemas teóricos e comparar diferentes heurísticas. Assim como o MSG, outros ambientes se destacam, sendo eles o SimDag, GRAS (*Grid Reality and Simulation*), SMPI, SURF e XBT (*eXtended Bundle of Tools*).

O SimDag é mais utilizado em tarefas paralelas. O GRAS possui semelhanças ao MSG, porém, pode ser executado tanto no simulador como em plataformas reais. De acordo com [Oliveira 2007], o SMPI permite a execução de aplicações MPI. O SURF disponibiliza funcionalidades para simulação de uma plataforma virtual. Por fim, o XBT implementa estruturas de dados usadas no ambiente SURF e providencia suporte à portabilidade do SimGrid. Neste trabalho foi utilizado o MSG, devido a necessidade de realizar apenas a simulação do ambiente.

A.2 Configuração do Ambiente de Simulação

Para execução de qualquer simulação no ambiente MSG, o SimGrid requer dois arquivos de configuração. O primeiro, denominado *platform.xml* é utilizado para definir a plataforma, ou seja, a estrutura física do sistema (por exemplo, *hosts*, roteadores, *links* de comunicação). A Figura A.2 ilustra um arquivo com as configurações básicas de uma plataforma no SimGrid.

No exemplo são definidos quatro processos (identificados por node), sendo distribuídos os processos 1 e 2 para o *cluster* 1 e os processos 3 e 4 para o *cluster* 2. Além disso, foi definido um *links* de comunicação, que por sua vez é responsável pela conexão entre os *clusters*.

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM
"http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">
<AS  id="AS0"  routing="Full">
    <cluster id="cluster1"
           prefix="node-" suffix="" radical="1-2"
           power="1Gf" bw="125MBps" lat="50us"
           bb_bw="2.25GBps" bb_lat="500us"
           router_id="router1"/>
    <cluster id="cluster2"
           prefix="node-" suffix="" radical="3-4"
           power="1Gf" bw="125MBps" lat="50us"
           bb_bw="2.25GBps" bb_lat="500us"
           router_id="router2"/>
    <link id="link_1_2" bandwidth="1.25GBps"
       latency="250us" />
    <ASroute src="cluster1" dst="cluster2"
           gw_src="router1" gw_dst="router2">
           <link_ctn id="link_1_2"/>
    </ASroute>
</AS>
</platform>
```

Figura A.2: Arquivos de configuração *platform.xml*
Fonte: [O Autor 2016]

No segundo arquivo, denominado *deployment.xml*, estão descritas as informações para aplicação, ou seja, estarão presentes os parâmetros de entrada. Neste arquivo é informado o tempo de início de execução do processo, sendo determinado pelo parâmetro *start_time*, bem como o tempo para suspensão do processo, determinado pelo parâmetro *kill_time*. A Figura A.3 apresenta o arquivo *deployment.xml* para a plataforma descrita anteriormente.

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM
"http://simgrid.gforge.inria.fr/simgrid.dtd">
<platform version="3">
    <process host="node-1"
        function="vcube.node"
        kill_time="0.0">
    </process>
    <process host="node-2"
        function="vcube.node"
        start_time="0.0">
    </process>
    <process host="node-3"
        function="vcube.node"
        start_time="0.0">
    </process>
    <process host="node-4"
        function="vcube.node"
        start_time="0.0">
    </process>
</platform>
```

Figura A.3: Arquivos de configuração *deployment.xml*
Fonte: [O Autor 2016]

A.3 Visualização da Simulação

Após a configuração dos arquivos *platform.xml* e *deployment.xml* é necessário desenvolver a aplicação desejada. No SimGrid as aplicações podem ser desenvolvidas utilizando às linguagens de programação Java, C, Scala, e Lua.

