UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Alisson Rafael Appio

ALGORITMO DISTRIBUÍDO PARA BACKUP REATIVO TOLERANTE A FALTAS BIZANTINAS EM REDES PEER-TO-PEER

Florianópolis

2012

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Alisson Rafael Appio

ALGORITMO DISTRIBUÍDO PARA BACKUP REATIVO TOLERANTE A FALTAS BIZANTINAS EM REDES PEER-TO-PEER

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Mestre em Ciência da Computação.
Orientadora: Profa. Dr. Carla Merkle Westphall

Florianópolis

2012

Alisson Rafael Appio

ALGORITMO DISTRIBUÍDO PARA BACKUP REATIVO TOLERANTE A FALTAS BIZANTINAS EM REDES PEER-TO-PEER

Esta Dissertação foi julgada adequada para obtenção do Título de Mestre em ciência da computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computção.
Florianópolis, 21/11/2012.

Ronaldo dos Santos Mello, Dr.
Coordenador do Curso

Banca Examinadora:

Carla Merkle Westphall, Dr.
Orientadora

Elias Procópio Duarte Jr., Dr., UFPR

João Bosco Mangueira Sobral, Dr., UFSC

Mario Antonio Ribeiro Dantas, Dr., UFSC

A minha esposa Glizauda que me apoio neste trabalho e meu filho Leonardo pela compreensão durante minha ausência para concepção deste trabalho.

Simples pode ser mais difícil que complexo. Você tem que trabalhar muito para chegar a um pensamento claro e fazer o simples.

Steve Jobs

AGRADECIMENTOS

Agradeço primeiramente a Deus, pela vida, por estar sempre ao meu lado me iluminando e guiando para as escolhas certas.

Agradeço aos professores que contribuíram diretamente ou indiretamente para o bom andamento deste trabalho, em especial a minha orientadora prof. Carla que aceitou, incentivou, apoiou e me orientou durante a concepção deste trabalho.

Agradeço a minha esposa que sempre me incentivou, tornando possível e viável a construção deste trabalho, a minha mãe e meu pai por sempre terem apoiado e incentivado meus estudos.

Por último, mas não menos importante, agradeço a Hoplon pelos horários flexíveis para realização deste trabalho.

RESUMO

Backup é uma cópia de dados para outro dispositivo de armazenamento com o propósito de restaurar o dado em caso de perda do original. Redes *Peer-to-Peer* (P2P) tem sido utilizadas para armazenar dados de usuários. Neste tipo de rede, a topologia muda constantemente e *peers* que estão armazenando o *backup* podem ser desligados sem nenhum aviso prévio dificultando a restauração do *backup* que foi distribuído, alguns *peers* podem ser maliciosos podendo negar o serviço de restauração, corromper o backup e até mesmo modificar o *backup* inserindo algum vírus. Neste contexto, temos dois grandes desafios: como assegurar a disponibilidade para restaurar o *backup* sem necessitar realizar uma cópia global; como assegurar que o dado é confiável e está integro independente do dono do arquivo ter realizado verificações de integridade quando o dado estava replicado na rede P2P. Para resolver estes problemas, este trabalho apresenta um algoritmo para *backup* P2P projetado e modelado para assegurar alta disponibilidade para restaurar *backup* usando o conceito de uma doença epidêmica. Para evitar que o comportamento epidêmico replique o dado para toda a rede, o algoritmo controla de forma distribuída o número de *peers* que estão armazenando o *backup*. O algoritmo utiliza recursos de *hardware* para otimizar todo o processo de armazenamento e restauração do *backup*. Para assegurar que o dado é confiável e está integro mesmo na presença de um *peer* malicioso, é usado o protocolo de acordo bizantino com os *peers* que estão armazenando o *backup* sem necessitar que o dono do arquivo fique *online*. Em nossos experimentos foi possível constatar uma alta disponibilidade para restaurar o arquivo chegando a taxas entre 97% a 99.9% mantendo um mínimo de redundância do *backup* na rede P2P.

Palavras-chave: Computação Distribuída, Backup Distribuído, Peer-to-Peer, Tolerância a Faltas Bizantinas.

ABSTRACT

A backup is a copy of data to another storage device with the proposed to restore the data in the case of the original data is lost. Peer-to-Peer (P2P) computer network is used to user's storage data. In this network, the topology is changing every time and the peers that are storage backup can be offline without any previous warning and hindering the backup restore, some peers may be classified as malicious, it can deny of restore service, corrupt the backup and it can insert a virus in the backup. In this context, we have two major challenges: how to ensure the availability to restore the backup without requiring to copy data to all peers; how to ensure that the data is reliable regardless owner's data have performed checks of integrity in the remote peer. To solve this problems, this work present an algorithm to backup in P2P network, the algorithm was designed to ensure high availability to restore backup using the concept on an epidemic disease. To avoid replicate data to all peers, the algorithm controls the number of peers that are storage the distributed backup. The algorithm uses hardware devices to optimize all process of storage and restore the backup. To ensure that the data is reliable even in the presence a malicious peer, the algorithm uses the Byzantine agreement protocol with the peers are storage backup without regardless owner's data is online. In our experiments was verified a high availability rate to restore the file, this rates vary between 97% to 99.9% maintaining the minimum redundancy of backup in P2P network.

Keywords: Distributed Computing, Distributed Backup, Peer-to-Peer, Byzantine Fault Tolerance.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

P2P *Peer-to-Peer*

GUID *Globally Unique Identifier*

TTL *Time to Live*

DHT *Distributed Hash Table*

DoS *Denial-of-Service*

DNA *Deoxyribonucleic acid*

RNA *Ribonucleic acid*

RPC *Remote Procedure Call*

API *Application Programming Interface*

TCP *Transmission Control Protocol*

UDP *User Datagram Protocol*

FIFO *First In, First Out*

SUMÁRIO

1 INTRODUÇÃO

Backup é uma cópia de dados para outro dispositivo de armazenamento com o propósito de restaurar o dado em caso de perda do original. O *backup* distribuído pode ser classificado em três grupos: *Backup* em cloud (BESSANI *et al.*, 2011), *backup* em servidor (KUBIATOWICZ *et al.*, 2000) e *backup* em arquitetura de redes *Peer-to-Peer* (P2P) (LANDERS *et al.*, 2004), (DABEK *et al.*, 2001), (LI; DABEK, 2006), (TRAN *et al.*, 2008) e (BATTEN *et al.*, 2002).

Redes P2P são compostas por nodos que tem capacidade para atuar como cliente e servidor ao mesmo tempo, são descentralizadas e tem alta escalabilidade. A topologia da rede muda constantemente com entrada e saída de membros. Outra característica de redes P2P é a sua capacidade para compartilhar recursos como ciclos de CPU, arquivos e espaço em disco (COULOURIS; DOLLIMORE, 2005).

Garantir a disponibilidade dos dados, a confiabilidade e a integridade são características importantes que devem ser consideradas em soluções de *backup* P2P. O arquivo restaurado deve estar intacto como foi disponibilizado pelo seu dono para o sistema de *backup* ser considerado confiável.

Uma das características de sistemas P2P, é sua capacidade de adaptação com a entrada e saída de membros sem aviso prévio. Como *peers* podem estar desligados (*offline*), a disponibilidade para restaurar o *backup* armazenado na rede é um dos problemas em aberto na literatura da área (DEFRANCE *et al.*, 2011), (TOKA *et al.*, 2012), (DUARTE *et al.*, 2010) e (PÀMIES-JUÁREZ *et al.*, 2010). Este problema é parcialmente resolvido quando é feita uma cópia global do arquivo para todos os membros da rede.

Os *backups* em arquiteturas de redes P2P podem ser classificados em dois grandes grupos. O primeiro grupo representa um *backup* global, onde é copiado o arquivo para todos os membros do sistema. O segundo grupo realiza uma cópia para um subconjunto da população de membros do sistema, normalmente uma cópia para os parceiros (figura 1). Cada círculo na figura representa um nodo, a conectividade entre os nodos são representadas pelas linhas que ligam um nodo a outro. Nodos em amarelo estão armazenando *backup*. A imagem (a) mostra a cópia global do arquivo. A imagem (b) mostra uma cópia parcial, somente para nodos parceiros. São considerados nodos parceiros de um determinado nodo, todos os nodos que estão diretamente conectados a este nodo.

Redes P2P são sujeitas a problemas de disponibilidade para restaurar o *backup*. Garantir a disponibilidade para restaurar o *backup* considerando o *churn* (entrada e saída de *peers*) deste tipo de rede, é um conceitos importan-

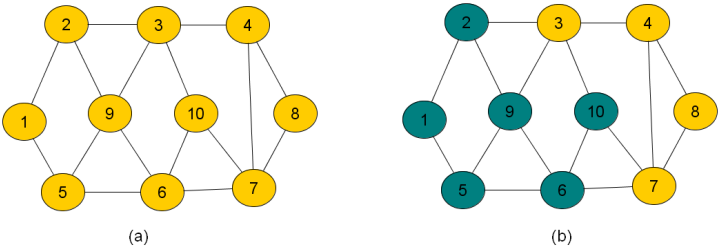

Figura 1: *Backup* em rede P2P.

tes que deve ser tratado *backup* P2P (PÀMIES-JUÁREZ *et al.*, 2010). Uma das características destas redes é que nodos podem entrar e sair do sistema sem aviso prévio. A figura 2 evidencia este problema quando o armazenamento remoto é realizado somente em nodos parceiros, tornando impossível restaurar o *backup*. Na imagem (a), o nodo 4 (amarelo) realiza a cópia do arquivo para seus parceiros, nodos 3, 7 e 8. A imagem (b) mostra a impossibilidade de restaurar o arquivo quando os parceiros estão *offline*.

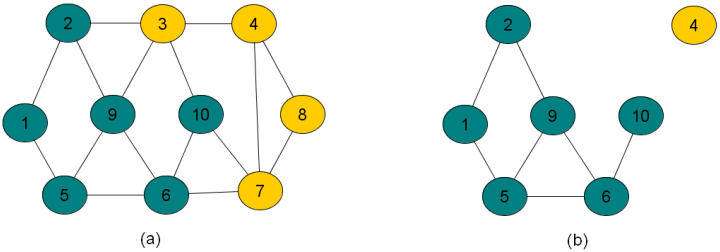

Figura 2: Problema de disponibilidade para restaurar o *backup* em rede P2P.

Outro problema na área, está relacionado com a confiabilidade do arquivo armazenado remotamente (TRAN *et al.*, 2008), (LIU *et al.*, 2012). Segundo Pàmies-Juárez *et al.* (2010), a confiabilidade do *backup* P2P deve atender dois requisitos principais: a) o arquivo remoto sempre deve estar disponível para o dono fazer a restauração; b) o processo de manutenção do dado deve recuperar a perda da informação. Para nós, além dos dois requisitos citados, a confiabilidade é a capacidade do sistema realizar e manter o *backup* distribuído intacto / integro, independente da existência de nodos maliciosos no sistema. A confiabilidade pode ser classificada em dois grupos: confiabilidade centralizada ou confiabilidade distribuída. A confiabilidade centralizada, depende que o dono do arquivo esteja *online* para verificar o arquivo remoto. Para isto,

o dono troca mensagens com os nodos que estão armazenando o seu *backup*. A confiabilidade distribuída não exige que o dono do arquivo esteja *online*, todos os nodos envolvidos no *backup* trocam mensagens entre si para verificar a integridade do arquivo. A figura 3 apresenta estes dois grupos, onde o nodo com o identificador *3* (pintado com a cor verde) é o dono do arquivo. Na imagem (a), o processo de integridade é centralizado no dono do arquivo, o dono do arquivo verifica em cada nodo remoto que está armazenando o *backup* se o dado está integro. Na imagem (b), a checagem da integridade é distribuída entre todos os nodos que possuem o *backup* (nodos em amarelo). Durante a verificação de integridade distribuída, um nodo que está armazenando o *backup* pode usar qualquer rota para se comunicar com outro nodo que também possui uma cópia do *backup*.

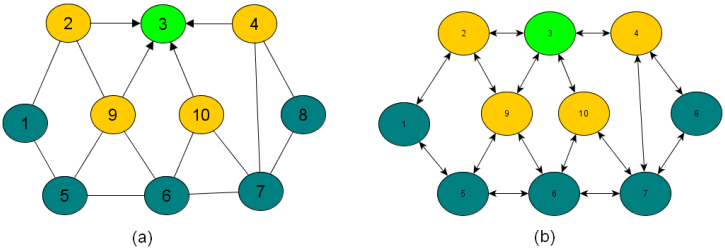

Figura 3: Problema de confiabilidade do *backup* em rede P2P.

1.1 QUESTÕES DE PESQUISA

Este trabalho apresenta uma estratégia de *backup* distribuído em redes P2P, onde as principais perguntas de pesquisa são:

- Como assegurar disponibilidade para restaurar o *backup* sem necessitar realizar uma cópia global?

- Como manter o *backup* distribuído intacto / integro independente de nodos maliciosos e sem necessitar que o dono realize verificações de integridade no dado distribuído?

1.2 CONTRIBUIÇÕES DA DISSERTAÇÃO

A principal contribuição e objetivo deste trabalho é a especificação de uma estratégia de *backup* distribuído para redes P2P que resolve os seguintes problemas:

- Disponibilidade para restaurar o *backup* sem necessitar copiar o arquivo para todos os membros da rede, descobrindo o valor para a variável que representa o número de réplicas ativas que estão armazenando o *backup*.
- Confiabilidade e integridade do *backup* remoto independente do dono estar *online* ou *offline*.
- Tolerar faltas bizantinas em todas as fases do *backup* (cópia, migração e restauração) assegurando confiabilidade e integridade do arquivo disponibilizado remotamente sem necessitar que o dono do arquivo esteja *online*.
- Otimizar consumo de banda distribuído durante a cópia do arquivo.

1.3 ESTRUTURA DA DISSERTAÇÃO

O Capítulo 2 apresenta os conceitos básicos sobre sistemas distribuídos, *backup* distribuído, conceito de grupo e replicação em sistemas distribuídos. Capítulo 3 relata o estado da arte em relação a *backup* P2P, como é realizado o armazenamento remoto de dados, o processo de restauração do *backup* remoto, problemas relacionados a confiabilidade e disponibilidade do dado armazenado remotamente. No capítulo 4 é apresentada nossa hipótese para um modelo de *backup* P2P, nossa estratégia e suas propriedades. São apresentados os resultados das simulações realizadas no capítulo 5. O capítulo 6 apresenta as conclusões sobre a pesquisa realizada.

2 CONCEITOS BÁSICOS

Neste capítulo são apresentados os conceitos de sistemas distribuídos, mais especificamente os conceitos de eleição de líder, arquitetura de rede P2P, simuladores P2P, armazenamento remoto de arquivos , sistema de arquivo distribuído e *backup* em ambiente P2P.

2.1 ELEIÇÃO DE LÍDER EM SISTEMAS DISTRIBUÍDOS

Um sistema distribuído pode ser definido como um conjunto de componentes de *hardware* e *software* interligados por uma rede com capacidade de comunicação entre os processos tendo capacidade para coordenar suas ações apenas enviando mensagens entre si. Cada nodo pode ser visto como um processo autônomo que possui uma identificação única no sistema, um *Globally Unique Identifier* (GUID). Normalmente, cada mensagem trocada entre os nodos contem a informação do GUID do emissor. Em algumas atividades distribuídas é necessário escolher um processo em particular para coordenar as ações dos demais processos, este algoritmo é denominado algoritmo de eleição de líder (COULOURIS; DOLLIMORE, 2005). Na literatura são apresentados alguns algoritmos para eleger um líder, dentre os mais conhecidos estão o anel físico (CHANG; ROBERTS, 1979) e o *Bully* (GARCIA-MOLINA, 1982).

O algoritmo em anel físico (CHANG; ROBERTS, 1979), é usado quando os processos estão dispostos em um anel físico. Cada processo conhece apenas o seu vizinho da direita e da esquerda. O objetivo do algoritmo é eleger um líder e assegurar que todos os subordinados sejam informados sobre o novo líder.

O algoritmo de *Bully* (GARCIA-MOLINA, 1982) também é usado para eleger um líder entre um conjunto de processos com a vantagem de não necessitar de uma topologia de rede em anel, permitindo assim, que cada processo mande mensagem para qualquer outro. O objetivo do algoritmo é eleger um líder e assegurar que todos os subordinados sejam informados sobre o novo líder.

As propriedades comuns entre os algoritmos de eleição de líder são:

- Em um dado momento, apenas um processo é eleito como sendo o líder.
- Um processo pode estar em um dos seguintes estados: líder, subordinado ou em eleição. Todos os processor subordinados sabem quem é o

seu líder.

- A comunicação entre os processos pode ser bidirecional ou unidirecional.
- O processo pode ou não saber a quantidade de nodos presentes no sistema.

Normalmente, a decisão de qual processo vai ser o líder é baseada em uma variável distribuída que é enviada para cada um dos processos. O processo que tiver o maior valor para a variável torna-se o líder. Inicialmente cada processo inicializa a variável que representa o líder, com o seu identificador. No final deste protocolo, a variável terá o maior identificador que representará o líder do grupo. Basicamente, quando o processo está na fase de eleição, cada mensagem recebida tem o identificador do processo que está tentando se eleger. O processo receptor extrai o valor do identificador que está contido na mensagem e compara com o seu identificador. Se o valor recebido for maior do que o valor armazenado no processo, o valor recebido é gravado localmente e a mensagem é encaminhada adiante pois existe pelo menos um processo com o identificador maior. Caso contrário, a mensagem recebida é descartada pois o identificador do processo é maior que o identificador recebido e uma mensagem com o seu identificador já foi enviada ao grupo.

2.2 ARQUITETURA PEER-TO-PEER

A arquitetura de rede P2P têm como objetivo suportar serviços e aplicativos distribuídos, permitindo o compartilhamento de dados e recursos computacionais disponíveis nos computadores pessoais e nas estações de trabalho. Esta arquitetura possui escalabilidade global, pode ter mecanismos de anonimato e mecanismos para garantir a segurança dos dados disponibilizados na rede, nodos podem entrar e sair da rede sem aviso prévio, nodos podem compartilhar recursos, nodos são auto-organizáveis não necessitando de um coordenador. Os nodos que participam do sistema formam uma rede sobreposta (*overlay*) onde os enlaces de rede representam os canais de comunicação entre os nodos (SCHOLLMEIER, 2001), (BANDARA; JAYASUMANA, 2012) e (SHEN *et al.*, 2009). As arquiteturas de redes P2P podem ser classificadas em: centralizadas, descentralizadas ou híbrida.

Arquitetura centralizadas utilizam um servidor central para gerenciar o controle de acesso e busca de recursos. Na arquitetura descentralizada,

cada nodo tem o controle de acesso e busca por um recurso. O problema da arquitetura centralizada é sua vulnerabilidade a ataques que podem tornar indisponível o sistema inteiro, esse problema não acontece nas arquiteturas descentralizadas. Arquiteturas híbridas, possuem nodos com papel diferenciado dos outros membros da rede, denominados *supernodos*. Os *supernodos* podem indexar os recursos compartilhados, aumentando o desempenho e eficiência na localização de recursos sem prejudicar a rede com inundação (*flooding*) de mensagens. O *Kazaa*(KAZAA, 2012), *BitTorrent*(BITTORRENT, 2012) e *Skype* (SKYPE, 2012) são exemplos de sistemas P2P que utilizam arquitetura híbrida. Nós consideramos neste texto a equivalência entre as palavras: nodo, nó e *peer*.

Todos os nodos que participam da rede P2P, na teoria, possuem as mesmas capacidades funcionais. Eles podem assumir três papeis simultaneamente: clientes, servidores e roteadores. Quando um nodo solicita algum recurso, ele se comporta como um cliente. Quando o nodo disponibiliza o recurso está se comportando como servidor. O papel de roteador acontece quando o nodo recebe uma solicitação de um recurso, mas o recurso não está disponível no nodo, então o nodo encaminha a solicitação para seus vizinhos até encontrar o recurso ou a até a mensagem ser descartada.

