ALEXANDRE LUIS NODARI

REPLICAÇÃO OTIMISTA DE DADOS ESTRUTURADOS EM REDES PEER-TO-PEER

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática.

CURITIBA

2007

ALEXANDRE LUIS NODARI

REPLICAÇÃO OTIMISTA DE DADOS ESTRUTURADOS EM REDES PEER-TO-PEER

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para obtenção do título de Mestre em Informática.

Área de Concentração: Sistemas Distribuídos

Orientador: Prof. Dr. Alcides Calsavara

CURITIBA

2007

Nodari, Alexandre Luis.
Replicação otimista de dados estruturados em redes peer-to-peer.
Curitiba, 2007. 109p.

Dissertação (Mestrado) – Pontifícia Universidade Católica do Paraná.
Programa de Pós-Graduação em Informática.

1. Peer-to-peer. 2. Replicação Otimista. 3. Alta Disponibilidade.

I.Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e de Tecnologia. Programa de Pós-Graduação em Informática Aplicada

Para meus pais.

Agradecimentos

Agradeço ao meu orientador, Alcides, pelo incentivo, apoio, dedicação e colaboração prestadas no desenvolvimento deste trabalho.

Agradeço ao amigo e colega de mestrado, Akatsu, pelo companheirismo e colaboração prestados durante todo o período do curso.

Agradeço aos amigos, Aron, Caimi, Leonardo, Matias e Ramon, e a empresa Objective Solutions pelo incentivo e apoio.

Agradeço minha mulher, Gisela, por entender que o período de pouca atenção e tempo de minha parte era passageiro.

Agradeço meus pais pela vida, formação e incentivo ao crescimento pessoal e profissional.

Agradeço a todos os que não estão nesta lista e que de alguma forma contribuíram para que eu chegasse até aqui.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Os sistemas que são executados sobre redes peer-to-peer têm evoluído rapidamente nos últimos anos. Essa arquitetura é caracterizada pela capacidade de se adaptar a falhas e acomodar nós transientes, mantendo um desempenho aceitável. Atualmente as aplicações mais comuns são as de compartilhamento de arquivos, mas uma variedade de novas aplicações executadas sobre essa plataforma está surgindo.

Aplicações para compartilhar arquivos utilizam replicação dos mesmos, portanto a replicação atua principalmente sobre dados não estruturados e imutáveis. Porém, a evolução das aplicações peer-to-peer revelam a necessidade de replicação de outros tipos de dados, incluindo dados estruturados. Este trabalho tem por objetivo avaliar as várias opções de replicação otimista de dados estruturados para redes peer-to-peer.

Para atingir este objetivo foram realizadas pesquisas sobre sistemas distribuídos, redes peer-to-peer e replicação de objetos. Foram avaliadas várias características de replicação otimista. E uma implementação de referência foi desenvolvida para testar as opções dessas características. A principal contribuição deste trabalho é a apresentação dessa implementação de referência e uma análise dos resultados das simulações feitas com as opções citadas acima, destacando as relacionadas ao ambiente peer-to-peer.

Abstract

Peer-to-peer systems are quickly evolving. This architecture can adapt to failures and support transient nodes, keeping a good performance. Nowadays the most common type of applications is multimedia file sharing, but new kinds of applications are being developed.

File sharing applications already use replication of those files, which usually are unstructured and immutable data. But the evolutions of peer-to-peer applications indicate a need for different types of replication, including structured data replication. This thesis has the objective of evaluating the many options for structured data replication in peer-to-peer networks.

To achieve this objective a research on distributed systems, peer-to-peer networks and data replication took place. Many characteristics of optimistic replication were evaluated. A reference implementation was developed to test the choices. The main contribution of this work is this implementation and an analysis of the various simulations upon it, with emphasis in peer-to-peer characteristics.

Capítulo 1 Introdução

O sucesso da Internet resulta em um número elevado de pesquisas envolvendo sistemas distribuídos. Os objetivos destas pesquisas, dentre outros, têm sido oferecer um melhor desempenho e maior confiabilidade a estes sistemas [BHAGWAN, 2002]. Uma arquitetura que se destaca nesse ambiente é a de redes peer-to-peer.

Por sua capacidade de descentralização, facilitando a entrada e saída de nós, vários sistemas peer-to-peer têm feito sucesso e isto torna esta arquitetura um tema de pesquisa bastante procurado. Toda esta evolução pode ser comprovada pela adoção desta tecnologia pelas grandes plataformas de desenvolvimento de software comercial, como é o exemplo do JXTA [JXTA, 2007] para a plataforma Java e C++.

A replicação de informações neste tipo de rede é fundamental para este sucesso. Uma vantagem de replicar dados é aumentar sua disponibilidade. Com várias réplicas de uma informação disponíveis é possível melhorar o desempenho de uma aplicação. E com algum controle a mais sobre as réplicas consegue-se atingir tolerância à faltas. Além das vantagens citadas acima, para aplicações de compartilhamento de arquivos, a replicação de dados pode auxiliar na resistência à censura.