Para demonstrar um exemplo de visualização da simulação, foi desenvolvida uma simulação do tempo de envio e recebimento de mensagens entre os processos utilizando o SimGrid, definida por uma estrutura baseada em dois *clusters*. Os principais parâmetros para definição de um *cluster* são a quantidade de processos que o mesmo irá possuir, sua potência, a largura de banda, a latência para os *links*, bem como a especificação de qual roteador será utilizado na aplicação. Estes parâmetros estão apresentados na Figura A.2, sendo eles: *radical*, *power*, *bw*, *lat* e o *router*, respectivamente. A conexão entre estes *clusters* foi estabelecida através de um *link*. Para essa aplicação foi definido que o primeiro processo irá falhar no início da mesma, e através de testes realizados por outros processos, será descoberto o processo falho.

A Figura A.4 apresenta a execução do algoritmo de acordo com as especificações comentadas anteriormente, bem como a apresentação do *host* de cada processo, as ações que serão executadas e também o *timestamp* referente a cada processo.

```
[node-1:vcube_icv.node:(1) 0.000000] [jmsg/INFO]       node-0 send to node-1 [0 0 0 0]
[node-2:vcube_icv.node:(2) 0.000000] [jmsg/INFO]       node-1 send to node-0 [0 0 0 0]
[node-3:vcube_icv.node:(3) 0.000000] [jmsg/INFO]       node-2 send to node-3 [0 0 0 0]
[node-4:vcube_icv.node:(4) 0.000000] [jmsg/INFO]       node-3 send to node-2 [0 0 0 0]
[node-2:vcube_icv.node:(2) 0.001301] [jmsg/INFO]               node-1 receive from node-0 [0 0 0 0]
[node-2:vcube_icv.node:(2) 0.001301] [jmsg/INFO]                       node-1 reply to node-0 [0 0 0 0]
[node-4:vcube_icv.node:(4) 0.001301] [jmsg/INFO]               node-3 receive from node-2 [0 0 0 0]
[node-4:vcube_icv.node:(4) 0.001301] [jmsg/INFO]                       node-3 reply to node-2 [0 0 0 0]
[node-3:vcube_icv.node:(3) 0.001301] [jmsg/INFO]               node-2 receive from node-3 [0 0 0 0]
[node-3:vcube_icv.node:(3) 0.001301] [jmsg/INFO]                       node-2 reply to node-3 [0 0 0 0]
[node-1:vcube_icv.node:(1) 0.001301] [jmsg/INFO]               node-0 receive from node-1 [0 0 0 0]
[node-1:vcube_icv.node:(1) 0.001301] [jmsg/INFO]                       node-0 reply to node-1 [0 0 0 0]
[node-4:vcube_icv.node:(4) 1.002602] [jmsg/INFO]       node-3 send to node-1 [0 0 0 0]
[node-3:vcube_icv.node:(3) 1.002602] [jmsg/INFO]       node-2 send to node-0 [0 0 0 0]
[node-2:vcube_icv.node:(2) 1.002602] [jmsg/INFO]       node-1 send to node-3 [0 0 0 0]
[node-2:vcube_icv.node:(2) 1.003903] [jmsg/INFO]               node-1 receive from node-3 [0 0 0 0]
[node-2:vcube_icv.node:(2) 1.003903] [jmsg/INFO]                       node-1 reply to node-3 [0 0 0 0]
[node-4:vcube_icv.node:(4) 1.003903] [jmsg/INFO]               node-3 receive from node-1 [0 0 0 0]
[node-4:vcube_icv.node:(4) 1.003903] [jmsg/INFO]                       node-3 reply to node-1 [0 0 0 0]
[node-3:vcube_icv.node:(3) 2.002602] [jmsg/INFO] node-2 have fail node-0
```

Figura A.4: *Log* de execução do VCube no SimGrid
Fonte: [O Autor 2016]

Referências Bibliográficas

[Androutsellis-Theotokis e Spinellis 2004] ANDROUTSELLIS-THEOTOKIS, S.; SPINELLIS, D. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, v. 36, p. 335–371, 2004.

[Bianchini e Buskens 1992] BIANCHINI, J. R.; BUSKENS, R. Implementation of online distributed system-level diagnosis theory. *IEEE Transactions on Computers*, IEEE Computer Society, Los Alamitos, CA, USA, v. 41, n. 5, p. 616–626, 1992. ISSN 0018-9340.