A localização de um recurso em uma arquitetura P2P é um serviço que merece muita atenção do projetista, pois dependendo da técnica usada pode tornar a comunicação distribuída ineficiente. Basicamente existe duas técnicas de localização de um recurso: inundação (*flooding*) e *Distributed Hash Table* (DHT).

A técnica de localização de um recurso por inundação pode gerar muito tráfego de mensagem, a busca por um recurso é lenta e não tem garantia de sucesso. Para evitar que as mensagens se propaguem indefinidamente é implementado o mecanismo de *Time to Live* (TTL), isto é, um tempo de vida para as mensagens (SCHOLLMEIER, 2001).

A busca por inundação pode ser executada como uma busca em largura ou em profundidade. Na busca em largura, o nodo requisitante envia a mensagem para todos os seus vizinhos, que por sua vez enviam para seus vizinhos até que o recurso seja encontrado ou que o TTL seja atingido. Normalmente o TTL é incrementado uma unidade a cada salto da mensagem para um novo nodo.

A figura 4 apresenta uma busca por um recurso usando a técnica de inundação com TTL. A cada salto da mensagem de um nodo para outro é incrementado o TTL, quando o TTL atinge o valor máximo, no exemplo TTL = 2, a requisição não é mais replicada para os vizinhos. Como pode

ser observado, os nodos em amarelo não receberam a mensagem, pois o TTL atingiu o valor máximo.

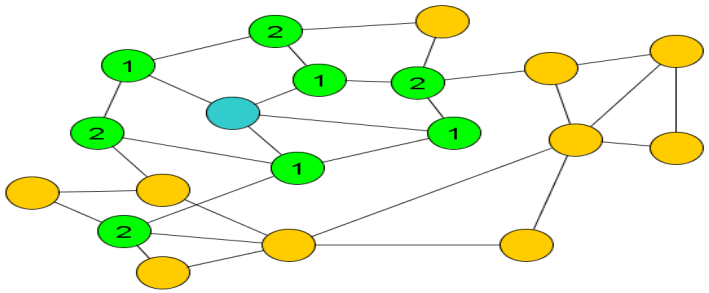

Figura 4: Busca em rede P2P usando TTL = 2. Nodo em azul requisita recurso, os nodos que recebem a solicitação estão com cor verde. Nodos em amarelo não recebem a solicitação do recurso.

A técnica de localização de um recurso usando DHT consegue localizar com eficiência logarítmica um recurso na rede sem necessitar inundar a rede com mensagens (NAGAO; SHUDO, 2011).

A principal função de uma DHT é oferecer um serviço de pesquisa (*lookup*) baseado em pares ordenados de (chave, valor). As principais propriedades de uma DHT são: totalmente descentralizada, alta escalabilidade e tolerância a entrada e saída de *peers*. Alguns exemplos de protocolos que implementam uma rede estruturada podem ser encontrados em: Chord (STOICA *et al.*, 2001), Pastry (ROWSTRON; DRUSCHEL, 2001), Kademlia (MAYMOUNKOV; MAZIèRES, 2002), entre outros.

2.2.1 Simuladores Peer-to-Peer

Um simulador é projetado para simular o comportamento de um sistema replicando os mesmos comportamentos e fenômenos que acontecem em um sistema real (RAUNAK *et al.*, 2011). Existem diversos simuladores disponíveis, tanto comerciais como *open source*. A literatura atual apresenta diversos simuladores para arquitetura de rede P2P.

O P2PSim (MONTRESOR; JELASITY, 2009) é baseado em eventos discretos e atua sobre rede estruturada. Este simulador é escrito em C++ e tem suporte para os protocolos de roteamento e pesquisa (*lookup*) (Chord, Accordion, Koorde, Kelips, Tapestry e Kademlia), efetuando o mapeamento de uma

chave em um nodo. Este simulador tem o objetivo de facilitar e comparar os protocolos P2P.

OverSim (BAUMGART *et al.*, 2007) é um simulador baseado em eventos discretos, permite a programação usando o paradigma *Publish-Subscribe*, onde o nodo se registra para receber eventos específicos (*subscribe*) que são publicados (*publish*) por outros nodos. Este simulador suporta os diversos protocolos: Chord, Pastry, Bamboo, Koorde, Broose, Kademlia, GIA, NICE, NTree, Quon, Vast entre outros. É permitido configurar a topologia de rede, a largura de banda consumida, o percentual de perda de pacotes, o tempo médio de atraso de pacote, além de diversas outras estatísticas de envio e recebimento de dados. O OverSim é escrito em na linguagem C++.

Diversos trabalhos foram publicados usando o simulador PeerSim (MONTRESOR; JELASITY, 2009). Este simulador foi escrito levando em consideração requisitos de escalabilidade para simular milhões de nodos. O PeerSim é *open source*, sendo escrito na linguagem Java. Duas arquiteturas de simulação são disponibilizadas, uma baseada em eventos e outra baseada em ciclos. A arquitetura baseada em ciclos permite uma maior escalabilidade por ignorar detalhes da camada de transporte. A arquitetura baseada em eventos é menos eficiente que o motor de ciclo, porém é mais realista.

A principal *Application Programming Interface* (API) do PeerSim é definida através das seguintes interfaces: $Protocol$, $CDProtocol$, $EDProtocol$, $Transport$, $Control$ (tabela 1). A interface Protocol é a interface base para criação de qualquer tipo de protocolo, sendo definido o método clone que é utilizado para criar uma instância de um protocolo. As interfaces $CDProtocol$, $EDProtocol$, $Transport$ estendem a interface Protocol para acrescentar comportamentos específicos de cada protocolo. O $CDProtocol$ define um protocolo baseado em ciclos com intervalos definidos. Durante a execução de um experimento, o protocolo consegue tratar todos os nodos que participam da simulação, ou seja, a cada laço (loop) da simulação, o protocolo recebe o nodo e um identificador do protocolo através da chamada ao método $nextCicle(node, protocolId)$, este método deve ser implementado para modificar o experimento. A interface $EDProtocol$ permite que os experimentos sejam realizados através de um modelo dirigido por eventos. Um evento é qualquer tipo de objeto (mensagem) que é enviado de um nodo a para um nodo b. Para enviar uma mensagem para outro nodo é necessário usar o protocolo de transporte que é materializado pela classe *Transport*, chamando o método $send(nodeSrc, nodeDest, message, protocolId)$. Para o nodo realizar o tratamento da mensagem que foi enviada via o protocolo de transporte, é necessário implementar o método $processEvent(node, protocolId, objet)$ do

protocolo *EDProtocol*. O PeerSim, apesar de ser concebido como um simulador genérico não tem suporte para realizar experimentos que necessitem de eventos que acontecem em um dado tempo futuro (eventos temporizados), portanto, é difícil avançar ou retroceder no tempo do simulador para analisar os fenômenos que acontecem ao longo do tempo simulado.

Tabela 1: Principais APIs do PeerSim

Método	Interface	Descrição
clone	Protocol	Método chamado para criar uma nova instância de qualquer protocolo
nextCicle	CDProtocol	Método chamado durante cada ciclo da simulação para cada nodo
processEvent	EDProtocol	Método chamado para processar mensagens recebidas de outros nodos
send	Transport	Método chamado para enviar uma mensagem para outro nodo
getLatency	Transport	Método chamado quando necessário saber a latência entre dois nodos
execute	Control	Método chamado antes de cada ciclo da simulação

2.2.2 Sistemas de Arquivos Distribuídos

Um sistema de arquivo distribuído é definido como sendo um sistema que permite aos programas armazenarem e acessarem arquivos remotos exatamente como se fossem locais com as mesmas características que um sistema de arquivo centralizado, a diferença é que os dados são armazenados remotamente com transparência de acesso e localização, heterogeneidade de sistema operacional e tratamento para modificações concorrentes permitindo que os usuários acessem os arquivos a partir de qualquer computador conectado a uma rede (COULOURIS; DOLLIMORE, 2005). Um sistema de arquivo distribuído pode ser divido em três grupos (tabela 2) (GE *et al.*, 2003). O sistema de compartilhamento de arquivos é caracterizado por não ter escrita concorrente (somente o dono pode modificar o arquivo) mas permite leituras simultâneas. Um sistema de *backup* de arquivos permite somente um escritor e um leitor, ambas as operações são executas pelo dono do arquivo. Sistemas que permitem múltiplos escritores e múltiplos leitores são denominados sistemas de

arquivos compartilhado .

Normalmente, os computadores envolvidos no sistema de arquivo distribuído usam técnicas de replicação para sincronizar os dados entre as máquinas. Sistema de arquivo distribuído tem duas funções principais: gerenciar a distribuição dos arquivos compartilhados e organizar a pesquisa de um arquivo no sistema distribuído.

Tabela 2: Tipos de sistema de arquivo distribuído

Tipo	Escrita	Leitura
Sistema de compartilhamento de arquivos	single	multiple
Sistema *backup* de arquivos	single	single
Sistema de arquivo	multiple	multiple

Diversos aplicativos P2P são usados para compartilhar arquivos entre os usuários, como Kazaa (KAZAA, 2012), eMule (EMULE, 2012), eDonkey (EDONKEY, 2012), BitTorrent (BITTORRENT, 2012) (GUO *et al.*, 2005). Entretanto, os arquivos ficam disponíveis somente dentro dos clientes de cada aplicação. Para resolver o problema de disponibilizar um arquivo independente do aplicativo cliente, alguns protocolos foram concebidos como o JXTA (JXTA, 2012), XNap (XNAP, 2012) entre outros.

O Hadoop (BORTHAKUR *et al.*, 2011) é um projeto *Open Source* licenciado pela Apache Software Foundation, um *framework* que permite o processamento distribuído de grandes volumes de dados. Provê um sistema de arquivos distribuídos com alto *throughput* para acessar os dados, é baseado na tecnologia do *Google File System* (GHEMAWAT *et al.*, 2003). O Hadoop é projetado para ter uma alta escalabilidade, podendo ter milhares de máquinas no *cluster* para executar operações ou armazenar dados.

Arquivos também podem ser armazenados na nuvem (*cloud*) (BESSANI *et al.*, 2011) para aumentar a taxa de disponibilidade e segurança. O Amazon Simple Storage Service (Amazon S3) (PALANKAR *et al.*, 2008) é um sistema de armazenamento de dados em servidores com alta escalabilidade, confiabilidade e segurança. O Dropbox (DROPBOX, 2012), Box.net (BOX, 2012), Oosah (CRUNCHBASE, 2012), JungleDisk (JUNGLEDISK, 2012) são serviços de armazenamento de arquivos em nuvem.

Os arquivos armazenados em *cloud* estão armazenados em *clusters* de servidores controlados por empresas particulares.

2.3 BACKUP EM AMBIENTE PEER-TO-PEER

Backup é uma cópia de dados para outro dispositivo de armazenamento com o propósito de restaurar o dado em caso de perda do original. O *backup* distribuído pode ser classificado em três grupos. *Backup* em *cloud* Bessani *et al.* (2011), *backup* em servidor Kubiatowicz *et al.* (2000) e *backup* em arquitetura de redes P2P Landers *et al.* (2004), Dabek *et al.* (2001), Li e Dabek (2006), Tran *et al.* (2008) e Batten *et al.* (2002).

Um recurso que implica diretamente na escalabilidade é o serviço de busca ou pesquisa por um arquivo no ambiente P2P. O serviço de pesquisa no Napster é centralizado (SAROIU *et al.*, 2003), já o sistema de pesquisa no eDonkey é distribuído entre os servidores (YANG *et al.*, 2006). O Kademlia apresenta um sistema de pesquisa cooperativo que é realizado entre os participantes (MAYMOUNKOV; MAZIèRES, 2002).

O *backup* pode ser realizado de maneira incremental ou completa. No *backup* incremental, ao modificar um arquivo já disponibilizado remotamente, é enviado apenas a diferença entre os dados do arquivo. Para restaurar o *backup* é necessário buscar o arquivo original e aplicar todas as consecutivas modificações (na mesma ordem) que o arquivo sofreu.

O *backup* completo sempre envia o arquivo novamente para os nodos remotos. Para restaurar a última versão do *backup*, basta solicitar o arquivo para o nodo remoto. A fase de restauração do *backup* completo é mais eficiente que o *backup* incremental, por outro lado a fase de replicação / modificação do *backup* é mais eficiente para *backups* que utilizam o conceito incremental.

Sistemas de *backup* distribuído normalmente usam protocolos para detectar anomalias nos dados distribuídos. Detecção de erros, correção de erros, replicação, tolerância a faltas bizantinas são algumas das técnicas usadas para tolerar e detectar faltas (CRISTIAN, 1991).

A integridade e consistência são propriedades que devem ser conservadas em sistemas de *backup* em ambiente P2P. Quando o dado for restaurado deve estar intacto sem nenhuma modificação de qualquer natureza (vírus, *malware*, etc) que não foi realizada pelo seu dono.

A confidencialidade e privacidade do dado armazenado remotamente deve ser preservada, impedindo que outros usuários leiam ou acessem o conteúdo de um arquivo que está na rede de *backup* P2P, *i.e.*, somente o dono do arquivo tem privilégios de leitura e escrita sob o *backup*.

O objetivo primário do *backup* é assegurar a disponibilidade do dado quando seu dono requisitar restaurá-lo. A disponibilidade deve ser resistente

a falhas maliciosas, ataques de *Denial-of-Service* (DoS) e nodos *offline*.

Outro problema comum em *backup* P2P diz respeito à sinergia de espaço disponibilizado para realizar o *backup* de outros nodos e o espaço que o sistema disponibiliza para realizar o *backup*. A troca de espaços deve ser justa, se um nodo disponibiliza 5 Mb de espaço para *backup*, ele deveria ter pelo menos 5 Mb de espaço no sistema para fazer seu *backup*.

2.4 ACORDO EM SISTEMAS DISTRIBUÍDOS

Segundo (CHARRON-BOST, 2001) um dos problemas que constitui a base fundamental dos sistemas distribuídos é o acordo. A dificuldade do acordo distribuído está em garantir que os envolvidos no sistema consigam chegar satisfatoriamente em um acordo comum para realizar com êxito uma computação. O acordo pode ser simples ou complexo, indo desde uma simples troca de informações entre dois ou mais processos até uma sincronização total dos estados dos processos. O resultado final do protocolo é que todos devem atingir uma mesma resposta ou acordo para a realização de uma tarefa.

O acordo e seu respectivo problema vem sendo alvo de estudos há mais de trinta anos, tanto em sua praticidade quanto na teoria (LAMPORT *et al.*, 1982).

2.4.1 Consenso em Sistemas Distribuídos

Em sistemas distribuídos, o consenso é em geral o problema de acordo mais conhecido, que consiste em garantir que todos os processos corretos executem a mesma operação tendo o mesmo resultado como saída ou os mesmos utilizem um mesmo valor proposto pelos processos.

A diferença entre consenso e acordo está diretamente relacionada com o início da computação, porém o resultado final é o mesmo, ou seja, todos os processos terem o mesmo valor. O acordo acontece quando somente um processo tem o valor inicial da computação e utiliza técnicas para que os demais processos tenham ciência deste valor. Já no consenso todos os processos tem um valor inicial, é executada uma abordagem democrática para decidir o valor resultante da computação em questão, o resultado final é que todos os processos corretos decidem de forma irrevogável e unânime sobre o valor da computação.

Quando trabalhamos com sistemas tolerantes a faltas, precisamos de-

finir qual o modelo de faltas e os tipos de falhas que o sistema tolera, bem como as principais entidades que interagem no decorrer do tempo.

Em sistemas distribuídos, um processo é um modulo que está distribuído na rede de computadores podendo ter capacidade de executar computações, não tendo memória compartilhada. Uma aplicação distribuída permite que os processos interajam entre si trocando mensagens através de canais de comunicação. Um sistema distribuído pode ser definido como um conjunto finito Π de $n > 1$ processos, $\Pi = \{p1, p2, ..., pn\}$. Todos os processos são considerados corretos ou ativos, exceto para os processos que não executam o comportamento especificado, neste caso eles são considerados processos faltosos. No conjunto de n processos, supomos que um total de f são faltosos ($0 \leq f \leq n$) e que n - f são corretos (GREVE, 2005).

Uma falta, em um sistema distribuído, pode ser definida como algo que impede o processo de executar o comportamento especificado, podendo ser originada através de uma falha. Os principais tipos de falhas de processos encontrados na literatura são: falhas por parada, por omissão, de desempenho e arbitrária (LAPRIE, 1995) e (VERISSIMO; RODRIGUES, 2001). A falha por parada é caracterizada quando todas as ações que um processo pode executar, param de funcionar permanentemente levando o processo a entrar em um estado de colapso. A falha por omissão pode levar o processo a não executar determinadas ações omitindo o envio ou recebimento de mensagens, esta falha também pode ser classificada com um tipo especial de falha de desempenho. Na falha de desempenho o processo não executa a ação no tempo determinado. Entretanto, todas estas falhas podem ser vistas como falhas arbitrárias. Nestas, os processos podem executar ações fora da sua especificação passando a incorporar um comportamento imprevisível. Este comportamento pode ter sido originado a partir de um *bug* no código ou até mesmo pela alteração realizada no código por um *hacker*. As falhas por parada, por omissão e de desempenho são ditas benignas, porque os processos comportam-se seguindo as ações especificadas até o momento em que tornam-se faltosos. A falha arbitrária por definição admite a existência de um comportamento maligno (GREVE, 2005).

Os modelos de falhas de canais de comunicação entre os processos devem definir se o canal pode corromper a mensagem e se é permitido mensagens duplicadas. Também deve ser estabelecido restrições em relação à perda, ordenação e atraso na recepção das mensagens. Em um canal confiável, uma mensagem enviada por um processo p_i a um processo p_j será recebida por p_j, não havendo perda de mensagens e com garantia que o conteúdo enviado por p_i é o mesmo quando recebido pelo processo p_j. Se o canal é *First In, First*

Out (FIFO), entre p_i e p_j, a ordenação das mensagens é a mesma da emissão (GREVE, 2005). Neste trabalho, os algoritmos apresentados assumem que não há perda de mensagens, mas a mensagem pode ter seu conteúdo corrompido durante o processo de envio até a mensagem chegar no processo receptor.

O consenso é um denominador comum para os problemas de acordo, podendo ser definido informalmente da seguinte maneira: cada processo p_i propõe um valor v_i, apesar da existência de falhas, todos os processos corretos decidem por um valor v único, escolhido dentre aqueles que foram propostos por todos os processos (GREVE, 2005). Segundo (CHARRON-BOST, 2001) o problema do consenso tem as seguintes propriedades:

- **Acordo**: Se um processo correto decide por um valor v, então todos os processos corretos devem decidir pelo mesmo valor v.
- **Validade**: Se um processo decide por um valor $v \in V$, então v foi previamente proposto por algum processo.
- **Integridade**: Para cada valor $v \in V$ que foi proposto pelo sistema, cada processo decide no máximo uma vez.
- **Terminação**: Todos os processos corretos computam o mesmo valor de decisão.

Um dos protocolos conhecidos na literatura que resolve as propriedades citadas é o protocolo de acordo Bizantino.

2.4.2 Acordo Bizantino

Acordo bizantino é uma generalização do consenso, onde é assumido que existe a possibilidade de falta. Uma falta impede o processo de executar o comportamento especificado. O objetivo básico do acordo bizantino é chegar a um consenso entre os processos levando em consideração que podem existir processos faltosos durante o processamento distribuído. Cada processo deve tomar uma decisão baseada nos valores recebidos de outros processos e todos os processos não defeituosos devem chegar a mesma decisão. Uma solução para o consenso distribuído é considerar, se a maioria dos participantes é correta, então é possível chegar ao consenso, porém esta premissa não é válida se admitirmos a possibilidade de faltas bizantinas. Para que ocorra um consenso ou acordo em um ambiente bizantino são necessários pelo menos $3f + 1$ participantes corretos, onde f representa o número máximo de processos faltosos (LAMPORT *et al.*, 1982).