1.1 Motivação

Os mecanismos de replicação existentes hoje em sistemas peer-to-peer atua principalmente sobre dados não estruturados e imutáveis, pois sua principal utilização é nas aplicações de compartilhamento de arquivos multimídia, como indicam as pesquisas [CUENCA, 2003] E [LIN, 2004]. Nessas aplicações a replicação é implícita no mecanismo de

compartilhamento, usando eventualmente fragmentos dos arquivos compartilhados. Como tantos os arquivos quanto seus fragmentos são dados imutáveis, não é necessário utilizar um mecanismo de replicação que mantenha consistência entre as réplicas. Além de imutáveis, os dados não são estruturados, o que simplifica sua fragmentação.

Porém, a evolução desses sistemas e o surgimento de outros tipos de aplicações, como as de comunicação e colaboração, ou até a convergência da arquitetura peer-to-peer com a de computação em grade sugere também a necessidade de replicação de dados estruturados, como discute [ANDROUTSELLIS, 2004]. A Tabela 1 apresenta uma comparação de exemplos.

Tabela 1. Exemplos de aplicações distribuídas de acordo com as características dos dados.

	Dado imutável	**Dado mutável**
Dado não estruturado	Compartilhamento de arquivos: • Música • Vídeos	Edição colaborativa de arquivos: • Controle de versões
Dado estruturado	Compartilhamento de metadados de arquivos: • Referências Bibliográficas	Compartilhamento de metadados de aplicações: • Lista de contatos • Dados de gerência de redes • Dados de execução de aplicações em grade • Dados de autenticação

Várias aplicações podem tirar proveito de replicação de dados estruturados. Algumas aplicações de compartilhamento de arquivos replicam metadados sobre os arquivos compartilhados. Essa replicação poderia permitir consultas mesmo em nós desconectados. Ou permitir consultas estruturadas como, por exemplo, “buscar músicas do álbum nome_do_album do intérprete nome_do_intérprete”.

Aplicações de comunicação e colaboração poderiam evitar a necessidade de um servidor central se as funcionalidades de autenticação, inicialização e atualização de lista de contatos estivessem replicadas.

Um sistema de arquivos distribuído tira proveito da replicação de metadados para aperfeiçoar o acesso aos arquivos, como em [BOLOSKY, 2000]. Assim como uma ferramenta de armazenamento distribuída pode aperfeiçoar o acesso aos dados, como acontece em [KUBIATOWICZ, 2000].

Aplicações distribuídas poderiam replicar dados de controle de redes, como por exemplo, [GRANVILLE, 2005]. Ou replicar dados de execução da aplicação em grade de forma a se tornarem adaptáveis ao ambiente, redirecionando tarefas para nós com maior banda ou maior processamento disponível, conforme [SCHULER, 2003].

1.2 Objetivo

Existem ainda outras aplicações distribuídas, que tratam de dados estruturados e mutáveis, que normalmente são implementadas na arquitetura cliente-servidor. Algumas dessas aplicações poderiam ter melhorias em disponibilidade, desempenho e escalabilidade se convertidas para peer-to-peer (Tabela 1).

Para utilizar replicação na arquitetura peer-to-peer com dados mutáveis, faz-se necessário o emprego de um mecanismo de replicação que garanta a consistência das réplicas. Essa garantia depende da estruturação do dado. Caso ele seja estruturado, a granularidade da consistência será de cada um dos itens, e possíveis subitens recursivamente, do dado. Caso contrário, a granularidade será o próprio dado, e as operações de escrita afetarão o mesmo completamente.

O objetivo deste trabalho é avaliar opções de replicação de dados estruturados em redes P2P. Em redes de longa distância ou formada por computadores móveis, os algoritmos

mais indicados são os de replicação otimista [SAITO, 2005]. A avaliação é feita executando simulações sobre uma implementação de referência.

1.3 Desafios

Para atingir esse objetivo o primeiro desafio é identificar protocolos de replicação otimista adequados a dados estruturados em redes peer-to-peer, avaliando as opções das seguintes características, entre outras:

- Quantas réplicas recebem comandos de alteração de informações.
- Ordenar as operações recebidas nas réplicas.
- Transferir as alterações entre as réplicas.
- Garantia suportada para divergência entre as mesmas.

O segundo desafio é implementar os protocolos de replicação identificados sobre plataformas próprias para o desenvolvimento sobre arquitetura peer-to-peer. Por fim, uma avaliação do desempenho e eficácia dos protocolos implementados deve ser feita através de simulações.

1.4 Metodologia

A primeira fase desse trabalho é composta por uma pesquisa sobre a arquitetura peer-to-peer. Segue-se um estudo sobre replicação de dados. Verifica-se que a replicação otimista é mais interessante para aplicações com esta arquitetura, então os estudos sobre replicação otimista são aprofundados.

A segunda fase do trabalho é uma análise sobre arcabouços de infra-estrutura para aplicações com arquitetura peer-to-peer. Seleciona-se um arcabouço e sobre ele é desenvolvida uma aplicação de referência. Esta aplicação é utilizada para uma série de simulações, variando-se parâmetros de configuração de mecanismos de replicação.