[BitTorrent 2016] BITTORRENT. *BitTorrent The Original BitTorrent Client*. 2016. Disponível em: <http://www.bittorrent.com/>.

[Bona et al. 1995] BONA, L. C. E.; DUARTE, E. P.; MELLO, S. L. V.; FONSECA, K. *HyperBone: Uma Rede Overlay Baseada em Hipercubo Virtual sobre a Internet*. Curitiba, PR, BRA, 2006.

[Casanova et al. 2014] CASANOVA, H.; GIERSCH, A.; LEGRAND, A.; QUINSON, M.; SUTER, F. *Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms*. jul. 2014. Disponível em: <https://hal.inria.fr/hal-01017319/PDF/simgrid3-journal.pdf>.

[Clarke et al. 2001] CLARKE, I.; SANDBERG, O.; WILEY, B.; HONG, T. W. Freenet: A distributed anonymous information storage and retrieval system. In: *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*. New York, NY, USA: Springer-Verlag New York, Inc., 2001. p. 46–66. ISBN 3-540-41724-9. Disponível em: <http://dl.acm.org/citation.cfm?id=371931.371977>.

[Cohen 2003] COHEN, B. *Incentives Build Robustness in BitTorrent*. maio 2003. Disponível em: <http://bitconjurer.org/BitTorrent/bittorrentecon.pdf>.

[Coulouris, Dollimore e Kindberg 2007] COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Distributed Systems: Concepts and Design*. 4. ed. Reading: Pearson, 2007.

[Coulouris et al. 2013] COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T.; BLAIR, G. *Sistemas Distribuidos: Conceitos e Projeto*. 5. ed. Reading: Bookman, 2013.

[Dijkstra 1959] DIJKSTRA, E. W. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, v. 1, n. 1, p. 269–271, 1959.

[Duarte e Nanya 1998] DUARTE, E. P.; NANYA, T. *A hierarchical adaptive distributed system-level diagnosis algorithm.* Curitiba, PR, BRA, 1998.

[Elmasri e Navathe 2010] ELMASRI, R.; NAVATHE, S. *Fundamentals of Database Systems*. 6th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0136086209, 9780136086208.

[Fischer, Lynch e Paterson 1985] FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM*, ACM, New York, NY, USA, v. 32, n. 2, p. 374–382, abr. 1985. ISSN 0004-5411. Disponível em: <http://doi.acm.org/10.1145/3149.214121>.

[Ghodsi 2006] GHODSI, A. *Distributed k-ary system: Algorithms for distributed hash tables*. Tese (Doutorado) — Department of Electronic, Computer, and Software Systems, KTH-Royal Institute of Technology, October 2006. Disponível em: <http://www.ist-selfman.org/wiki/images/3/3f/Dissert.pdf>.

[GNutella 2016] GNUTELLA. *GNutella A Protocol for a Revolution*. 2016. Disponível em: <http://rfc-gnutella.sourceforge.net/>.

[Gupta, Liskov e Rodrigues 2004] GUPTA, A.; LISKOV, B.; RODRIGUES, R. Efficient routing for peer-to-peer overlays. In: *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*. Berkeley, CA, USA: USENIX Association, 2004. (NSDI'04), p. 9–9. Disponível em: <http://dl.acm.org/citation.cfm?id=1251175.1251184>.

[Hadzilacos e Toueg 1994] HADZILACOS, V.; TOUEG, S. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Ithaca, NY, USA, 1994.

[Koppe 2013] KOPPE, J. P. *HYPERDHT - DHT DE UM SALTO BASEADA EM HIPERCUBO VIRTUAL DISTRIBUÍDO*. Dissertação (Dissertação de Mestrado) — UFPR – Universidade Federal do Paraná, Curitiba - PR, Agosto 2013.

[Krull, Wu e Molina 1992] KRULL, J. W.; WU, J.; MOLINA, A. M. Evaluation of a fault tolerant distributed broadcast algorithm in hypercube multicomputers. In: *Proceedings of the 1992 ACM Annual Conference on Communications*. New York, NY, USA: ACM, 1992. (CSC '92), p. 459–466. ISBN 0-89791-472-4. Disponível em: <http://doi.acm.org/10.1145/131214.131272>.