A origem do nome faltas bizantinas deve-se ao problema clássico dos generais bizantinos apresentado em (LAMPORT *et al.*, 1982): eles provaram que se existir 3 processos e houver pelo menos 1 traidor não existe acordo.

Para os generais bizantinos, para ganhar a guerra todos os participantes precisam executar a mesma ação coordenadamente. O ambiente é composto por dois tipos de participantes: generais e tenentes. Existem duas ações que os participantes executam: atacar ou retirar. A ordem de qual ação executar sempre é provida do general, quando um tenente recebe uma mensagem do general ele réplica esta mensagem para o outro tenente com o objetivo de validar a mensagem recebida.

A figura 5 mostra a impossibilidade de realizar um acordo bizantino sendo que um deles é faltoso. A figura 5a mostra o general como sendo o traidor, ele envia mensagens aleatórias para cada processo. Uma mensagem de retirar e outra de atacar, portanto os tenentes não conseguem tomar uma decisão sobre qual ação realizar. A figura 5b mostra o tenente como sendo o traidor. Neste caso sempre que o tenente traidor recebe uma mensagem ele envia para o outro tenente o comando inverso, exemplo: se receber a mensagem para atacar envia para o outro tenente uma mensagem de retirar.

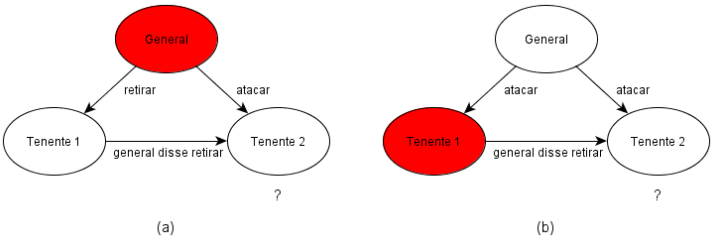

Figura 5: Impossibilidade de Acordo Bizantino. A circunferência em vermelho representa o processo malicioso

A figura 6 mostra um cenário que atende a premissa $3f + 1$ participantes com um processo faltoso. Neste caso o general é o processo faltoso e tem como comportamento enviar uma mensagem de atacar para um tenente e retirar para outro tenente assim consecutivamente. Como os tenentes trocam mensagem para validar e coordenar a ação do general, os tenentes 1, 2 e 3 decidem executar a ação de atacar, sendo esse o comportamento esperado.

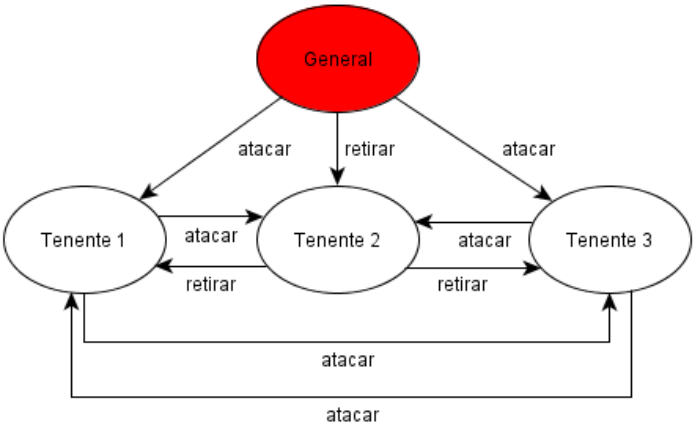

Figura 6: Acordo Bizantino $3f + 1$ com $f = 1$. A circunferência em vermelho representa o processo malicioso

3 TRABALHOS RELACIONADOS

Neste capítulo nós apresentamos o estado da arte sobre *backup* em ambiente P2P, evidenciando como os dados são armazenados remotamente e como eles são restaurados. A pesquisa sobre *backup* em sistemas P2P continua evoluindo, buscando por soluções para os problemas de armazenamento, confiabilidade (restaurar o dado intacto / integro) e disponibilidade para restauração: Defrance *et al.* (2011), Toka *et al.* (2012), Toka *et al.* (2010), Pàmies-Juárez *et al.* (2010), Duarte *et al.* (2010).

Uma das limitações mais significativas das aplicações de *backup online* é o custo: largura de banda e espaço de armazenamento são caros, causando uma oferta limitada de espaço em disco para *backup* gratuito. Para baratear os custos, algumas abordagens exploram espaço em disco e largura de banda presentes na "borda" da Internet, isto é, nas máquinas de usuários finais, a fim de executar aplicações de *backup* que garantem disponibilidade, mas com baixa latência na rede. Entretanto, obter armazenamento confiável explorando recursos não confiáveis é muito difícil ou praticamente impossível Toka *et al.* (2010).

Sistemas de *backup* P2P são fortemente afetados por falhas permanentes e temporárias de *peers*. Por isso, um dos principais requisitos é garantir que os dados armazenados sempre podem ser recuperados. Para alcançar esse requisito, sistemas de *backup* P2P introduzem grandes quantidades de redundância para manter a disponibilidade dos dados perto de 100%, introduzindo altos custos de armazenamento. Uma das questões pesquisadas é como diminuir a redundância mantendo a disponibilidade em taxas aceitáveis Pàmies-Juárez *et al.* (2010).

A disponibilidade representa a quantidade de tempo que um dado copiado está presente e *online* para ser possível sua recuperação. A medição das taxas de disponibilidade representa um desafio nos sistemas dinâmicos de *backup* P2P (Toka *et al.* (2012), Defrance *et al.* (2011), Duarte *et al.* (2010)).

No artigo Toka *et al.* (2010) os autores propõem o uso combinado ou uma abordagem híbrida de *backup* usando P2P e *datacenters* (como as soluções de computação em nuvem). Dois são os elementos principais da proposta: a escolha do local para copiar os dados e a alocação de largura de banda. É feita uma simulação para medir o impacto no desempenho causado pelo tempo necessário para completar operações de *backup* e recuperação. A simulação também calcula custos relacionados com o armazenamento dos dados. Os resultados de simulação correspondem a uma coleção de 10 execuções, cobrindo um período de 97 dias. Um sistema com 870 *peers*

é simulado, cada um dos *peers* requisitando um *backup* de 10 GB de dados. Cada objeto do *backup* é dividido em 32 fragmentos antes de aplicar redundância, resultando em um tamanho de fragmento de 320 MB. Nesse trabalho assume-se que cada *peer* tem 50 GB de espaço em disco para realizar *backup* dos *peers* remotos. Os experimentos do artigo demonstram que, com o uso de políticas de alocação de largura de banda e espaço de armazenamento usado temporariamente em provedores de nuvem, a abordagem proposta pode atingir bom desempenho com custo razoável.

No trabalho de Defrance *et al.* (2011) é usada a abordagem híbrida de *backup* usando servidores e redes P2P. Os computadores domésticos que ficam conectados por longos períodos em estado *online*, de acordo com estatísticas, são usados em conjunto com gateways que fazem o papel de *buffers*. Essa estrutura armazena dados na estrutura P2P e auxilia a prover a disponibilidade dos dados ao longo do tempo. Os dados são copiados para servidores da rede P2P que usam tabelas DHT para armazenar informações sobre os dados para que os dados possam ser localizados. A estratégia de redundância usada é a técnica conhecida como *erasure code* - os dados são divididos em um número n de fragmentos e cada fragmento tem um número *k* de réplicas. O trabalho realiza simulações para avaliar a proposta e usa os seguintes parâmetros: largura de banda da rede, pedidos de *backup* e de recuperação, disponibilidade do *peer* e política de redundância. Foi construído um simulador próprio baseado em ciclos e as simulações foram conduzidas com um tempo de avanço de ciclo definido em 5 minutos, totalizando 50 simulações. Algumas métricas importantes foram avaliadas: tempo de *backup*, tempo de recuperação dos dados, disponibilidade dos *peers*, número e quantidade dos fragmentos, tamanho da rede em número de nós e quantidade de dados a serem copiados e restaurados.

O artigo Toka *et al.* (2012) propõe um mecanismo de gerenciamento de redundância para aplicações de *backup* P2P. Normalmente, nos sistemas de *backup* P2P a redundância de dados existe para prover a disponibilidade necessária no momento de restauração, já que os *peers* podem entrar e sair da rede em qualquer instante e também podem sofrer falhas. Políticas de redundância devem ser bem definidas para evitar exageros no número de cópias que garantem a disponibilidade dos dados. O uso de técnicas de *erasure code* normalmente causa um exagero no número de réplicas tendo um alto custo. Os autores mencionam que, já que em um sistema de *backup* os dados são lidos somente durante os processos de recuperação, o gerenciamento de redundância deve tratar a durabilidade dos dados ao invés de tentar deixar cada pedaço de informação sempre disponível. A proposta define uma forma de

cálculo do tempo de durabilidade dos dados baseado no número de atrasos por detecção de falhas e no tempo de recuperação dos dados. Assim, pela proposta, cada *peer* define uma quantia de redundância suficiente para conter os efeitos da saída de *peers* da rede, mas preserva tempos aceitáveis de recuperação de dados. O trabalho usa simulação para coletar métricas de desempenho, comparando a estratégia proposta com a estratégia de redundância padrão de *erasure code*. Durante a simulação, cada *peer* tem 10GB de dados para fazer *backup* e tem 50 GB de espaço de armazenamento. O tamanho do fragmento é definido como 160 MB implicando no mínimo em 64 fragmentos necessários para recuperação. Os gráficos obtidos durante a simulação mostram taxas de redundância de acordo com a fórmula proposta, tempo de *backup* e tempo de recuperação. Nas conclusões, o artigo cita que os experimentos mostraram que em um cenário realista, a redundância que tem o objetivo de tratar a durabilidade dos dados pode representar menos da metade do que é necessário quando se quer garantir a disponibilidade.

O artigo Duarte *et al.* (2010) apresenta uma proposta de algoritmo que pode ser usado em redes P2P para quantificar e garantir a disponibilidade para recuperar arquivos em qualquer instante. O algoritmo proposto melhora a confiabilidade das redes P2P sendo possível a descoberta de valores percentuais sobre a disponibilidade de um *peer*. Experimentos de simulação foram feitos para medir tempos de *backup* de arquivos com tamanhos entre 3KB e 120 MB. Diferentes tamanhos de blocos na divisão dos arquivos foram usados ao longo da simulação.

O trabalho de Pàmies-Juárez *et al.* (2010) propõe um framework analítico para prever os efeitos causados pela redução da redundância, medindo tempos de recuperação de dados variando circunstâncias de falhas na rede. Foram realizadas simulações usando conjuntos de dados reais de redes P2P como *Skype* e *eMule*. Os resultados demonstram que mantendo a taxa de disponibilidade em 80%, mais da metade das recuperações dos dados não experimenta variações nos tempos de recuperação. Entretanto, reduzindo a taxa de disponibilidade de 100% para 80%, a redundância pode ser reduzida até 35% e assim também os custos associados.

Nenhum dos trabalhos citados nos parágrafos anteriores (Defrance *et al.* (2011), Toka *et al.* (2012), Toka *et al.* (2010), Pàmies-Juárez *et al.* (2010), Duarte *et al.* (2010)) usa algoritmo tolerante a faltas bizantinas.

Em (COURTES *et al.*, 2007) é apresentado um estudo sobre *backup* cooperativo em dispositivos móveis. A estratégia de replicação é pré-definida usando *erasure code (n, k)*, onde *n* representa o número de fragmentos do arquivo e *k* o número de réplicas que receberam a cópia. Os valores de *k*

e *n* são configurados *a priori* baseado na confiabilidade e confidencialidade que o usuário deseja para seus dados. Apenas um fragmento é replicado para cada nodo correndo o risco de não replicar todos os fragmentos no final do processo. Quando o dispositivo móvel não consegue acessar a Internet para realizar o *backup*, ele forma uma rede *ad hoc* com os dispositivos móveis ao ser redor (vizinhos) e envia para cada um deles apenas um fragmento do arquivo. Quando o dispositivo móvel consegue acesso a Internet, ele envia o fragmento do dado para uma infra-estrutura de *backup* na Internet. O dado enviado tem a informação do seu respectivo dono. Eles assumem que o serviço de *backup* na Internet é confiável, então somente após enviar o dado para a Internet é que ele pode ser considerado confiável e poderá ser restaurado mais tarde. Dispositivos que não conseguem acesso a Internet são tratados como faltas. A técnica de *erasure code* é usada para reconstruir o arquivo completamente mesmo que algum bloco esteja faltando.

Em resumo, o trabalho (COURTES *et al.*, 2007), quando um dispositivo móvel não consegue acesso a Internet para realizar o *backup*, ele monta uma rede *ad hoc* com os seus vizinhos, fragmenta o arquivo em diversos blocos usando a técnica *erasure code* e envia para cada dispositivo móvel um bloco. Quando um dispositivo móvel consegue acesso a Internet, ele envia os dados que esta armazenando para o serviço de *backup* na Internet. O bloco enviado tem a informação de quem é o seu dono. O dado é considerado seguro somente após a realização do *backup* no serviço de Internet.

Em (RANGANATHAN *et al.*, 2002) é proposto um mecanismo para armazenar dados em ambientes P2P visando manter alta disponibilidade. Replicar o dado em diversos nodos tende a ter uma alta taxa de disponibilidade. Cada nodo está autorizado para criar réplicas para os arquivos que armazena. A decisão de criar uma nova réplica é descentralizada. Um nodo decide onde replicar o arquivo baseado em um modelo que compara o custo e o beneficio para criar uma réplica de um arquivo em particular levando em consideração a localização do nodo onde o arquivo será copiado. A decisão é tomada baseada em uma série de parâmetros, o tempo médio que um nodo fica *online*, o tempo de transferência do arquivo entre os nodos, o custo de armazenamento físico do arquivo. Nodos que não querem armazenar dados retornam infinito para o custo de armazenamento físico. A disponibilidade do arquivo depende da taxa de falhas dos nodos e do serviço de localização de réplicas. A precisão do serviço de localização determina o percentual de arquivos acessíveis. O serviço de localização retorna um conjunto de nodos localizados em diferentes áreas geográficas. Também é apresentada uma fórmula para calcular o número de réplicas necessárias para obter um arquivo

levando em consideração um limiar de disponibilidade. A fórmula considera a disponibilidade do nodo estar *online*, a precisão do serviço de localização de réplicas. Para mais detalhes sobre a fórmula, consultar Ranganathan *et al.* (2002). Depois que um nodo conhece o número de réplicas necessário para armazenar o arquivo, é usado o serviço de localização de réplicas para verificar se o dado precisa ser copiado para outros nodos. Cada nodo consegue calcular o custo para criar uma réplica levando em consideração o tempo para transferir o dado e o custo do armazenamento remoto. O custo de armazenamento remoto é calculado usando os nodos origem, destino e nodo do usuário. O nodo origem contém o dado que será copiado para o nodo destino, o nodo do usuário é definido como sendo o nodo de onde partirá a maioria das requisições sobre o dado.

Em sistemas P2P nodos podem entrar e sair sem aviso prévio prejudicando a disponibilidade para restaurar o dado. Para evitar esse problema é proposto uma checagem periódica do número de réplicas existentes com o número de réplicas requisitadas. Quando o número de réplicas requisitadas é menor do que o número de réplicas disponíveis, cada réplica pode automaticamente criar novas réplicas. Portanto, diversos nodos podem criar réplicas para o mesmo arquivo consumindo banda desnecessariamente. Para amenizar este problema, é proposto que a cada passo da replicação, apenas 25% dos nodos disponíveis no sistema sejam capazes de criar réplicas simultaneamente.

No trabalho (LI; YUN, 2010) é proposto um algoritmo para *backup* em rede P2P estrutura usando a técnica *erasure code* levando em consideração segurança e persistência do dado. Quando uma parte do dado fica indisponível, uma nova cópia é replicada no cluster para garantir tolerância a faltas. O sistema é composto por três partes, clientes, nodos de *backup* e nodos supervisores. Os clientes são responsáveis por codificar o dado e enviar para os servidores. O grupo de nodos supervisores, é responsável por executar o protocolo de acordo bizantino, fazer o roteamento de mensagens e disponibilizar o *checksum* do dado quando solicitado por um cliente durante o processo de restauração. O grupo tem a informação global da topologia da rede e conhece todos os dados que estão sendo armazenado em cada *peer*. Os nodos de *backup* estão logicamente ligados através de uma rede *Chord* formando um cluster e sua principal responsabilidade é armazenar o dado codificado. Para mais informações sobre redes *Chord* favor consultar (STOICA *et al.*, 2001).

Para realizar o *backup* no trabalho (LI; YUN, 2010), o cliente divide o dado em diversas partes, codifica cada parte usando uma matriz de coeficientes aleatórios. Após a codificação, é iniciado a construção de uma sequência

de *hash* para identificar cada parte codificada. Por fim, é executado o protocolo de acordo bizantino para assegurar que todos estão armazenando o mesmo dado e é registrado a meta informação do dado nos nodos supervisores.

Para ler o dado remoto no trabalho (LI; YUN, 2010), o cliente solicita o identificador e *checksum* com o grupo de supervisores. O grupo de supervisores responde a mensagem informando os parâmetros solicitados e a localização do dado no cluster. O cliente solicita o dado para $F + 1$, calcula o *hash* do dado recebido comparando com o *hash* do dado disponibilizado pelo grupo de supervisores, se o *hash* for igual o dado é aceito. Quando o grupo de supervisores detecta uma falha, é iniciado o protocolo de recuperação. O protocolo de recuperação consiste em localizar todas as partes do dado que ainda estão disponíveis, recriar partes que não estão presentes baseado no *erasure coding*, e redistribuí-las para o cluster.

O *pStore* Batten *et al.* (2002) foi concebido para permitir que usuários consigam armazenar e restaurar *backups* em uma rede distribuída com *peers* não confiáveis. O arquivo é replicado para assegurar a confiabilidade do *backup*. O *pStore* mantém a versão do arquivo replicado através de *snapshots* armazenados para permitir que o usuário possa restaure uma versão especifica do arquivo. O arquivo é particionado em blocos e replicado na rede. A integridade é verificada através de uma comparação do *hash* do bloco recebido com o *hash* armazenado no metadados que também foi distribuído na rede P2P.

Nos trabalhos Li e Dabek (2006) e Tran *et al.* (2008) o *backup* é armazenado somente em nodos confiáveis usando uma rede social. Neste tipo de aplicação, a presença de nodos maliciosos é minimizada devido aos laços de confiança estabelecidos no mundo real.

O Tran *et al.* (2008) apresenta uma abordagem P2P para fazer *backup* cooperativo usando nodos confiáveis. Para isso o sistema faz uso da lista de amigos e colegas de uma rede de relacionamento social que o usuário participa. Os nodos são confiáveis porque é o próprio usuário que estabelece um contrato com seus amigos/colegas que ele considera confiáveis de acordo com seu relacionamento social. Cada nodo armazena seu *backup* em um conjunto de nodos escolhidos pelo usuário dono do dado, onde cada nodo escolhido representa um colega ou amigo providos de uma rede de relacionamentos, com isso é oferecida uma solução não técnica para o problema de disponibilidade e danificação do serviço. Os dados são armazenados cifrados, para isso cada membro da população de usuário possui um nome, uma chave privada usada para cifrar e decifrar os dados. Cada nodo é administrado pelo seu usuário e

tem duas responsabilidades: fazer o *backup* local e ajudar os outros a fazer o *backup*. A capacidade de armazenamento é limitada por dois tipos de recurso: largura de banda e espaço disponível no disco rígido. Largura de banda é um recurso limitado principalmente quando é necessário re-copiar os dados para os *backups*. Para prevenir que um nodo armazene mais dados que ele possa manter é proposto que cada nodo calcule a capacidade disponível baseado na largura de banda de *upload* e na quantidade de espaço que ele usa para armazenar seus dados no sistema. Quando a largura de banda é o recurso limitado o sistema propõe uma troca por espaço em disco, para isso é usado o esquema *XOR*, onde é duplicando a quantidade de dados armazenados em um nodo ao custo de transferir duas vezes os dados durante a fase de restauração do dado original. O dono do arquivo, periodicamente checa se o *backup* remoto está intacto (saúde) requisitando um *hash* de um dado aleatoriamente. Ele verifica localmente se o *hash* retornado é igual ao *hash* calculado, se o resultado da comparação for falso, significa que o dado está danificado, portando a cópia é apagada remotamente e criada uma nova. *Friendstore* não tem suporte para atualização de *backup*. Quando um nodo não responde o comando de checagem por um período, é enviado o conteúdo daquela réplica para outra réplica. O timeout usado é de 200 horas para distinguir se um nodo falhou.

O *OceanStore* Kubiatowicz *et al.* (2000) provê mecanismos que permitem o acesso contínuo de um dado armazenado no sistema usando uma infra-estrutura de servidores não confiáveis. O dado é protegido (cifrado) e usando técnicas de *erasure code*, o dado é replicado para os servidores. A localização do dado na rede é um recurso transparente. A mensagem de localização contém apenas uma meta-informação (GUID). O dado pode ser cacheado em alguma localização aumentando assim o desempenho do sistema de localização. O processo de sincronização entre os dispositivos de armazenamento e o processo de transparência de localização de um dado permite que novos dispositivos de armazenamento entrem e saiam do sistema sem a necessidade de haver uma nova configuração do sistema. A união dos dois processos citados permite que o usuário acesse o dado de diferente localização geográfica. A infra-estrutura proposta permite que sejam enviadas apenas as mudanças que ocorreram no dado, ou seja, suporta o conceito de *update*. Os servidores devem estar distribuídos geograficamente e podem cachear dados perto dos clientes. A informação pode migrar livremente de um servidor para outro. Os dois objetivos principais do *OceanStore* são: conceber uma uma infra-estrutura de servidores para armazenamento de dados e suporte para migração de dados de forma transparente do cliente. Toda a informação privada que entra no sistema deve estar cifrada para evitar que

servidores maliciosos conheçam a informação. É assumido que os servidores estão trabalhando corretamente. Quando um cliente envia um dado, a primeira camada de servidores executa o acordo bizantino enquanto a segunda camada de servidores começa a espalhar o dado de forma epidêmica. Após a primeira camada de servidores finalizar o acordo bizantino, o dado é enviado via *multicast* para todas as réplicas secundárias.

O *PeerStore* Landers *et al.* (2004) gerencia o metadados e o *backup* de forma independente uma da outra. O metadado é gerenciado através do uso de uma DHT garantindo um bom desempenho para localizar o arquivo. O *backup* é baseado em um esquema de troca simétrica, ou seja, para realizar o *backup* remoto o nodo deve armazenar localmente um *backup* de outro nodo.

Em resumo, os trabalhos relacionados apresentados neste capítulo podem ser classificados como: trata consumo de banda; é um ambiente P2P puro (não necessita usar recursos extras como: *datacenter*, *pool* de servidores); faz verificação de confiabilidade remota, ou seja, não necessita que o dono do arquivo verifique a integridade do dado remoto; tem tratamento para nodos maliciosos (tabela 3).

Tabela 3: Comparação entre os trabalhos relacionados

Trabalho	**Consumo Banda**	**P2P Puro**	**Confiabilidade Remota**	**Nodo Malicioso**
Toka 2012	Não	Sim	Não	Não
Defrance et al, 2011	Sim	Não	Não	Não
Toka 2010	Sim	Não	Não	Não
Li; Yun 2010	Não	Sim	Não	Sim
Tran et al, 2008	Não	Sim	Não	Sim
Courtes et al, 2007	Sim	Não	Não	Não
Ranganathan et al. 2002	Sim	Sim	Não	Não
Batten et al, 2002	Não	Sim	Não	Sim
kubiatowicz et al, 2000	Sim	Não	Não	Sim

4 DAREIN: MODELO DE SISTEMA E ALGORITMO DE BACKUP P2P

Neste capítulo nós apresentamos a hipótese de pesquisa, as definições para o modelo de sistema, como o *backup* é replicado / distribuído em uma rede P2P, como o *backup* é restaurado, como nós resolvemos o problema de disponibilidade do *backup* sem necessitar copiar para toda a rede P2P, como nós asseguramos a integridade do *backup* sem necessitar da presença do dono do arquivo. Com base nos tópicos citados neste parágrafo, nós apresentamos algoritmo para *backup* P2P.

O algoritmo foi desenvolvido, tendo como objetivos: alta disponibilidade para restaurar o *backup* distribuído em uma rede P2P sem necessitar copiar o arquivo para todos os membros da rede, otimizar os recursos de *hardware* que são usados durante todo o processo de armazenamento remoto, restaurar o *backup* remoto com confiabilidade assegurando a integridade do dado distribuído sem a necessidade do dono do arquivo fazer checagens para verificar a integridade, tolerar faltas bizantinas durante todas as fases do *backup*.

4.1 HIPÓTESE

Nossa hipótese é baseada em um comportamento viral. Os vírus são parasitas que necessitam de um organismo vivo para se reproduzir. O vírus da gripe existe há muitos anos, sua transmissão ocorre através do contato entre os animais. O constante contato entre os animais é uma das principais causas da sobrevivência do vírus por vários anos. Nosso algoritmo é baseado no comportamento epidêmico (DEMERS *et al.*, 1987) controlado por um número adequado de réplicas ativas.

Em Biologia, um vírus pode ser definido como um agente (parasita intracelular) que fica hospedado em um hospedeiro, tem capacidade de autoreplicação e pode causar doenças em animais e vegetais. Um vírus contamina outro hospedeiro através de algum tipo de contato. A estrutura molecular de um vírus é chamada de partícula viral e pode ser formada por ácido nucléico, proteínas, lipídios e carboidratos. O material genético do vírus é representado pelo seu *Deoxyribonucleic acid* (DNA) ou *Ribonucleic acid* (RNA) (ACHESON, 2006).

Na Informática, um vírus pode ser definido como um software projetado para causar algum tipo de problema em dados de usuários, o vírus é

normalmente considerado um software malicioso. O vírus é construído com capacidade de fazer cópias de si mesmo e tentar infectar outros computadores com as quais tenha contato.

Para restaurar o *backup*, é preciso ter mecanismos que garantam que o dado está disponível a maior parte do tempo sem necessitar replicar para toda população. Então, em nossa proposta, nós modelamos um agente com o comportamento de uma doença epidêmica (como um vírus) para realizar auto-replicação do *backup* quando perceber que ainda não atingiu o número adequado de cópias ativas *online*. Para evitar que toda a população seja contaminada através do comportamento epidêmico, o agente realiza uma computação distribuída controlando assim o tamanho do conjunto de *peers* infectados. O agente possui um bloco de *bytes* originados a partir do pedido de *backup* remoto de um arquivo.

4.2 MODELO DE SISTEMA

O modelo de sistema consiste em um conjunto infinito de *peers* $P = \{p1, p2, ...\}$, onde cada *peer* pode estar ligado a um ou vários *peers* formando uma topologia de rede P2P. Quando um *peer* entra ou sai da rede, a topologia, bem como a lista de rotas (caminhos) que cada *peer* conhece são alteradas.

Um *peer* pode ser classificado como:

- **Correto**: O *peer* executa todas as funções conforme foi especificado. Todos os *peers* corretos possuem as mesmas responsabilidades e podem executar as seguintes tarefas: eleger um líder e replicar o *backup* para outros *peers* usando os códigos que estão presentes nele.

- **Malicioso**: Um *peer* malicioso pode corromper o arquivo que está armazenando, negar o serviço de *backup* e até mesmo enviar mensagens erradas para tentar danificar outro *backup*. Nós assumimos que faltas que impedem o *peer* de executar o comportamento especificado pode acontecer em um *peer* malicioso independente da ocorrência nos demais.

A arquitetura de rede P2P é puramente descentralizada e não-estruturada. A comunicação entre os nodos acontece somente via a topologia de rede sobreposta usando a técnica de inundação. Cada *peer* pode se comportar de três maneiras diferentes. Como um cliente quando ele inicia o ciclo de vida do *backup*. Como um servidor quando armazena o *backup* e

como roteador quando tem que rotear as mensagens do algoritmo de um *peer* para outro que não está diretamente conectado. Como o foco da pesquisa não é desempenho, por simplicidade de desenvolvimento foi escolhido fazer comunicação por inundação, entretanto para aumentar o desempenho e diminuir o tráfego de mensagens entre os nodos, poderia ser adicionada uma DHT para otimizar a localização do *backup*.

Cada *peer* contém um identificador único que pertence somente a um nó durante toda a simulação e um conjunto de *peers* parceiros. Um *peer* é considerado um parceiro se e somente se tiver uma ligação entre eles.

Também assumimos que o dono do arquivo cifrou o arquivo antes de enviá-lo para o *backup* remoto, sendo ele o responsável por assegurar a privacidade e confidencialidade da chave usada para cifrar.

Nós assumimos que as mensagens trocadas são assíncronas e que um *peer* é capaz de identificar quem enviou a mensagem através da assinatura digital que está contida na mesma. Também é possível identificar o recebimento das mensagens enviadas. A topologia de rede não é conhecida diretamente pelos *peers*, cada *peer* conhece apenas seus parceiros. Quando um *peer* envia uma mensagem para um membro que não é seu parceiro, esta mensagem é roteada através de seus parceiros e dos parceiros do parceiros até encontrar um caminho para o *peer* de destino. O caminho existe somente se tiver uma ou mais arestas que interligam um *peer* a outro formando uma rota do *peer* de origem até o *peer* de destino.

Cada *peer* é concebido como um organismo que pode estar suscetível ou não a contaminação. A contaminação acontece via os contatos (parceiros) que cada *peer* possui. Para um *peer* estar suscetível à contaminação é necessário que ele atenda os seguintes requisitos:

- **Espaço em disco**: o *peer* deve ter espaço livre em disco maior ou igual ao tamanho do bloco de *bytes*.

- **Rota**: durante um ciclo de execução a topologia da rede não se modifica, devendo existir pelo menos um caminho (rota) onde cada *peer* pode se comunicar com qualquer outro *peer* infectado. Caso um *peer* não consiga se comunicar com outro, durante o processamento distribuído, este *peer* deixará de pertencer ao conjunto de *peers* infectados.

Em nosso modelo, o agente representa uma entidade que pode ser replicada para qualquer *peer* através do comportamento epidêmico, desde que o *peer* seja suscetível à contaminação. O agente conhece todos os *peers* que

estão infectados, tecnicamente, o agente é um objeto replicado em cada *peer* infectado por ele. O agente também tem um material genético que deu origem a sua existência, possui um conjunto de *bytes* que representa um bloco de dados de um arquivo e também consegue criar uma partícula viral para ser usada no consenso bizantino. A partícula viral deste comportamento epidêmico é considerada benigna, ela não altera nenhum dado do *backup* distribuído e é usada exclusivamente durante o consenso bizantino.

Analogamente aos vírus biológicos, o material genético serve para identificar / classificar os tipos de vírus. Uma pessoa pode estar com gripe sendo contaminado pelo vírus da gripe influenza *A* ou influenza *B* ou influenza *C*. Em nosso modelo, quando um arquivo é particionado em vários blocos, é gerado um material genético único que representa a identificação do arquivo. Para cada bloco, é criado um agente e injetado nele o bloco e uma cópia do material genético. É através do material genético que nós garantimos que um *peer* não armazene mais de 1 bloco do mesmo arquivo.

4.2.1 Replicando e Restaurando o Backup

Antes de iniciar o *backup* remoto, nós particionamos o arquivo em vários blocos. Cada bloco que será distribuído na rede é concebido como um agente e o comportamento epidêmico controlado é usado para evitar que o dado contido no agente se replique para toda a população. Nós definimos uma regra para encontrar o número adequado de *peers* infectados. Este número é dado pela seguinte função:

$f(x)$: tal que x representa o número de *peers* que estão infectados e $x \geq 3f + 1$, onde f representa o número máximo de faltosos. Neste trabalho foi considerado $f = 1$.

A figura 7 mostra o processo que transforma um arquivo em um ou mais agentes. Quando o usuário deseja realizar um *backup* de um arquivo, a entidade "Criador de vírus" solicitado à entidade "Gerador de material genético" que crie um material genético (GUID) único para identificar o arquivo no sistema distribuído. O arquivo compactado que foi gerado é particionado em vários blocos de *bytes*. Cada bloco de *bytes* unido com o material genético é transformado em um agente que será distribuído na rede P2P (figura 7).

Depois que os agentes são criados, o algoritmo contamina uma parte da população, como pode ser observado através da figura 8. Inicialmente foi enviado um arquivo para o "Criador de vírus" que gerou 2 agentes (amarelo e

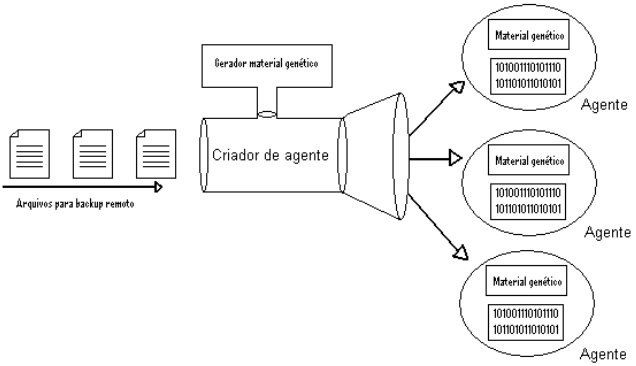

Figura 7: Mecanismo usado para transformar um arquivo em um agente

azul). Como esses dois agentes tiveram sua origem no mesmo arquivo, para seguir a definição apresentada acima, um *peer* não pode armazenar ao mesmo tempo o agente amarelo e azul.

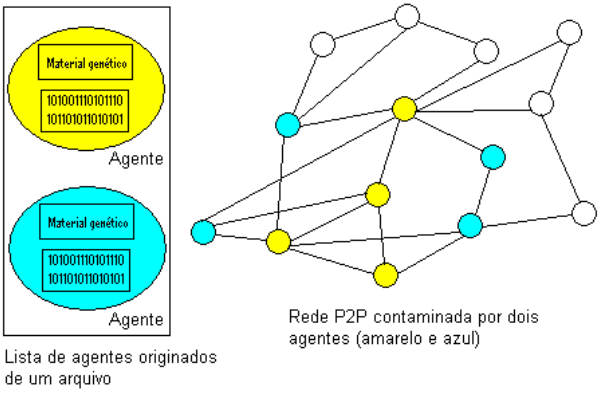

Figura 8: Contaminação com 2 agentes originados do mesmo arquivo

Todas as decisões do agente são tomadas de forma distribuída. As principais propriedades usadas pelo agente durante a fase de contaminação são:

- **Contaminação rápida**: o agente escolhe o *peer* que tem o menor tempo de transmissão dos dados pela rede somado com o menor tempo para persistir e ler o dado do disco rígido. Para isso é considerado o tempo de *download* de um *peer* suscetível a contaminação, o tempo que um *peer* contaminado leva para fazer o *upload* do dado, o tempo que um *peer* contaminado leva para ler o dado do disco rígido, o tempo que um *peer* suscetível gasta para persistir o dado no disco rígido. Também é considerado que um *peer* pode estar fazendo *downloads* e / ou *uploads* concorrentes como em um ambiente real. A figura 9 apresenta todas as propriedades envolvidas durante a escolha do *peer*.

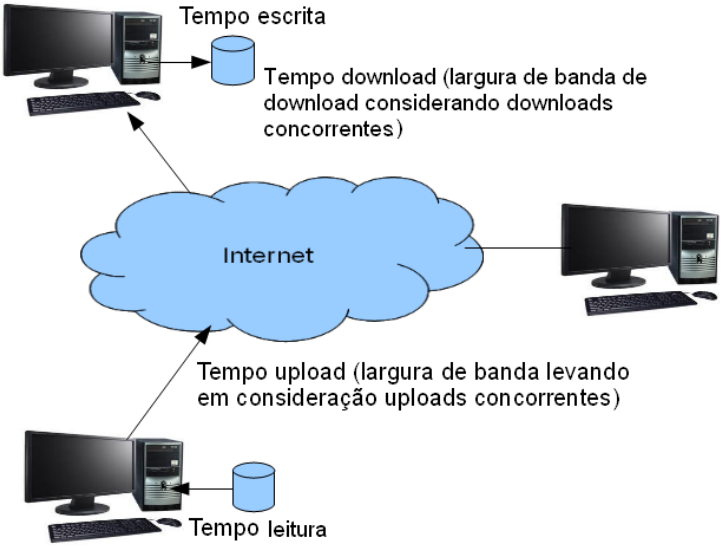

Figura 9: Propriedades usadas durante a contaminação rápida

- **Eleição do melhor peer**: o agente, através do *peer*, elege o melhor *peer* para ser infectado levando em consideração a propriedade *Contaminação rápida*. A eleição acontece completamente distribuída, sendo executada por todos os *peers* que estão infectados pelo agente. Em resumo o comportamento deste algoritmo é eleger o melhor *peer* suscetível à contaminação. A figura 10 mostra como a eleição acontece do ponto de vista de cada *peer* e depois globalmente. A figura 10(a) apresenta uma rede P2P onde os *peers* infectados estão representados

com a cor amarela e os *peers* em azul estão suscetíveis a contaminação. A figura 10(b) mostra que cada *peer* executou uma eleição para encontrar qual é o melhor candidato para ser infectado do seu ponto de vista. As arestas (linhas) estão com um número que representa o custo para transferir o agente para o outro membro baseado na propriedade *Contaminação rápida*. Na figura 10(c) temos uma tabela que é compartilhada entre todos os membros infectados pelo agente, nesta tabela é representado o melhor *peer* com seu respectivo custo para cada um dos membros infectados pelo agente, como resultado, a figura 10d) mostra o melhor *peer* eleito de maneira distribuída, no nosso caso o *peer* com o identificador 3 vai contaminar o *peer* com identificador 10.

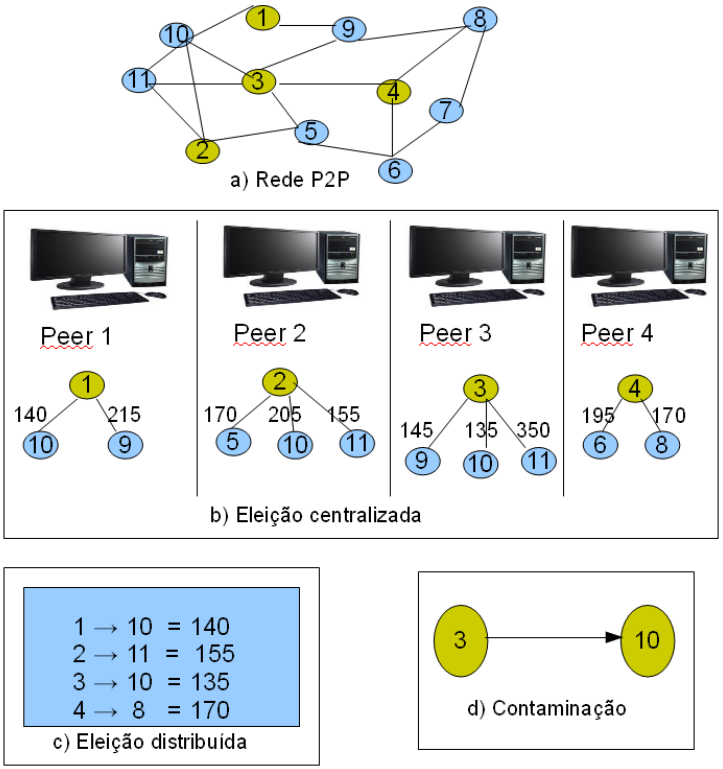

Figura 10: Eleição distribuída para escolher o melhor peer para ser contaminado

O *peer* deve estar suscetível à contaminação para ser infectado por um

agente. Para o *peer* estar suscetível à um agente, ele deve ter as seguintes propriedades:

- O *peer* não foi infectado por nenhum agente que tenha o mesmo material genético, ou seja, não está armazenando nenhum bloco de *bytes* que foi originado do mesmo arquivo.
- Exista espaço livre no disco rígido maior ou igual ao tamanho do bloco de dados que está contido no agente.