[Kshemkalyani e Singhal 2008] KSHEMKALYANI, A. D.; SINGHAL, M. *Distributed Computing: Principles, Algorithms, and Systems*. 1. ed. Reading: Cambridge, 2008.

[Kubiatowicz et al. 2000] KUBIATOWICZ, J.; BINDEL, D.; CHEN, Y.; CZERWINSKI, S.; EATON, P.; GEELS, D.; GUMMADI, R.; RHEA, S.; WEATHERSPOON, H.; WEIMER, W.; WELLS, C.; ZHAO, B. Oceanstore: An architecture for global-scale persistent storage. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 35, n. 11, p. 190–201, nov. 2000. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/356989.357007>.

[Kurose e Ross 2012] KUROSE, J. F.; ROSS, K. W. *Computer Networking: A Top-Down Approach*. 6. ed. Reading: Pearson, 2012.

[Lamport 1978] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 7, p. 558–565, jul. 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359545.359563>.

[Legrand, Marchal e Casanova 2016] LEGRAND, A.; MARCHAL, L.; CASANOVA, H. *Scheduling Distributed Applications: the SimGrid Simulation Framework*. 2016. Disponível em: <http://ieeexplore.ieee.org/document/1199362/>.

[Melo, Vieira e Libório 2012] MELO, C. A. V.; VIEIRA, D.; LIBÓRIO, J. da M. *Impacto do Churn em Políticas de Gerenciamento de Objetos em Sistemas CDN-P2P*. Manaus, AM, BRA, 2012.

[Mockapetris 2016] MOCKAPETRIS, P. *DOMAIN NAMES - CONCEPTS AND FACILITIES*. 2016. Disponível em: <https://tools.ietf.org/html/rfc1034>.

[Monnerat 2010] MONNERAT, L. R. R. *Tabelas Hash Distribuídas Com Consulta de Um Salto Com Baixa Carga de Manutenção*. Tese (Tese de Doutorado) — Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Setembro 2010.

[Napster 2016] NAPSTER. *Napster Serviço de música por assinatura*. 2016. Disponível em: <http://http://br.napster.com/>.

[Neto e Rodrigues 2015] NETO, A. B.; RODRIGUES, L. A. *Estudo comparativo de ferramentas de simulação para computação em nuvem*. Cascavel, PR, BRA, 2015.

[Oliveira 2007] OLIVEIRA, L. J. de. *Comparação de ferramentas de simulação de grades computacionais*. Dissertação (Dissertação de Mestrado) — UNESP – Universidade Estadual Paulista, São José do Rio Preto - SP, Novembro 2007.

[Patterson, Gibson e Katz 1988] PATTERSON, D. A.; GIBSON, G.; KATZ, R. H. A case for redundant arrays of inexpensive disks (raid). *SIGMOD Rec.*, ACM, New York, NY, USA, v. 17, n. 3, p. 109–116, jun. 1988. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/971701.50214>.

[Ratnasamy et al. 2001] RATNASAMY, S.; FRANCIS, P.; HANDLEY, M.; KARP, R.; SHENKER, S. A scalable content-addressable network. In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. New York, NY, USA: ACM, 2001. (SIGCOMM '01), p. 161–172. ISBN 1-58113-411-8. Disponível em: <http://doi.acm.org/10.1145/383059.383072>.

[Rodrigues 2014] RODRIGUES, L. A. *Uma solução autonômica para k-exclusão mútua em sistemas distribuídos*. Tese (Tese de Doutorado) — Universidade Federal do Paraná, Curitiba, PR, Agosto 2014.

[Rowstron e Druschel 2001] ROWSTRON, A.; DRUSCHEL, P. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 35, n. 5, p. 188–201, out. 2001. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/502059.502053>.

[Rowstron e Druschel 2001] ROWSTRON, A. I. T.; DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. London, UK, UK: Springer-Verlag, 2001. (Middleware '01), p. 329–350. ISBN 3-540-42800-3. Disponível em: <http://dl.acm.org/citation.cfm?id=646591.697650>.