A figura 11 mostra o processo que transforma um conjunto de agentes com seus respectivos blocos de *bytes* em um arquivo. Todos os agentes que tem o mesmo material genético são reunidos e processados para restaurar o arquivo distribuído remotamente. Para restaurar um arquivo, é realizada uma busca na rede P2P para coletar todos os agentes que tem o mesmo material genético, onde mais tarde serão unidos todos os blocos de *bytes* para criar novamente o arquivo original.

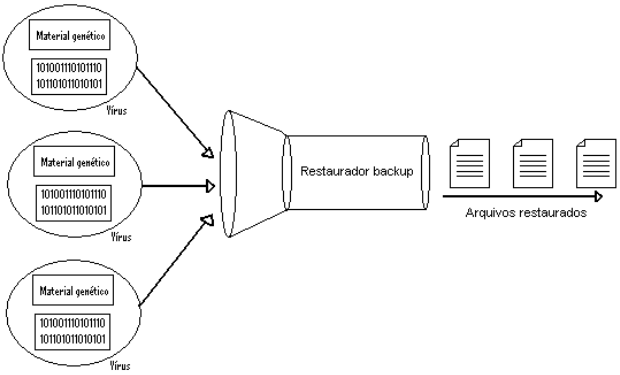

Figura 11: Mecanismo usado para transformar um conjunto de agentes com seus blocos de *bytes* em um arquivo

Em resumo, para realizar o *backup* de um arquivo, inicialmente o arquivo é particionado em vários blocos de *bytes*, para cada bloco é criado um agente. O agente tem por função primária replicar o bloco em um conjunto de *peers* escolhendo (a cada rodada do algoritmo) o melhor *peer* para enviar o dado. Para restaurar o *backup* é realizada uma busca para encontrar todos os agentes que estão armazenando um bloco de *bytes* que foi originado

inicialmente do arquivo que se deseja restaurar. Para realizar esta busca, é necessário apenas conhecer o material genético que foi criado quando o arquivo foi particionado em blocos. Por fim, os blocos são unidos para restaurar o arquivo.

4.2.2 Disponibilidade e Integridade do Backup

Para assegurar a disponibilidade de restauração do *backup* sem necessitar copiar o arquivo para toda população de *peers*, nós restringimos a propriedade de infecções que um *peer* pode ter, evitando assim que mais de um bloco de dados do mesmo arquivo seja armazenado no mesmo *peer*.

A fase de sobrevivência / disponibilidade do dado é executada em vários passos. Primeiro é escolhido, usando a técnica de eleição em Anel, um novo líder para cada agente. O líder é um *peer* que pertence ao conjunto de nodos infectados pelo agente. O segundo passo é assegurar a confiabilidade (integridade) do dado distribuído usando a técnica de acordo bizantino entre todos os *peers* pertencentes ao conjunto de nodos infectados. O terceiro passo é continuar a contaminação (replicação) para outros *peers* que ainda não foram infectados. A disponibilidade para restaurar o *backup* é assegurada com o comportamento epidêmico controlado do agente, evitando assim que toda a população seja contaminada.

A integridade do *backup* distribuído é assegurada através do protocolo bizantino, sendo necessários pelo menos $3f + 1$ *peers* armazenando o *backup*, onde f é o número de *peers* faltosos. Um *peer* faltoso pode danificar o *backup* que está armazenando, enviar mensagens erradas ou até mesmo negar o serviço de *backup*. Para otimizar o envio de mensagens em nosso algoritmo, o protocolo bizantino somente pode ser iniciado pelo líder do grupo. Por questões de simplicidade de implementação, o algoritmo utiliza a técnica de eleição em Anel (CHANG; ROBERTS, 1979) para eleger o líder de um grupo de agentes distribuídos na rede.

4.2.2.1 Eleição do novo líder

Dentre os algoritmos de eleição de um líder disponíveis na literatura, nós usamos o algoritmo em Anel proposto em (CHANG; ROBERTS, 1979). Os *peers* conseguem formar um anel circular virtual com todos os membros infectados pelo agente, onde cada *peer* conhece o seu vizinho da direita e da

esquerda. O objetivo do algoritmo é eleger o *peer* que tem o maior identificador e notificar todos os membros sobre o resultado da eleição.

Durante a execução do algoritmo, cada *peer* envia a mensagem *leaderElectionProtocol* levando o seu identificador como parâmetro. Quando um *peer* recebe a mensagem *leaderElectionProtocol* é verificado se o identificador que está na mensagem é o mesmo identificador do *peer*, caso positivo a eleição terminou e o *peer* foi eleito. O *peer* eleito tem o dever de notificar todos os *peers* sobre o resultado da eleição enviando a mensagem *leaderElectionResult* com o seu idenficador. Se um *peer* recebe uma mensagem *leaderElectionProtocol* com um identificador menor do que o maior identificador que ele conhece, a mensagem é finalizada. Caso contrário, significa que a eleição não finalizou e que o *peer* deve enviar a mensagem com o seu identificador para o vizinho no anel virtual. Para assegurar a terminação do algoritmo de eleição, nós assumimos que durante a execução da eleição de um líder a topologia de rede não é alterada.

4.2.2.2 Protocolo Bizantino para Backup Distribuído

Cada *peer* infectado pelo agente executa o protocolo bizantino para assegurar que o dado distribuído está intacto. Nossa implementação do protocolo bizantino distingue dois papéis, o líder e os membros. Somente o líder tem permissão para iniciar o protocolo.

O protocolo é executado sobre o conjunto de *peers* infectados $I = \{ p1, p2, ..., pn\}$ tal que $| I | \geq 3f + 1$. Também deve existir pelo menos um caminho (rota) em que um *peer* pode mandar uma mensagem para outro membro pertencente ao conjunto de infectados sem ter que alterar a topologia de rede, ou seja, todos os membros conseguem mandar mensagens para cada membro do conjunto infectado, inclusive para o líder.

As mensagens enviadas são denominadas partícula viral. Cada *peer* infectado solicita que o agente crie uma partícula viral que será enviada para cada membro do conjunto de *peers* infectados. Uma partícula viral é uma sequência de *bytes* que foram gerados a partir do bloco de dados com posição de inicio e fim gerados aleatoriamente. É calculado um *hash* baseado nos *bytes* que será usado para validar o acordo distribuído. A partícula viral também possui uma identificação para informar qual foi o agente que a originou.

Quando um *peer* recebe a mensagem *byzantinePropose* ele guarda a partícula viral recebida para ser validada em um momento futuro e se a mensagem veio do líder ela é disseminada para todos os parceiros infectados,

caso contrário essa mensagem é morta.

4.3 ALGORITMO DE BACKUP P2P

Nosso algoritmo foi batizado com o nome *Darein*, este nome não tem nenhuma relação com a solução proposta para *backup* distribuído. Os detalhes sobre o funcionamento do algoritmo *Darein* são descritos nos próximos parágrafos.

O algoritmo *Darein* é segmentado em duas grandes fases: a fase de replicação do arquivo e a fase de sobrevivência do dado na rede. A fase de replicação do arquivo é executada em várias rodadas, sendo iniciada com o dono particionando o arquivo em vários blocos de *bytes*, cada bloco forma um agente que será replicado. Cada agente tem um conjunto de nodos infectados por ele. Este conjunto é inicializado com o *peer* dono do arquivo. O dono do arquivo deve replicar o bloco para outros nodos, para isto ele escolhe um *peer* dentre o seu conjunto de *peers* parceiros e replica o bloco. O *peer* escolhido é adicionado ao conjunto de nodos infectados pelo agente. O próximo passo é escolher outro *peer* parceiro para replicar o bloco. Cada *peer* pertencente ao conjunto de infectados escolhe um candidato dentre a lista de seus *peers* parceiros, após cada um escolher o seu candidato, é escolhido um único *peer* que receberá o *backup*, este passo é repetido até que o bloco de *bytes* seja replicado em pelo menos $3f + 1$ nodos.

Os *peers* são classificados de acordo com o seu estado interno: servidor, cliente ou roteador. Quando um *peer* envia uma mensagem para outro que não está em seu conjunto de *peers* parceiros, é necessário rotear a mensagem até que ela chegue ao seu destinatário. Quando um nodo solicita um *backup*, ele se comporta como um cliente e quando o *peer* armazena o *backup* se comporta como servidor. As chamadas remotas que podem ser roteadas entre os nodos são:

- *prepareLeaderElection* indica que o protocolo de eleição de um novo líder será iniciado, portanto o *peer* deve se preparar para a eleição;
- *leaderElectionProtocol* é chamado quando o *peer* está executando o protocolo de eleição de um novo líder;
- *leaderElectionResult* é executado quando o novo líder foi escolhido por todos os *peers*;

- *byzantinePropose* é invocado durante a execução do protocolo bizantino para propor um valor;
- *infect* é chamado durante a fase de contaminação para infectar um *peer* com um determinado agente;
- *getAgentData* é chamado durante a fase de restauração do *backup* levando como parâmetro o material genético do agente.

Para facilitar o entendimento, o algoritmo de eleição do melhor *peer* é apresentado em duas partes. O *algoritmo*1 recebe um agente como entrada e retorna o melhor candidato suscetível a contaminação levando em consideração todos os possíveis melhores candidatos que foram eleitos por cada *peer*, nós percorremos a lista de *peers* infectados pelo agente e escolhemos o melhor nodo para ser contaminado. O *algoritmo 2* mostra como é feita a busca $A*$ (RUSSELL; NORVIG, 2010) com a heurística do menor caminho para escolha do *peer* a ser infectado levando em consideração a propriedade *Contaminação rápida*.

Durante a eleição, cada *peer* infectado pelo agente executa o algoritmo de busca $A*$ para eleger o seu melhor candidato (levando em consideração o custo para infectar cada *peer*) suscetível a contaminação. O *peer* eleito pode ou não ser um parceiro. Depois que cada *peer* infectado elegeu seu melhor candidato, o algoritmo retorna o melhor candidato dentre os melhores.

O ciclo viral (Algoritmo 3) é executado por cada um dos *peers* infectados pelo agente em intervalos de tempos definidos pela aplicação. Os principais comportamentos executados durante o ciclo viral são: assegurar a confiabilidade do bloco de dados distribuído e assegurar a sobrevivência do agente.

Para garantir que o dado está intacto, ou seja, que ele não foi alterado por nenhum *peer*, é necessário que os envolvidos executem o protocolo bizantino. Cada *peer* infectado pelo agente executa o ciclo viral em quatro passos:

1. Enviar a mensagem *prepareLeadElection* para todos os *peers* informando que vai acontecer uma eleição de um novo líder;
2. Enviar a mensagem *leaderElection* iniciando assim o procedimento de eleição de um novo líder;
3. Iniciar o protocolo bizantino localmente e remotamente enviando a mensagem *byzantinePropose* para todos os *peers* que fazem parte do conjunto de *peers* infectados pelo agente;

```
   Input: um virus
   Output: o melhor nodo candidato suscetível ou nulo
 1 oMelhor ← null;
 2 foreach pi ∈ virus.listaPeerContaminados do
 3 |   nodo ← Darein.encontrarMelhorPeerLocal(virus, pi) ;
 4 |   if nodo ≠ null then
 5 |   |   custoNodo ← custo acumulado para chegar até o nodo;
 6 |   |   custoMelhor ← custo acumulado para chegar até o
   |   |   melhor;
 7 |   |   if oMelhor is null OR custoNodo < custoMelhor then
 8 |   |   |   oMelhor ← nodo;
 9 |   |   end
10 |   end
11 end
12 return oMelhor
```

Algoritmo 1: Darein: algoritmo de eleição do melhor *peer* para ser infectado executado pelo participante i

4. Verificar localmente se houve um consenso distribuído assegurando que o dado está intacto, caso contrário o *peer* sai da lista de infecções do agente.

Após o término da fase de confiabilidade do dado replicado, o algoritmo executa a fase de sobrevivência tentando infectar um número adequado de *peers*. O *peer* eleito é infectado com o agente passando mais tarde a executar o Algoritmo 3 periodicamente.

O algoritmo 4 apresenta o pseudo-código para contaminar um *peer* levando em consideração o número adequado de réplicas ativas na rede. O número adequado é uma das variáveis que se deseja descobrir através de simulações. Este número é usado para evitar a contaminação em toda a população; esse número nunca pode ser menor do $3f + 1$, onde F representa o número de *peers* maliciosos que o sistema admite sem que o *backup* seja danificado.

```
   Input: um virus e um peer
   Output: um peer ou nulo
 1 fechados ← criar um conjunto de nodos vazio;
 2 abertos ← criar uma fila com prioridade ordenada pelo custo de
   cada nodo;
 3 nodo ← criar nodo com o estado inicial levando em consideração
   o virus e o peer recebidos como parâmetros;
 4 abertos.adicionarFila(nodo);
 5 while abertos is not vazia do
 6     nodo ← abertos.removerHead();
 7     if nodo is objetivo da busca then
 8         return nodo;
 9     end
10     if nodo ∉ fechados then
11         fechados.adicione(nodo);
12         foreach pi ∈ nodo.listaPeerParceiros do
13             tempoTransmissao ← calcular tempo transmissão do
               bloco de bytes que está no agente do nodo até o seu
               parceiro pi;
14             novoNodo ← criar nodo levando o
               tempoTransmissao como custo para gerar o nodo e o
               nodo como seu pai para poder calcular o custo
               acumulado;
15             if novoNodo is descendente novo do nodo then
16                 abertos.adicionar(novoNodo);
17             end
18         end
19     end
20 end
21 return null;
```

Algoritmo 2: Darein.encontrarMelhorPeer: algoritmo executado por um *peer* para encontrar um candidato

```
   Input: um virus
 1 foreach pi ∈ virus.getListaPeersContaminados do
 2 |   enviar mensagem prepareLeadElection(virus) para pi;
 3 end
 4 foreach pi ∈ virus.getListaPeersContaminados do
 5 |   enviar mensagem leaderElection(virus) para pi;
 6 end
 7 particulaViral ← criar uma particula viral levando em
   consideração o bloco de dados;
 8 particulaViral.gerarHash;
 9 foreach pi ∈ virus.getListaPeersContaminados do
10 |   peer pi inicia o protocolo bizantino propondo particulaViral;
11 end
12 foreach pi ∈ virus.getListaPeersContaminados do
13 |   if não entrou em acordo bizantino com pi then
14 |   |   virus.removePeer(pi);
15 |   end
16 end
17 foreach pi ∈ virus.getListaPeersContaminados do
18 |   if pi.tamanhoListaInfectados() < 3f + 6 then
19 |   |   pi.contaminar(virus) // chamando algoritmo 4;
20 |   end
21 end
```

Algoritmo 3: Darein: algoritmo do ciclo de vida viral executado pelo participante i para assegurar confiabilidade e disponibilidade

```
   Input: um virus
 1 Static: tamanhoOtimoInfeccoes: número inteiro;
 2 if virus.getTamanhoListaPeersInfectados ≥
   tamanhoOtimoInfeccoes then
 3 |   return;
 4 end
 5 oMelhor ← virus.eleicaoDoMelhorCandidato();
 6 oMelhor.infectar(virus);
```

Algoritmo 4: Darein.contaminar: Algoritmo para contaminar um peer

5 ASPECTOS DE IMPLEMENTAÇÃO E RESULTADOS DA SIMULAÇÃO

Este capítulo detalha o simulador desenvolvido e os resultados obtidos nos vários cenários de simulação.

O algoritmo *Darein* é uma solução alternativa para *backup* em redes P2P. Nós apresentamos neste capítulo os aspectos relacionados com o algoritmo *Darein*, os diagramas de classes usado para implementar o algoritmo, diagramas de classes usados para construir o simulador e o diagrama de pacotes que contempla os dois tipos de experimentos que simulamos: o experimento de disponibilidade e o experimento de desempenho.

A comunicação entre os *peers* (troca de mensagens) é realizada através de técnicas de chamadas de procedimentos remotos (*Remote Procedure Call* (RPC)) onde é possível executar um procedimento que está implementado em outro *peer* de forma transparente, não se preocupando com o protocolo de transporte (*Transmission Control Protocol* (TCP), *User Datagram Protocol* (UDP)). Através de técnicas de programação sofisticadas, é possível combinar *design patterns* para criar um *proxy* entre o objeto local e o objeto remoto de tal maneira que fique simples para trabalhar com objetos distribuídos sem prejudicar o desempenho e elegância do sistema.

5.1 ASPECTOS DE IMPLEMENTAÇÃO DO ALGORITMO DAREIN

Nós construímos duas aplicações, uma para simular a taxa de disponibilidade e outra para avaliar o desempenho do nosso algoritmo. O diagrama de pacotes que representa estas duas aplicações é mostrado na figura 12. O pacote *benchmark* contém classes para executar o experimento de disponibilidade e desempenho, este pacote importa o pacote do simulador e o pacote *Darein*.

Na figura 13 nós apresentamos o principal diagrama de classes da *Darein*. As principais responsabilidades da classe *DareinAlgorithm* são: realizar o *backup* de um bloco de *bytes* e executar o ciclo de vida viral. O ciclo viral é executado por cada um dos *peers* infectados pelo agente em intervalos de tempos definidos pela simulação. Os principais comportamentos executados durante o ciclo viral são: assegurar a confiabilidade e assegurar a disponibilidade do *backup*. A confiabilidade do bloco de dados é assegurada de forma distribuída através da troca de mensagens realizada durante acordo bizantino, onde o agente cria uma partícula viral com um *hash* gerado a partir

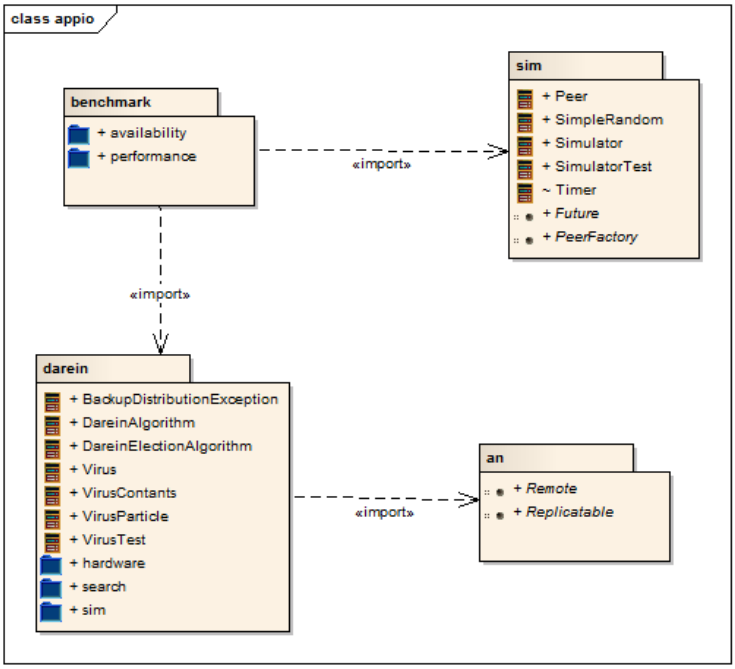

Figura 12: Diagrama de pacote da Darein

de um pedaço do bloco de *bytes*. A partícula viral é enviada para cada *peer* infectado. Cada nodo infectado calcula o *hash* e verifica se o *hash* calculado é igual ao *hash* recebido. A disponibilidade do *backup* é assegurada através do comportamento epidêmico, onde o algoritmo tenta contaminar uma parte da população de *peers*. Ainda na figura 13, foram omitidos os métodos e atributos da classe *DareinPeer* que são apresentados na figura 16. Nos parágrafos seguintes nós vamos detalhar as principais classes apresentadas na figura 13.

A classe *Virus* contém um *array* de *bytes* que representa um bloco (sequência) de *bytes* originados do arquivo que foi realizado o *backup*. O material genético do agente é usado para identificar os agentes que foram criados a partir do mesmo arquivo, ou seja, quando um arquivo é particionado em x blocos, é criado um agente com o mesmo material genético para cada bloco que será replicado na rede. O agente conhece todos os *peers* que ele está contaminando. O agente é um objeto distribuído / replicado na rede e pode executar os seguintes comportamentos:

- contaminar outros *peers* quando o método *contaminate* é invocado.

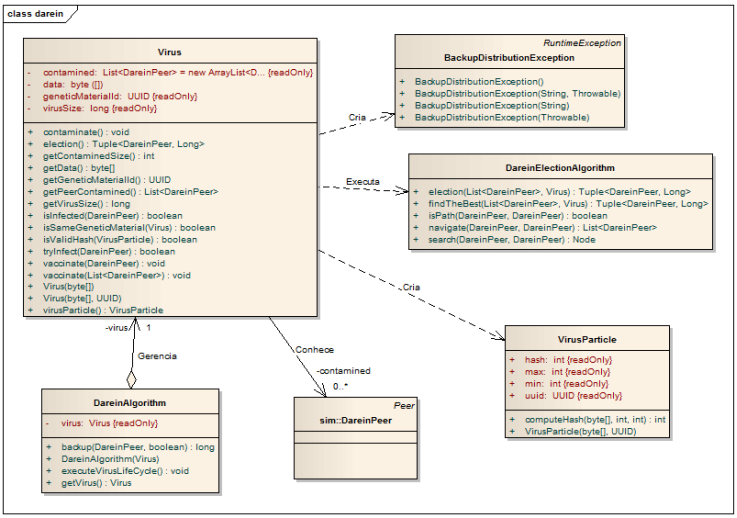

Figura 13: Diagrama de classes da Darein

Este método é executado em três momentos distintos: a) quando o agente identifica que algum de seus *peers* infectados foi desligado. Para isto, todo o grupo de *peers* infectados, enviam e recebem mensagens informando que o *peer* está vivo. b) quando o agente identifica que o tamanho da população de *peers* infectados ainda não atingiu o número adequado de contaminações. c) quando o agente identifica que existe um *peer* malicioso. Como o algoritmo é distribuído, no final da execução, os *peers* que estão *offline* ou são maliciosos são removidos da lista de infecções por cada um dos *peers* infectados.

- descobrir de forma distribuída quem é o melhor *peer* para realizar a contaminação. Este comportamento é executado quando o método *election* é chamado.

- criar (através do método *virusParticle*) e verificar (através do método *isValidHash*) uma partícula viral que é usada durante a fase de confiabilidade / integridade do *backup* distribuído (através da técnica do acordo bizantino).

- infectar um *peer*. Para o *peer* ser infectado ele deve estar suscetível ao agente, ou seja, não estar armazenando nenhum bloco de *bytes* do mesmo arquivo (através do método *isSameGeneticMaterial*) e ter

espaço livre no disco rígido. Por isso o método $tryInfect$ pode ou não ser executado.

Por questões de organização do código e melhor compreensão deste trabalho, nós criamos a classe *DareinElectionAlgorithm* que executa o algoritmo de eleição para escolher o melhor *peer*. Também é responsábilidade desta classe verificar se existe uma rota possível entre um nodo *a* até um nodo *b* usando a rede sobreposta formada pela arquitetura P2P. A principal responsábilidade desta classe é eleger o melhor *peer* suscetível à contaminação segundo as propriedades *Contaminação rápida* e *Eleição do melhor peer* apresentadas no capítulo 4.

Quando o agente cria uma partícula viral que é representada pela classe *VirusParticle*, um objeto é criado contendo: o material genético do agente, dois índices escolhidos aleatoriamente que representam a posição inicial e final de uma sequência de *bytes* que será usada para calcular um *hash* que deve ser validado por cada membro que está infectado.

Para realizar o *backup* remoto, o método *backup* da classe *DareinAlgorithm* deve ser executado. Este método é executado de forma distribuída por todos os *peers* que estão infectados pelo agente até que o agente consiga pelo menos $3f + 1$ réplicas ativas. O comportamento deste método é bem similar ao processo de eleição distribuído para escolher o melhor *peer* para ser infectado (figura 10). A diferença é que a cada replicação do agente, a população de infectados é alterada e a descoberta do melhor *peer* para realizar o *backup* deve ser reiniciada. A figura 14 mostra as fases que são executadas durante o processo de *backup*.

A figura 14(a) mostra a topologia da rede onde o nodo com o identificador 1 deseja realizar um *backup* remoto. A figura 14(b) mostra o processo de eleição do melhor *peer* para receber o *backup*. O melhor *peer* é aquele que tem o menor tempo de *download* e *upload* levando em conta a propriedade *Contaminação rápida*. Para facilitar o entendimento, nós colocamos em cada aresta o custo baseado na propriedade *Contaminação rápida*, neste caso o nodo com o identificador 1 escolhe replicar o *backup* no nodo com o identificador 10 a um custo de 140. Na figura 14(c) a eleição para a escolha do melhor *peer* é distribuída entre os nodos com os identificadores 1 e 10. Cada um desses nodos realiza uma eleição e eles decidem que o nodo com o identificador 10 deve replicar o *backup* para o nodo com o identificador 2 com um custo de 130. Na figura 14(d) todos os nodos que possuem uma cópia do *backup*, ou seja, os nodos 1, 10, 2 executam a eleição de forma distribuída e acordam que o nodo com o identificador 2 deve realizar a cópia do *backup* para o nodo com o identificador 11 com um custo de 200. Como o sistema

estava configurado para $3f + 1$ cópias do *backup* com $F = 1$, é necessário replicar o *backup* para 4 *peers*, portanto a fase de cópia terminou pois existe 4 réplicas do *backup* na rede nos nodos com os identificadores: $1, 10, 2$ e 11.

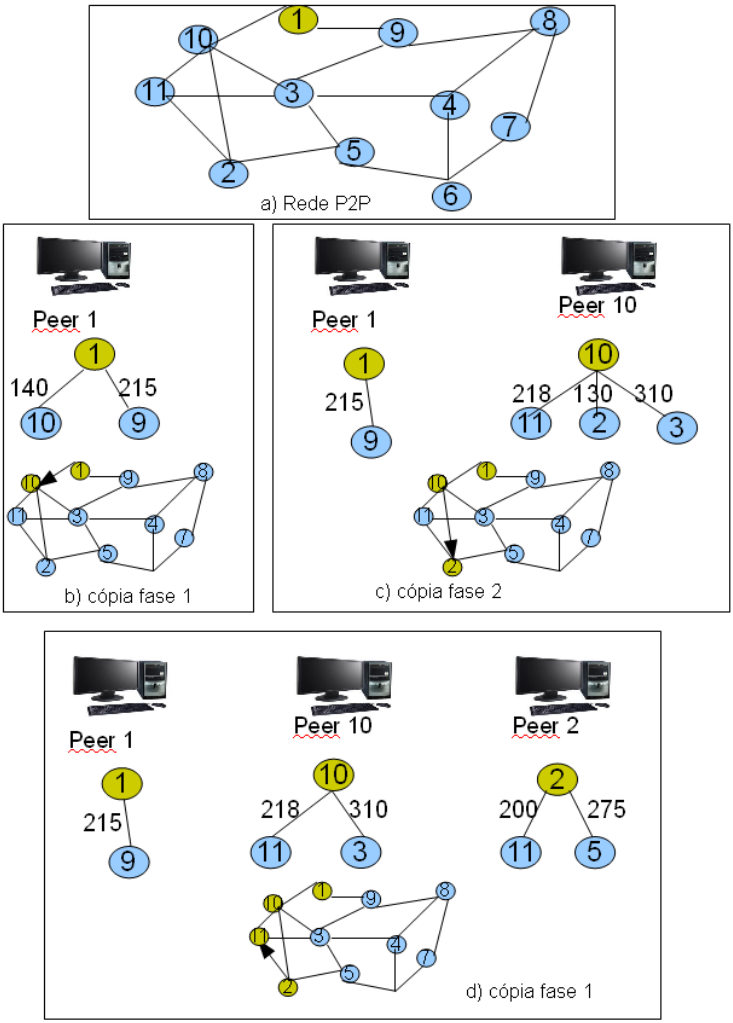

Figura 14: Fases de replicação do Backup na rede P2P com F = 1

5.2 O SIMULADOR SIMPLESIMULATOR

Um simulador é projetado para simular o comportamento de um sistema tendo os mesmos comportamentos e fenômenos que acontecem em um sistema real. Nesta seção detalharemos o simulador desenvolvido denominado *SimpleSimulator*, apresentamos o algoritmo principal e o diagrama de classes do simulador.

Os motivos para o desenvolvimento do *SimpleSimulator* foram: todos os simuladores estudados e apresentados na seção 2.2.1 são de propósito geral, alguns foram escritos em C++ e outros em *Java*. Por questões de simplicidade de implementação, o algoritmo *Darein* foi implementado em *Java*, portanto todos os simuladores que não foram escritos em *Java* foram descartados. Dentre os simuladores estudados, o mais indicado seria usar o PeerSim (MONTRESOR; JELASITY, 2009), mas o mesmo não apresentou características como: agendar eventos para acontecerem em um determinado tempo futuro. Como o PeerSim é *open source*, nós poderíamos ter alterado o seu código, mas antes precisaríamos compreender toda sua arquitetura. Como o objetivo deste trabalho não é entender / documentar um simulador em específico, nós fizemos uma analise do custo / beneficio e chegamos a uma conclusão: precisamos de um simulador onde seja possível agendar eventos para eles serem executados em um determinado tempo futuro, ou seja, a complexidade para criar um simulador com esta característica é baixa em relação a entender toda a complexidade da arquitetura de um simulador de propósito geral.

O simulador foi implementado seguindo os conceitos de orientação a objetos, sendo constituído pelas seguintes classes: *Timer*, *SimpleRandom*, *Simulator*, *Peer*. A classe *Timer* contém um atributo privado do tipo inteiro que representa o tempo (hora) da simulação e métodos para avançar no tempo durante a simulação. A classe *SimpleRandom* é implementada sob o *design pattern* chamado *wrapper*, envolvendo um objeto *Random* do pacote *java.util* do Java. A classe *SimpleRandom* possui métodos para gerar um valor aleatoriamente dentro de uma faixa especificada por um valor mínimo e um valor máximo. A classe *Peer* contém uma lista privada de *peers* que são considerados os parceiros, um identificador único para identificar o *peer* e método para ligar dois *peers*, tornando eles parceiros na rede P2P. A classe *Simulator* possui um conjunto de eventos ordenados pelo tempo (hora), onde o tempo representa o horário em que o evento deve ser executado pelo simulador. Esta classe possui métodos para adicionar um evento, criar novos *peers* e agendar um evento para o fim da simulação. O diagrama de classes pode ser visto na figura 15.

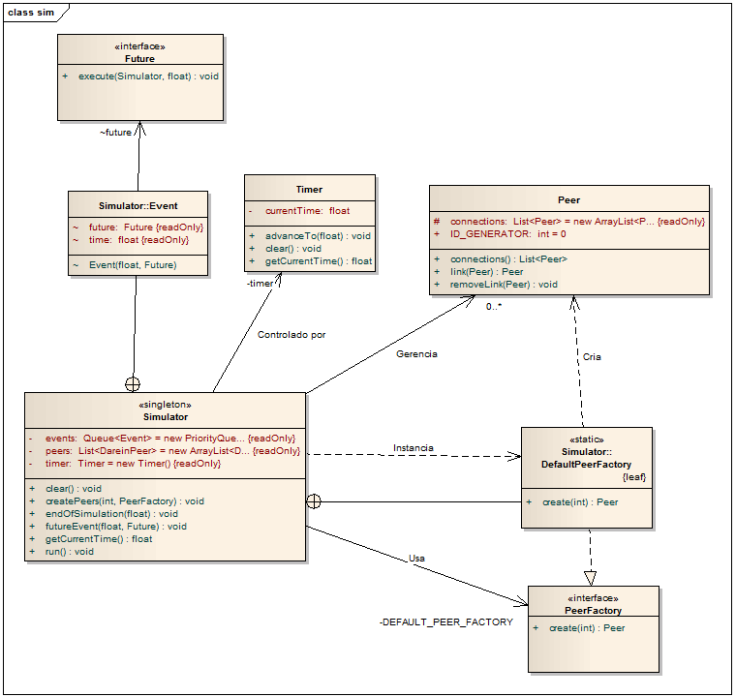

Figura 15: Diagrama de classes do simulador

O Algoritmo 5 apresenta o *loop* principal do simulador. Este algoritmo recebe como entrada um conjunto de eventos ordenados pelo tempo que cada evento deve ser executado. Quando um evento é criado, o simulador registra uma função que futuramente vai ser executada quando o simulador chegar no tempo especificado pelo evento.

O *simpleSimulator* foi concebido, mesmo sendo um simulador P2P simples, pode ser estendido facilmente através da especialização da classe *Peer* . A figura 16 apresenta o diagrama de classes especializado na *Darein* e usado pelo simulador. A interface *IRouter* contém todos os métodos que um *peer* pode chamar remotamente. Cada método desta interface pode lançar uma exceção caso não exista nenhum caminho (segundo a topologia de rede sobreposta) entre os *peers*.

A classe *DareinPeer* representa um *peer* para o ambiente simulado, esta classe é responsável por executar o acordo bizantino e o protocolo de eleição de um líder. Um *DareinPeer* conhece todos os agentes com os quais

```
Input: uma fila de eventos ordenados pelo tempo
1 evento ← remove e retorna a cabeça da fila de eventos;
2 while evento ≠ null do
3 |   avançar o relógio do simulador para o tempo que está
  |   armazenado no evento;
4 |   execucaoFutura ← callback que deve ser executada para o
  |   evento;
5 |   executar a callback execucaoFutura levando o parâmetro
  |   tempo atual do relógio da simulador;
6 |   evento ← remove e retorna a cabeça da fila de eventos;
7 end
```

Algoritmo 5: Algoritmo principal do simulador SimpleSimulator

está infectado, possuindo métodos para descobrir se está contaminado pelo agente especifico ou se está contaminado por algum agente que tenha um material genético específico. Esta classe também possui métodos para permitir que um agente contamine o *peer*.

Quando um objeto da *DareinPeer* é criado, ele recebe como parâmetro em seu construtor um objeto de *HardwareMachine* e um objeto que implementa a interface *IRouter*. Um *HardwareMachine* tem informações do *hardware* do *peer*, como: largura de banda para *upload*, *download*, número de *uploads* concorrentes e número de *downloads* concorrentes, velocidade do disco rígido para escrita e leitura. A interface *IRouter* concentra todas as chamadas remotas que um *peer* pode realizar, esta interface é implementada pelas seguintes classes: *RouterPeer*, *MaliciousRouter* e *SimulatableRouter*. Um objeto de *RouterPeer* tem a responsábilidade de rotear todas as chamadas remotas, encontrando caminhos que liguem um *peer* de origem até seu destino. Esta classe recebe como parâmetro em seu construtor dois objetos da *DareinPeer*, um representando um *peer* de origem (que está solicitando uma invocação remota) e outro de destino (onde o método será invocado remotamente). A *SimulatableRouter* é uma generalização em rotear mensagens dentro do ambiente de simulação. A classe *MaliciousRouter* representa um *peer* malicioso que corrompe o *backup* e nega o serviço de restauração.

5.3 AVALIAÇÕES

A implementação do algoritmo e do simulador foi realizada na linguagem Java. Todos os experimentos foram realizados em um MacBook Pro 2.4

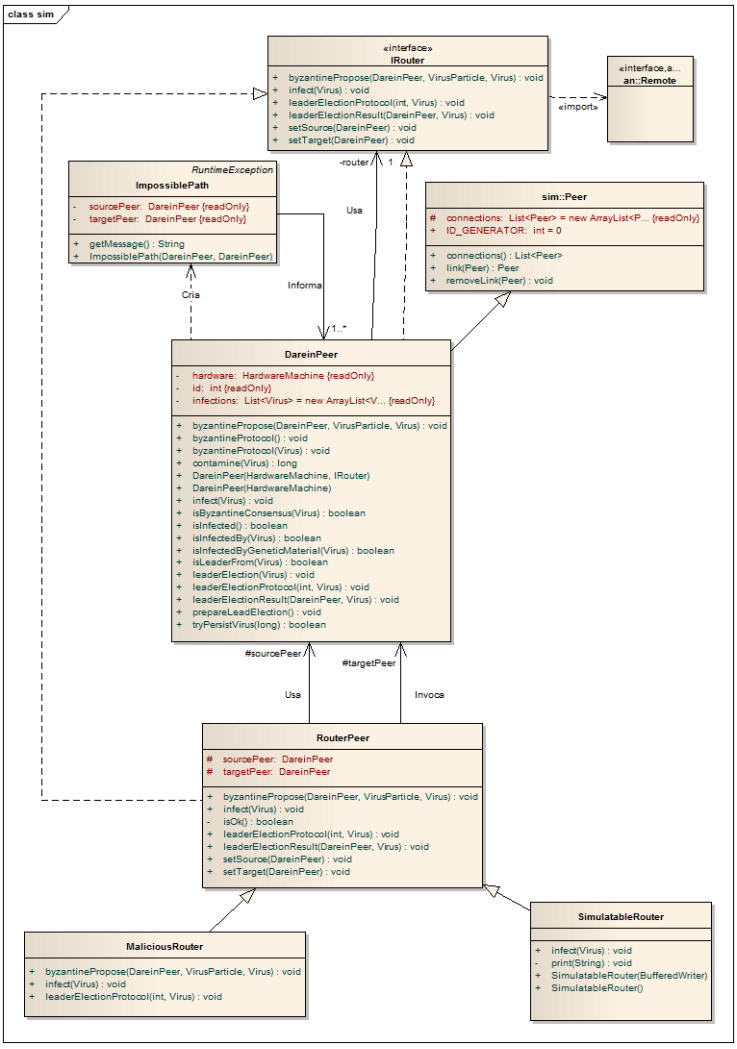

Figura 16: Diagrama de classes usado para especializar o simulador SimpleSimulator

GHz Intel Core i5, 4 GB de memória RAM e 250 GB de disco rígido, sistema operacional Mac OS X Lion 10.7.2 com Java2SE Runtime Environment 1.6.0_29. Todos os participantes da simulação P2P foram executados nessa

máquina compartilhando a mesma máquina virtual do Java.

Na figura 17 é mostrado o diagrama de pacotes que representa os dois aplicativos criados para realizar as simulações de disponibilidade e desempenho. Todos os dados gerados pelos nossos experimentos são gravados em arquivos para serem processados e analisados após o término da simulação.

O pacote usado para simular a disponibilidade (*availability*) contém classes para configurar o simulador usando o algoritmo proposto no capítulo 4. Também existem algumas classes auxiliares para manipular os arquivos de dados que foram gerados durante o experimento.

O pacote usado para simular o desempenho durante o experimento é denominado (*performance*). Este pacote possui classes para configurar o simulador usando o algoritmo proposto no capítulo 4. A classe *PerformanceParser* é usada para manipular os arquivos de dados que foram gerados durante este experimento.