[Ruoso 2013] RUOSO, V. K. *Uma estratégia de testes logarítmica para o algoritmo Hi-ADSD*. Dissertação (Dissertação de Mestrado) — Universidade Federal do Paraná, Curitiba, PR, Maio 2013.

[Schneider 1990] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 22, n. 4, p. 299–319, dez. 1990. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/98163.98167>.

[Stoica et al. 2001] STOICA, I.; MORRIS, R.; KARGER, D.; KAASHOEK M. F.; BALAKRISHNAN, H. 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*. New York, NY, USA: ACM, 2001. (SIGCOMM '01), p. 149–160. ISBN 1-58113-411-8. Disponível em: <http://doi.acm.org/10.1145/383059.383071>.

[Szymaniak, Pierre e Steen 2005] SZYMANIAK, M.; PIERRE, G.; STEEN, M. v. Latency-driven replica placement. In: *Proceedings of the The 2005 Symposium on Applications and the Internet*. Washington, DC, USA: IEEE Computer Society, 2005. (SAINT '05), p. 399–405. ISBN 0-7695-2262-9. Disponível em: <http://dx.doi.org/10.1109/SAINT.2005.37>.

[Tanenbaum e Steen 2007] TANENBAUM, A. S.; STEEN, M. V. *Sistemas Distribuídos: Princípios e Paradigmas*. 2. ed. Reading: Prentice-Hall, 2007.

[Velho et al. 2011] VELHO, P.; SCHNORR, L.; CASANOVA, H.; LEGRAND, A. *Flow-level network models: have we reached the limits?* 2011. Disponível em: <https://hal.inria.fr/hal-00646896/document>.

[Wang, Yin e Yu 2005] WANG, X.; YIN, Y. L.; YU, H. Finding collisions in the full sha-1. In: *Proceedings of the 25th Annual International Conference on Advances in Cryptology*. Berlin, Heidelberg: Springer-Verlag, 2005. (CRYPTO'05), p. 17–36. ISBN 3-540-28114-2, 978-3-540-28114-6. Disponível em: <http://dx.doi.org/10.1007/11535218_2>.

[Weber 2008] WEBER, A. *Um Algoritmo de Diagnóstico Distribuído para Redes Particionáveis de Topologia Arbitrária*. Tese (Tese de Doutorado) — Universidade Tecnológica Federal do Paraná, Curitiba, PR, Maio 2008.

[Wiesmann et al. 2000] WIESMANN, M.; PEDONE, F.; SCHIPER, A.; KEMME, B.; ALONSO, G. Understanding replication in databases and distributed systems. In: *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*. Washington, DC, USA: IEEE Computer Society, 2000. (ICDCS '00), p. 464–. ISBN 0-7695-0601-1. Disponível em: <http://dl.acm.org/citation.cfm?id=850927.851782>.

[Yamamoto, Maruta e Oie 2005] YAMAMOTO, H.; MARUTA, D.; OIE, Y. Replication methods for load balancing on distributed storages in p2p networks. In: *Proceedings of the The 2005 Symposium on Applications and the Internet*. Washington, DC, USA: IEEE Computer Society, 2005. (SAINT '05), p. 264–271. ISBN 0-7695-2262-9. Disponível em: <http://dx.doi.org/10.1109/SAINT.2005.53>.

[Zhang et al. 2011] ZHANG, C.; DHUNGEL, P.; WU, D.; ROSS, K. W. Unraveling the bittorrent ecosystem. *IEEE Trans. Parallel Distrib. Syst.*, IEEE Press, Piscataway, NJ, USA, v. 22, n. 7, p. 1164–1177, jul. 2011. ISSN 1045-9219. Disponível em: <http://dx.doi.org/10.1109/TPDS.2010.123>.

[Zhao, Kubiatowicz e Joseph 2001] ZHAO, B. Y.; KUBIATOWICZ, J. D.; JOSEPH, A. D. *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and*. Berkeley, CA, USA, 2001.