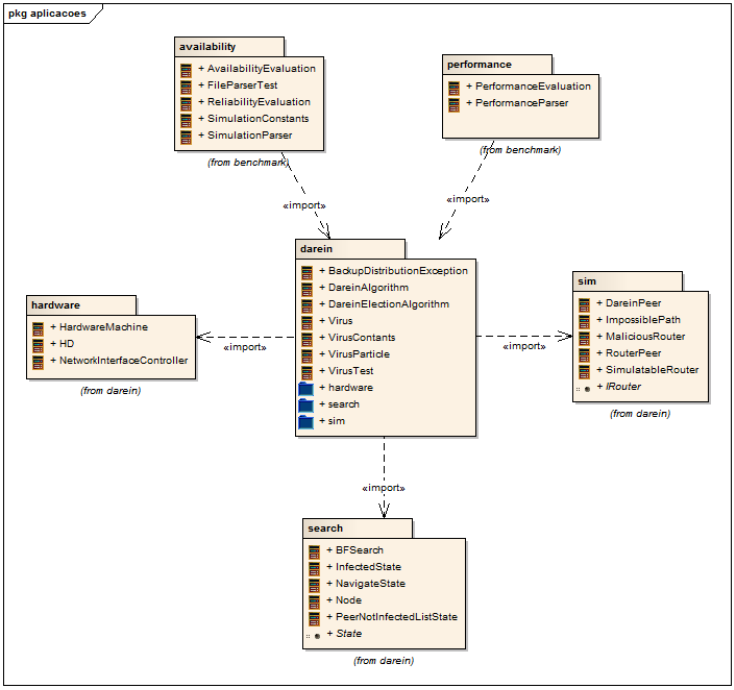

Figura 17: Diagrama de pacotes

O tamanho do conjunto de *peers*, bem como a taxa de entrada e saída

são definidos a partir de um grupo de regras com parâmetros gerados aleatoriamente durante nossos experimentos.

Segundo Pàmies-Juárez *et al.* (2010), diversos estudos tem analisado a duração da sessão de sistemas de compartilhamento de arquivos P2P (STEINER *et al.*, 2007), (STUTZBACH *et al.*, 2008) e (GUHA *et al.*, 2006). Nestes estudos é possível observar que os *peers* tendem a ficar *online* durante algumas horas. Nos experimentos realizados por Pàmies-Juárez *et al.* (2010), o *churn* usado foi de 7.14 horas e 16.7 horas. O *churn* representa a entrada e saída de *peers* da rede. A entrada e saída são eventos independentes, podendo ser temporário ou permanente. O *churn* é composto por dois componentes: o tempo entre duas chegadas consecutivas de novos *peers* e o tempo de sessão de cada *peer* (STUTZBACH; REJAIE, 2006).

Neste trabalho, o tempo simulado em todos os nossos experimentos foi de 60 dias, sendo o simulador inicializado com 100 *peers* e podendo chegar ao fim da simulação com no máximo 500 *peers*. Em todos os nossos experimentos, a cada 6 horas a topologia da rede é alterada com a entrada e saída de membros a uma taxa variável de 20% e 15% respectivamente do tamanho da população. A cada 4 horas é executado o ciclo de vida viral. O arquivo disponibilizado para *backup* tem seu tamanho variado em cada um dos experimento. A Tabela 4 mostra os dados usados pelo simulador durante a simulação.

Tabela 4: Parâmetros de configuração da simulação

Nome do Parâmetro	**Valor**
número de *peers* criados inicialmente	100
número máximo de *peers* ativos	500
fim da simulação em horas	1440
percentual de entrada de *peers*	20%
percentual de saída de *peers*	15%

Quando um *peer* é criado, para a simulação ser realista, é necessário configurar a velocidade de *download*, a velocidade *upload*, a velocidade de escrita do dado no disco rígido, a velocidade de leitura do dado do disco rígido, o espaço livre em disco rígido. Buscando ser realista, também configuramos o número de *downloads* e uploads concorrentes. O número de *download* e *upload* concorrente afeta diretamente o tempo para transmitir e receber um arquivo. Os dados usados durante todas as simulações pelos *peers* podem ser visualizados através da tabela 5.

Tabela 5: Parâmetros de configuração da simulação usados pelos *peers*

Nome do Parâmetro	Intervalo de Valores
espaço livre no *HD* em *kbyte*	524288 - 7340032
downloads concorrentes por *peer*	0 - 4
uploads concorrentes por *peer*	0 - 4
largura de banda para *upload* em *kbits*	100 - 4096
largura de banda para *dowload* em *kbits*	512 - 8196

5.3.1 Avaliação de Disponibilidade

Um sistema de alta disponibilidade pode ser definido como um sistema que procura manter os serviços o maior tempo possível *online*. Para evitar a interrupção dos serviços, os sistemas são projetados, tanto em nível de *software* como em *hardware* para assegurar redundância e quando algum componente falha, o mesmo é substituído sem a necessidade de parar todo o sistema.

A disponibilidade, em nosso trabalho, é medida através do tempo em que o arquivo fica disponível (*online*) para uma possível restauração. Nosso algoritmo procura manter o arquivo sempre disponível através do conceito de replicar o *backup* para um número adequado de *peers online*. Nós asseguramos que o *backup* está intacto através da técnica do acordo bizantino.

Todos os experimentos de disponibilidade foram simulados por um período de 60 dias. O eixo y de cada gráfico representa a taxa de disponibilidade, o eixo x representa o tempo simulado em dias. Cada gráfico apresentado possui quatro curvas de análise, sendo elas: $3f+1$ $infections$, $3f+3$ $infections$, $3f+5$ $infections$ e $3f+6$ $infections$. Para o protocolo bizantino, em todas as nossas simulações, nós configuramos o número de *peers* maliciosos para $f = 1$.

A figura 18 mostra dois grupos de experimentos com um arquivo de 4 *megabytes* de tamanho. Na imagem da esquerda, o arquivo foi distribuído na rede P2P como um único bloco de *bytes*. Na imagem da direita o arquivo distribuído foi particionado 4 blocos. Como podemos observar, para um arquivo de 4 *megabytes*, o algoritmo apresentou alta disponibilidade (98% chegando até 100%) para restaurar o *backup*, independente de quantos blocos o arquivo foi particionado. As curvas $3f+6$ $infections$ e $3f+5$ $infections$ apresentaram as melhores taxas de disponibilidade tanto para o arquivo distribuído em 4 ou em 1 bloco. Nestes experimentos com um arquivo de 4 *megabytes* de tamanho, o algoritmo apresentou uma alta taxa de disponibilidade (98% até

100%) durante o período simulado de 60 dias.

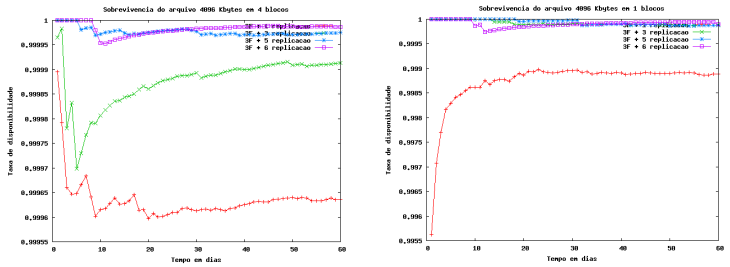

Figura 18: *Backup* de 1 arquivo com 4 *megabytes*. Na imagem (a), o arquivo foi distribuído em bloco de *bytes*. Na imagem (b), antes de distribuir, o arquivo foi particionado em blocos. Como resultado, temos uma alta taxa de disponibilidade em todas as curvas dos gráficos.

A figura 19 apresenta os gráficos da simulação do *backup* de um arquivo de 1 *gigabytes* de tamanho. Na imagem da esquerda o arquivo distribuído foi particionado em 4 blocos. Na imagem da direita, o arquivo foi distribuído em 1 bloco de *bytes*. Como podemos observar, distribuir um arquivo de 1 *gigabytes* particionado em 4 blocos apresentou melhores taxas de disponibilidade. Dentre as 4 curvas apresentadas na figura 19, a que apresentou a maior disponibilidade foi a $3f+6$ $infections$ mantendo uma média de 97% de taxa de disponibilidade.

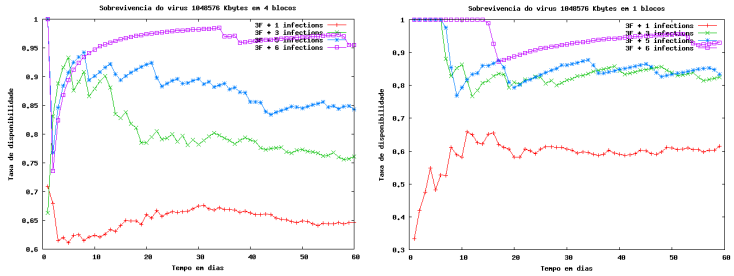

Figura 19: *Backup* de 1 arquivo com 1 *gigabytes* distribuído em 4 bloco de *bytes* (imagem da esquerda). *Backup* de 1 arquivo com 1 *gigabytes* distribuído em 1 bloco de *bytes* (imagem da direita). Como resultado, o arquivo particionado em blocos, na curva 3f + 6 apresentou a melhor taxa de disponibilidade comparada com as outras curvas.

A figura 20 apresenta os gráficos da simulação do *backup* de um arquivo de 10 *gigabytes* de tamanho. Na imagem da esquerda do canto superior, o arquivo distribuído foi particionado em 1 bloco de *bytes*. Na imagem da direita do canto superior o arquivo distribuído foi particionado em 4 blocos de *bytes*. Na imagem da esquerda do canto inferior, o arquivo distribuído foi particionado em 10 blocos de *bytes*. Na imagem da direita do canto inferior o arquivo distribuído foi particionado em 20 blocos de *bytes*. As maiores taxas de disponibilidade foram apresentadas nas curvas $3f+5$ $infections$ e $3f+6$ $infections$. Quanto menor o tamanho do bloco de *bytes*, maior é a taxa de disponibilidade para restaurar o *backup*. O gráfico onde o arquivo de 10 *gigabytes* de tamanho foi particionado em 20 blocos apresentou a melhor taxa de disponibilidade para restaurar o arquivo.

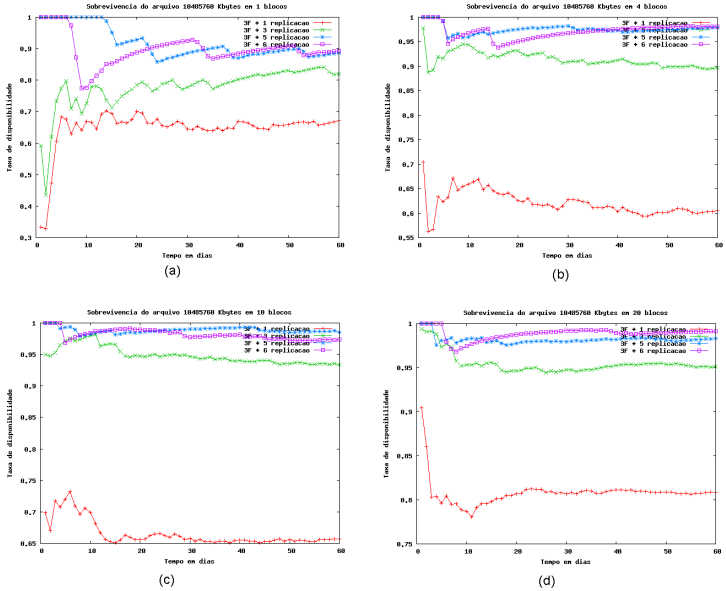

Figura 20: *Backup* de 1 arquivo com 10 *gigabytes* sendo distribuído em 1, 4, 10 e 20 blocos de *bytes*. Arquivo com 10 *gigabytes* particionado em 10 ou mais blocos apresentou alta taxa de disponibilidade.

O *pStore* Batten *et al.* (2002) apresenta disponibilidade de 95% nos experimentos realizados com uma taxa de 23% de *peers offline*. Nós realizamos experimentos com o algoritmo *Darein* levando em consideração as mesmas configurações usadas pelo *pStore*, simulamos o *backup* de dois arquivos com

tamanhos de 13 e 696 *megabytes* com uma taxa de 23% de *peers offline*. A figura 21 mostra a taxa (99.6% - 99.9%) de disponibilidade da *Darein* em um *backup* de 13 *megabytes* num experimento com 10 dias de tempo simulado. No gráfico da esquerda o arquivo foi particionado em 4 blocos de *bytes*. No gráfico da direita o arquivo foi particionado em 10 blocos de *bytes*. Ambos os gráficos, apresentaram uma alta taxa (98.6% - 99.9%) de disponibilidade. Ainda no *pStore* é apresentado um segundo experimento com um *backup* de um arquivo de 696 *megabytes*. Nós também realizamos o mesmo experimento e o resultado pode ser visualizado na figura 22. No experimento da figura 22(a) o arquivo foi particionado em 4 blocos. No experimento da figura 22(b) o arquivo foi particionado em 10 blocos. No experimento 22(c) o arquivo foi particionado em 20 blocos. O experimento que apresentou uma alta taxa (aproximadamente 97%) de disponibilidade foi (c) com a curva $3f+6$.

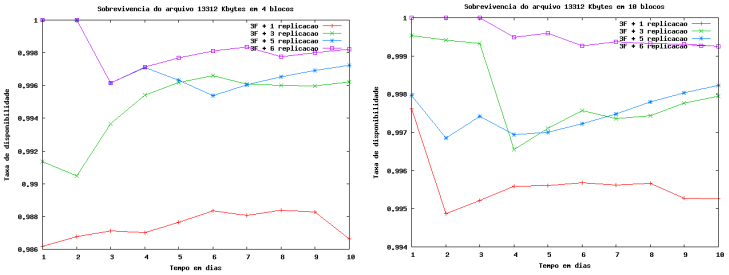

Figura 21: *Backup* de 1 arquivo com 13 *megabytes* sendo distribuído em 4 e 10 blocos de *bytes*. Ambos os gráficos apresentam taxa de disponibilidade superior ao pstore.

5.3.2 Avaliação de Desempenho

Nesta seção nós apresentamos os gráficos de desempenho do algoritmo comparando o tempo que o *backup* ficou disponível (*online*) em relação ao número de blocos que o arquivo foi particionado.

A figura 23 apresenta quatro gráficos onde foram realizados o *backup* de arquivos com tamanho: 4 *megabytes* (a), 15 *megabytes* (b), 1 *gigabyte* (c) e 10 *gigabytes* (d). Todos os gráficos apresentados na figura 23 tiveram o arquivo particionado em 1, 2, 4, 8, 10 e 20 blocos. A curva em verde representa o tempo em horas gasto para restaurar o *backup*. A curva em vermelho representa o tempo gasto para que o *backup* fique vivo no sistema.

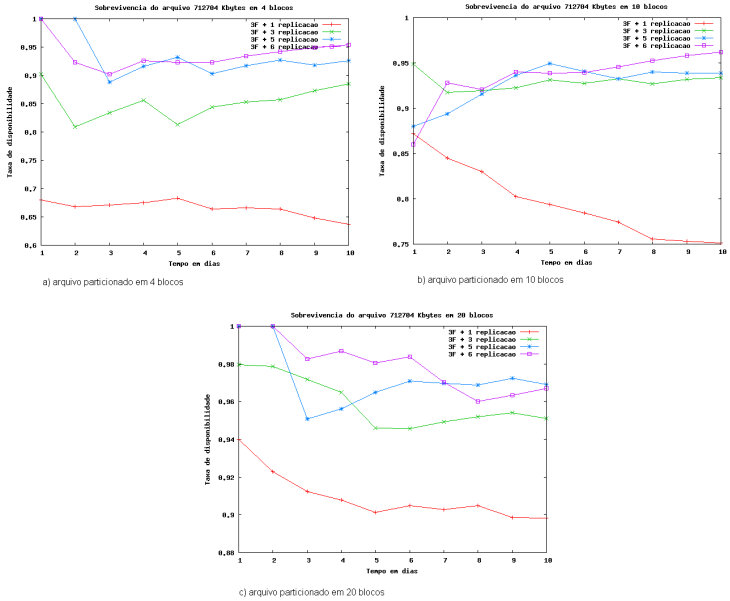

Figura 22: *Backup* de 1 arquivo com 696 *megabytes* sendo distribuído em 4, 10 e 20 blocos de *bytes*. O gráfico (c) apresenta taxas de disponibilidade superiores ao pstore.

Todos os experimentos foram simulados por um período de 10 dias com uma taxa de entrada e saída de 20% e 15% respectivamente. O experimento foi inicializado com 100 *peers* e podendo chegar até no máximo 500 *peers online*.

Para arquivos relativamente pequenos: 4 e 15 *megabytes*, referentes aos gráficos (a) e (b), quanto maior o número de blocos, menor é o tempo gasto para restaurar o *backup*. Para arquivos relativamente grandes: 1 e 10 *gigabytes*, referentes aos gráficos (c) e (d), quanto maior o número de blocos, maior é o tempo gasto para restaurar o *backup*.

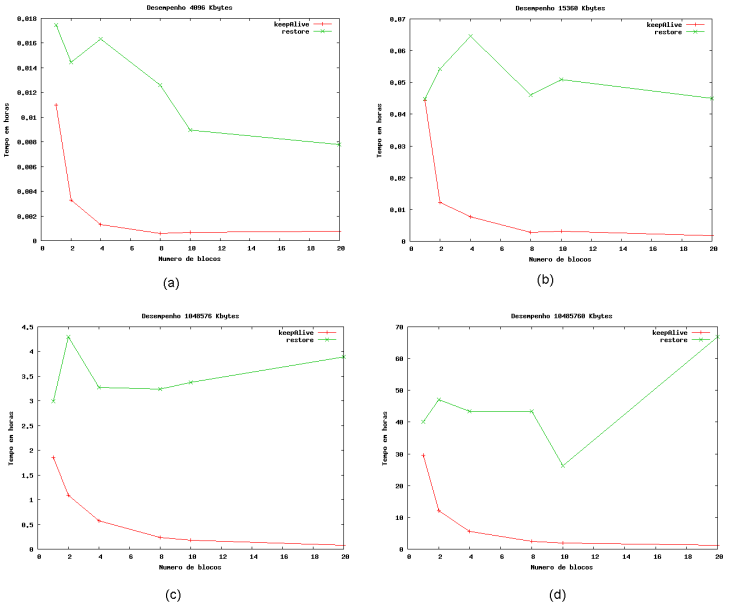

Figura 23: Desempenho do algoritmo comparando disponibilidade com tempo de restauração do *backup*. A taxa de disponibilidade estabiliza a partir de blocos. O tempo gasto para restaurar diminui quando o número de blocos aumenta. Quando o arquivo é grande, gráficos: (c) e (d), quanto maior o número de blocos, maior o tempo gasto para restaurar.

6 CONCLUSÕES

Este trabalho apresentou o algoritmo *Darein*, um algoritmo para *backup* de arquivos em redes P2P com alta disponibilidade para restaurar os arquivos sem necessitar replicar o *backup* para todos os *peers* da rede. Na restauração do *backup*, a integridade / confiabilidade foi preservada nos casos apresentados, mesmo com a presença de um nodo malicioso. Nossa implementação utiliza os recursos de *hardware* (placa de rede: capacidade de *download* e *upload*; velocidade do disco rígido) para otimizar o tempo de replicação, restauração e sobrevivência do *backup*.

O algoritmo *Darein* foi concebido utilizando o conceito de um agente benigno que tenta se replicar para nodos parceiros. Descobrimos um número adequado de réplicas ativas, evitando assim replicar o *backup* para toda população. O número adequado encontrado foi: $3f+6$ onde f é o número de processos faltosos que o sistema suporta.

Diminuir a redundância do *backup* em redes P2P mantendo a disponibilidade em taxas aceitáveis é um desafio (PÀMIES-JUÁREZ *et al.*, 2010). Através da hipótese de uma doença epidêmica controlada, conseguimos diminuir o número de redundâncias de dados para $3f+6$ réplicas mantendo uma alta taxa de disponibilidade.

Em nossos experimentos, em um ambiente onde a topologia da rede é alterada a cada 6 horas através da entrada e saída de membros a uma taxa variável de 20% e 15%, nós constatamos uma taxa de disponibilidade para restaurar o *backup* variando entre 97% - 99,9% quando o dado é replicado para $3f+6$ *peers*. O número de réplicas exigidas para realizar o *backup* em uma rede P2P, no melhor caso, onde a taxa de alteração da topologia de rede é próxima de zero, são necessárias $3f+1$ cópias do *backup*. No pior caso, onde a topologia é alterada constantemente, o *backup* deve ser replicado para pelo menos $3f+6$ réplicas, onde f representa o número de *peers* maliciosos. Neste trabalho usamos $f=1$.

O protocolo de acordo bizantino unido com o comportamento epidêmico controlado do algoritmo *Darein*, mostraram-se eficientes nos seguintes casos: no tratamento onde um *peer* foi configurado como malicioso; no tratamento para garantir a sobrevivência / disponibilidade do *backup* quando a topologia de rede sofre alterações com a entrada e saída de *peers*.

Durante a execução do *Darein*, todos os *peers* que estão armazenando uma cópia do *backup* trocam mensagens para verificar a integridade do dado armazenado. Após o *peer* ter recebido as mensagens, é feita uma avaliação para concluir se os outros *peers* são maliciosos ou não. Quando o nodo per-

cebe que existe um *peer* malicioso armazenando o *backup*, o *peer* malicioso saí da lista de *peers* que estão armazenando o *backup* e é iniciado uma busca por um novo *peer* que não esteja na lista de *peers* maliciosos. Nós consideramos que um *peer* malicioso pode: danificar / corromper o arquivo ou não responder corretamente as mensagens solicitadas pelo algoritmo. O tratamento para garantir a sobrevivência é similar ao comportamento de um *peer* malicioso que não responde as mensagens enviadas pelo algoritmo.

Também concluímos que a taxa de disponibilidade para restaurar o *backup* tem relação com o tamanho do arquivo e tamanho de cada bloco de *bytes* particionado. Quanto menor o bloco, maior é a taxa de disponibilidade, entretanto, quanto maior o arquivo e maior o número de blocos distribuídos, maior é o tempo para restaurar todo o *backup*.

A tabela 6 apresenta um comparativo entre o estado da arte de *backup* em rede P2P com a estratégia *Darein*. Nós classificamos os trabalhos em: o trabalho faz algum tratamento para diminuir consumo de banda; o trabalho usa um ambiente P2P puro (não necessita usar recursos extras como: *datacenter*, *pool* de servidores); o trabalho não necessita que o dono do arquivo verifique a integridade do dado remoto; o trabalho tem tratamento para nodos maliciosos.

Tabela 6: Comparação entre os trabalhos relacionados e o algoritmo Darein

Trabalho	**Consumo Banda**	**P2P Puro**	**Confiabilidade Remota**	**Nodo Malicioso**
Darein 2012	Sim	Sim	Sim	Sim
Toka 2012	Não	Sim	Não	Não
Defrance et al, 2011	Sim	Não	Não	Não
Toka 2010	Sim	Não	Não	Não
Li; Yun 2010	Não	Sim	Não	Sim
Tran et al, 2008	Não	Sim	Não	Sim
Courtes et al, 2007	Sim	Não	Não	Não
Ranganathan et al. 2002	Sim	Sim	Não	Não
Batten et al, 2002	Não	Sim	Não	Sim
kubiatowicz et al, 2000	Sim	Não	Não	Sim

6.1 TRABALHOS FUTUROS

Como proposta de trabalhos futuros, algumas possibilidades são apresentadas a seguir:

- Analisar a viabilidade de modificar o comportamento do algoritmo para ser adaptável a diferentes taxas de entrada e saída de nodos da rede P2P levando em consideração a presença de nodos maliciosos;
- Aumentar a tolerância a mais adversários. Este trabalho tolera apenas um adversário;
- Analisar a viabilidade de utilizar a técnica de *erase code* (DIMAKIS *et al.*, 2010) com o conceito de agente benigno proposto neste trabalho para verificar a taxa de disponibilidade durante a restauração do *backup*.

REFERÊNCIAS

ACHESON, N. H. **Fundamentals of Molecular Virology**. [S.l.]: John Wiley and Sons, Inc., 2006. ISBN 978-0471351511.

BANDARA, H. M. N. D.; JAYASUMANA, A. P. Collaborative applications over peer-to-peer systems - challenges and solutions. **CoRR**, abs/1207.0790, 2012.

BATTEN, C.; BARR, K.; SARAF, A.; TREPETIN, S. **pStore: A Secure Peer-to-Peer Backup System**. [S.l.], October 2002. 1-12 p.

BAUMGART, I.; HEEP, B.; KRAUSE, S. Oversim: A flexible overlay network simulation framework. In: **Proceedings of 10th IEEE Global Internet Symposium (GI '07)**. Anchorage, AK: IEEE, 2007. p. 79–84.

BESSANI, A.; CORREIA, M.; QUARESMA, B.; ANDRé, F.; SOUSA, P. Depsky: dependable and secure storage in a cloud-of-clouds. In: **Proceedings of the sixth conference on Computer systems**. New York, NY, USA: ACM, 2011. (EuroSys '11), p. 31–46. ISBN 978-1-4503-0634-8. Disponível em: <http://doi.acm.org/10.1145/1966445.1966449>.

BITTORRENT. [s.n.], 2012. Disponível em: <http://www.bittorrent.com/>.

BORTHAKUR, D.; GRAY, J.; SARMA, J. S.; MUTHUKKARUPPAN, K.; SPIEGELBERG, N.; KUANG, H.; RANGANATHAN, K.; MOLKOV, D.; MENON, A.; RASH, S.; SCHMIDT, R.; AIYER, A. Apache hadoop goes realtime at facebook. In: **Proceedings of the 2011 ACM SIGMOD International Conference on Management of data**. New York, NY, USA: ACM, 2011. (SIGMOD '11), p. 1071–1080. ISBN 978-1-4503-0661-4. Disponível em: <http://doi.acm.org/10.1145/1989323-.1989438>.

BOX. [s.n.], 2012. Disponível em: <https://www.box.com/>.

CHANG, E.; ROBERTS, R. An improved algorithm for decentralized extrema-finding in circular configurations of processes. **Commun. ACM**, ACM, New York, NY, USA, v. 22, p. 281–283, May 1979. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359104.359108>.

CHARRON-BOST, B. Agreement problems in fault-tolerant distributed systems. In: **Proceedings of the 28th Conference on Current Trends in**

Theory and Practice of Informatics Piestany: Theory and Practice of Informatics. London, UK: Springer-Verlag, 2001. (SOFSEM '01), p. 10–32. ISBN 3-540-42912-3. Disponível em: <http://dl.acm.org/citation-.cfm?id=647011.712835>.

COULOURIS, G. F.; DOLLIMORE, J. **Distributed systems: concepts and design**. Boston, MA, USA: Addison Wesley Longman Publishing Co., Inc., 2005. ISBN 0-201-18059-6.

COURTES, L.; HAMOUDA, O.; KAANICHE, M.; KILLIJIAN, M.-O.; POWELL, D. Dependability evaluation of cooperative backup strategies for mobile devices. In: **Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing**. Washington, DC, USA: IEEE Computer Society, 2007. p. 139–146. ISBN 0-7695-3054-0. Disponível em: <http://dl.acm.org/citation.cfm?id=1345534.1345798>.

CRISTIAN, F. Understanding fault-tolerant distributed systems. **Commun. ACM**, ACM, New York, NY, USA, v. 34, n. 2, p. 56–78, fev. 1991. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/102792-.102801>.

CRUNCHBASE. [s.n.], 2012. Disponível em: <http://www.crunchbase-.com/company/oosah>.

DABEK, F.; KAASHOEK, M. F.; KARGER, D.; MORRIS, R.; STOICA, I. Wide-area cooperative storage with cfs. **SIGOPS Oper. Syst. Rev.**, ACM, New York, NY, USA, v. 35, p. 202–215, October 2001. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/502059.502054>.

DEFRANCE, S.; KERMARREC, A.-M.; MERRER, E. L.; SCOUARNEC, N. L.; STRAUB, G.; KEMPEN, A. van. Efficient peer-to-peer backup services through buffering at the edge. In: **Proc. of the IEEE International Conference on Peer-to-Peer Computing (P2P)**. [S.l.]: IEEE, 2011. p. 142–151.

DEMERS, A.; GREENE, D.; HAUSER, C.; IRISH, W.; LARSON, J.; SHENKER, S.; STURGIS, H.; SWINEHART, D.; TERRY, D. Epidemic algorithms for replicated database maintenance. In: **Proceedings of the sixth annual ACM Symposium on Principles of distributed computing**. New York, NY, USA: ACM, 1987. (PODC '87), p. 1–12. ISBN 0-89791-239-X. Disponível em: <http://doi.acm.org/10.1145-/41840.41841>.

DIMAKIS, A. G.; GODFREY, P. B.; WU, Y.; WAINWRIGHT, M. J.; RAMCHANDRAN, K. Network coding for distributed storage systems. **IEEE Trans. Inf. Theor.**, IEEE Press, Piscataway, NJ, USA, v. 56, n. 9, p. 4539–4551, set. 2010. ISSN 0018-9448. Disponível em: <http://dx.doi.org/10-.1109/TIT.2010.2054295>.

DROPBOX. [s.n.], 2012. Disponível em: <https://www.dropbox.com/>.

DUARTE, M. P.; ASSAD, R. E.; FERRAZ, F. S.; FERREIRA, L. P.; MEIRA, S. R. de L. An availability algorithm for backup systems using secure p2p platform. In: HALL, J.; KAINDL, H.; LAVAZZA, L.; BUCHGEHER, G.; TAKAKI, O. (Ed.). **The Fifth International Conference on Software Engineering Advances, ICSEA 2010, 22-27 August 2010, Nice, France**. [S.l.]: IEEE Computer Society, 2010. p. 477–481. ISBN 978-0-7695-4144-0.

EDONKEY. [s.n.], 2012. Disponível em: <http://edonkey2000.en.softonic-.com/>.

EMULE. [s.n.], 2012. Disponível em: <http://www.emule.com/>.

GARCIA-MOLINA, H. Elections in a distributed computing system. **IEEE Trans. Comput.**, IEEE Computer Society, Washington, DC, USA, v. 31, p. 48–59, January 1982. ISSN 0018-9340. Disponível em: <http://dx.doi-.org/10.1109/TC.1982.1675885>.

GE, Z.; FIGUEIREDO, D. R.; JAISWAL, S.; KUROSE, J.; TOWSLEY, D. Modeling peer-peer file sharing systems. In: **IN PROCEEDINGS OF INFOCOM 2003**. [S.l.: s.n.], 2003. p. 2188–2198.

GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The google file system. In: **Proceedings of the nineteenth ACM symposium on Operating systems principles**. New York, NY, USA: ACM, 2003. (SOSP '03), p. 29–43. ISBN 1-58113-757-5. Disponível em: <http://doi.acm.org/10.1145-/945445.945450>.

GREVE, F. G. P. Protocolos fundamentais para o desenvolvimento de aplicações robustas. In: **23o Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos**. Fortaleza/CE, Brazil: Brazilian Computer Society, 2005. p. 330–398.

GUHA, S.; DASWANI, N.; JAIN, R. An Experimental Study of the Skype Peer-to-Peer VoIP System. In: **Proceedings of The 5th International Workshop on Peer-to-Peer Systems (IPTPS)**. Santa Barbara, CA: Springer-Verlag, 2006. p. 1 – 6.

GUO, L.; CHEN, S.; XIAO, Z.; TAN, E.; DING, X.; ZHANG, X. Measurements, analysis, and modeling of bittorrent-like systems. In: **Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement**. Berkeley, CA, USA: USENIX Association, 2005. (IMC '05), p. 4–4. Disponível em: <http://dl.acm.org/citation.cfm?id=1251086.1251090>.

JUNGLEDISK. [s.n.], 2012. Disponível em: <https://www.jungledisk.com-/>.

JXTA. [s.n.], 2012. Disponível em: <http://java.net/projects/jxta>.

KAZAA. [s.n.], 2012. Disponível em: <http://www.kazaa.com/>.

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, p. 190–201, November 2000. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/356989.357007>.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 4, p. 382–401, July 1982. ISSN 0164-0925. Disponível em: <http://doi.acm-.org/10.1145/357172.357176>.

LANDERS, M.; ZHANG, H.; TAN, K.-L. Peerstore: Better performance by relaxing in peer-to-peer backup. In: **Proceedings of the Fourth International Conference on Peer-to-Peer Computing**. Washington, DC, USA: IEEE Computer Society, 2004. (P2P '04), p. 72–79. ISBN 0-7695-2156-8. Disponível em: <http://dx.doi.org/10.1109/P2P.2004.38>.

LAPRIE, J. C. **DEPENDABLE COMPUTING AND FAULT TOLERANCE : CONCEPTS AND TERMINOLOGY**. IEEE, 1995. 2–11 p. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all-.jsp?arnumber=532603>.

LI, C.; YUN, W. A robust and secure overlay storage scheme based on erasure coding. In: **Pervasive Computing and Applications (ICPCA), 2010 5th**

International Conference. Maribor, Slovenia: IEEE Computer Society, 2010. (ICPCA '05), p. 177–182. ISBN 978-1-4244-9144-5. Disponível em: <http://dx.doi.org/10.1109/ICPCA.2010.5704094>.

LI, J.; DABEK, F. F2f: Reliable storage in open networks. In: **Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS)**. Santa Barbara, CA, USA: IEEE Computer Society, 2006. p. 1–6.

LIU, G.; SHEN, H.; WARD, L. An efficient and trustworthy p2p and social network integrated file sharing system. In: **Proceedings of the XII IEEE International Conference on Peer-to-Peer Computing (P2P 2012)**. Tarragona, Spain: IEEE Computer Society, 2012.

MAYMOUNKOV, P.; MAZIèRES, D. Kademlia: A peer-to-peer information system based on the xor metric. In: **Revised Papers from the First International Workshop on Peer-to-Peer Systems**. London, UK: Springer-Verlag, 2002. (IPTPS '01), p. 53–65. ISBN 3-540-44179-4. Disponível em: <http://dl.acm.org/citation.cfm?id=646334.687801>.

MONTRESOR, A.; JELASITY, M. PeerSim: A scalable P2P simulator. In: **Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09)**. Seattle, WA: IEEE, 2009. p. 99–100.

NAGAO, H.; SHUDO, K. Flexible routing tables: Designing routing algorithms for overlays based on a total order on a routing table set. In: **Peer-to-Peer Computing**. [S.l.: s.n.], 2011. p. 72–81.

PALANKAR, M. R.; IAMNITCHI, A.; RIPEANU, M.; GARFINKEL, S. Amazon s3 for science grids: a viable solution? In: **Proceedings of the 2008 international workshop on Data-aware distributed computing**. New York, NY, USA: ACM, 2008. (DADC '08), p. 55–64. ISBN 978-1-60558-154-5. Disponível em: <http://doi.acm.org/10.1145/1383519-.1383526>.

PÀMIES-JUÁREZ, L.; GARCIA-LOPEZ, P.; SANCHEZ-ARTIGAS, M. Availability and Redundancy in Harmony: Measuring Retrieval Times in P2P Storage Systems. In: **Peer-to-Peer Computing**. [S.l.: s.n.], 2010. p. 1–10.

RANGANATHAN, K.; IAMNITCHI, A.; FOSTER, I. Improving data availability through dynamic model-driven replication in large peer-to-peer communities. In: **Proceedings of the 2nd IEEE/ACM Interna-**

tional Symposium on Cluster Computing and the Grid. Washington, DC, USA: IEEE Computer Society, 2002. (CCGRID '02), p. 376–. ISBN 0-7695-1582-7. Disponível em: <http://dl.acm.org/citation-.cfm?id=872748.873217>.

RAUNAK, M. S.; OSTERWEIL, L. J.; WISE, A. Developing discrete event simulations from rigorous process definitions. In: **Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium**. San Diego, CA, USA: Society for Computer Simulation International, 2011. (TMS-DEVS '11), p. 117–124. Disponível em: <http://dl.acm.org/citation.cfm?id=2048476.2048491>.

ROWSTRON, A.; DRUSCHEL, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In: **IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)**. [S.l.: s.n.], 2001. p. 329–350.

RUSSELL, S. J.; NORVIG, P. **Artificial Intelligence - A Modern Approach (3. internat. ed.)**. [S.l.]: Pearson Education, 2010. I-XVIII, 1-1132 p. ISBN 978-0-13-207148-2.

SAROIU, S.; GUMMADI, K. P.; GRIBBLE, S. D. Measuring and analyzing the characteristics of napster and gnutella hosts. **Multimedia Syst.**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 9, n. 2, p. 170–184, ago. 2003. ISSN 0942-4962. Disponível em: <http://dx.doi.org/10-.1007/s00530-003-0088-1>.

SCHOLLMEIER, R. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In: **Proceedings of the First International Conference on Peer-to-Peer Computing**. Washington, DC, USA: IEEE Computer Society, 2001. (P2P '01), p. 101–. ISBN 0-7695-1503-7. Disponível em: <http://dl.acm.org/citation-.cfm?id=882470.883282>.

SHEN, X.; YU, H.; BUFORD, J.; AKON, M. **Handbook of Peer-to-Peer Networking**. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2009. ISBN 0387097503, 9780387097503.

SKYPE. [s.n.], 2012. Disponível em: <http://www.skype.com>.

STEINER, M.; EN-NAJJARY, T.; BIERSACK, E. W. A global view of kad. In: **Proceedings of the 7th ACM SIGCOMM conference on Internet**

measurement. New York, NY, USA: ACM, 2007. (IMC '07), p. 117–122. ISBN 978-1-59593-908-1. Disponível em: <http://doi.acm.org/10.1145-/1298306.1298323>.

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>.

STUTZBACH, D.; REJAIE, R. Understanding churn in peer-to-peer networks. In: **Proceedings of the 6th ACM SIGCOMM conference on Internet measurement**. New York, NY, USA: ACM, 2006. (IMC '06), p. 189–202. ISBN 1-59593-561-4. Disponível em: <http://doi.acm.org/10-.1145/1177080.1177105>.

STUTZBACH, D.; REJAIE, R.; SEN, S. Characterizing unstructured overlay topologies in modern p2p file-sharing systems. **IEEE/ACM Trans. Netw.**, IEEE Press, Piscataway, NJ, USA, v. 16, n. 2, p. 267–280, abr. 2008. ISSN 1063-6692. Disponível em: <http://dx.doi.org/10.1109-/TNET.2007.900406>.

TOKA, L.; AMICO, M. Dell; MICHIARDI, P. Online data backup : a peer-assisted approach. In: **P2P 2010, 10th IEEE International Conference on Peer-to-Peer Computing, August 25-27, 2010, Delft, The Netherlands**. Delft, PAYS-BAS: IEEE, 2010. Disponível em: <http://www.eurecom.fr/publication/3140>.

TOKA, L.; CATALDI, P.; DELL'AMICO, M.; MICHIARDI, P. Redundancy management for p2p backup. **CoRR**, abs/1201.2360, 2012. Disponível em: <http://dblp.uni-trier.de/db/journals/corr/corr1201.html>.

TRAN, D. N.; CHIANG, F.; LI, J. Friendstore: cooperative online backup using trusted nodes. In: **Proceedings of the 1st Workshop on Social Network Systems**. New York, NY, USA: ACM, 2008. (SocialNets '08), p. 37–42. ISBN 978-1-60558-124-8. Disponível em: <http://doi.acm.org-/10.1145/1435497.1435504>.

VERISSIMO, P.; RODRIGUES, L. **Distributed Systems for System Architects**. Norwell, MA, USA: Kluwer Academic Publishers, 2001. ISBN 0792372662.

XNAP. [s.n.], 2012. Disponível em: <http://xnap.sourceforge.net/>.

YANG, J.; MA, H.; SONG, W.; CUI, J.; ZHOU, C. Crawling the edonkey network. In: **Grid and Cooperative Computing Workshops - GCC 2006, 5th International Conference, Changsha, Hunan, China, 21-23 October 2006, Proceedings**. [S.l.]: IEEE Computer Society, 2006. p. 133–136. ISBN 0-7695-2695-0.