A terceira fase do trabalho é a análise dos resultados das simulações. Nessa fase são gerados tabelas e gráficos que facilitam o entendimento dos resultados possibilitando as conclusões finais.

1.5 Organização deste documento

Este documento está organizado em sete capítulos. Neste primeiro capítulo foi apresentada uma introdução à pesquisa.

O próximo capítulo apresenta o estudo sobre redes peer-to-peer. Alguns projetos com estruturas relevantes para a pesquisa também são abordados.

O terceiro capítulo apresenta o estudo sobre replicação de dados. Projetos relevantes também são abordados.

O quarto capítulo apresenta o mecanismo proposto. Apresentam-se aqui as escolhas feitas e o escopo do projeto.

O quinto capítulo apresenta a implementação do mecanismo e as definições das simulações executadas.

O sexto capítulo apresenta os resultados obtidos com as simulações e seus gráficos.

O último capítulo finaliza este trabalho com conclusões finais e trabalhos futuros.

Capítulo 2 Arquitetura Peer-to-Peer

2.1 Definição

Encontra-se na literatura um número relativamente grande de definições para sistemas com arquitetura peer-to-peer. Uma definição bastante abrangente, mas que especifica bem o termo é a seguinte: “Sistemas de compartilhamento de recursos computacionais (conteúdo, armazenamento, ciclos de CPU) por contato direto, não requerendo a intermediação ou suporte de um servidor ou autoridade central” [ANDROUTSELLIS, 2004].

Assim como computação em grade, peer-to-peer se aplica ao compartilhamento de recursos em ambientes distribuídos de larga escala. E à medida que a arquitetura peer-to-peer evolui, com aplicações de distribuição de dados estruturados, colaboração remota e computação distribuída em redes parece haver uma forte convergência entre estas arquiteturas.

Este tipo de arquitetura tem sido uma opção à arquitetura cliente / servidor, pois evita a necessidade de investimento em um servidor central. Além disso, se mostra mais robusta com relação a certos tipos de problemas, como nós transientes ou falhas de rede, em relação a sistemas cliente / servidor. A arquitetura peer-to-peer também apresenta o benefício da escalabilidade, devido à facilidade de adicionar ou remover nós. Mas poucas aplicações tiram proveito destas vantagens, pois muitas usam um sistema híbrido, onde algumas funcionalidades ainda dependem de um ou mais servidores.

2.2 Classificação

A partir da definição apresentada acima, pode-se fazer a seguinte classificação.

Comunicação e colaboração: aplicações que oferecem as funcionalidades de comunicação e colaboração, normalmente em tempo real entre dois nós. Exemplos: Skype e Google Talk.

Computação distribuída: aplicações que são executadas tirando proveito da capacidade de processamento de nós disponíveis na rede, quebrando uma tarefa complexa em unidades pequenas de trabalho e distribuindo para eles, como citado em [RANGANATHAN, 2002]. Exemplos: Seti@Home e Genome@Home.

Compartilhamento de conteúdo: aplicações para distribuição de arquivos com conteúdo multimídia ou outros dados, como analisado em [TSOUMAKOS, 2003]. Exemplos: Gnutella e Kazaa.

Outros serviços: algumas pesquisas têm sido feitas sobre sistemas gerenciadores de banco de dados distribuídos, onde podemos destacar como exemplos o Edutella e o OceanStore [KUBIATOWICZ, 2000]. Outros serviços, principalmente de infra-estrutura também têm sido pesquisados, como veremos na seção 2.5 .

2.3 Roteamento e Localização

Com relação ao roteamento de mensagens e localização de nós, as redes com arquitetura peer-to-peer são classificadas em três diferentes categorias. Algumas, denominadas **centralizadas**, possuem um servidor central ao qual os usuários podem submeter comandos e consultas. Outras redes, denominadas **descentralizadas**, não possuem um servidor central. Os nós que formam essa rede trocam comandos e consultas através de seus vizinhos. Uma terceira categoria é denominada **híbrida**, onde alguns nós assumem funções mais importantes que outros, como proposto em [RODRIGUES, 2002].

2.4 Estrutura

Uma rede descentralizada ou híbrida pode ser **estruturada**, indicando um acoplamento entre sua topologia e a localização das informações dentro dela. Redes **não estruturadas** não determinam relação entre sua topologia e a localização das informações, conforme [COHEN, 2002]. O tipo mais comum de rede atualmente é a não estruturada sem servidor central, como por exemplo, nas aplicações Gnutella e Kazaa, mas as aplicações de comunicação e colaboração, que são centralizadas, têm crescido rapidamente.

Tabela 2 Classificação de projetos com arquitetura peer-to-peer

		Centralização		
		Híbrido	**Parcial**	**Não**
Estrutura	**Não**	Napster, Skype, Google Talk, Messenger	Kazaa	Gnutella
	Estruturado (Sistema)			OceanStore, Past, Kademlia
	Estruturado (Infra)			Chord, Can, Tapestry, Pastry

2.5 Projetos de infra-estrutura peer-to-peer

Diversas plataformas para o desenvolvimento de aplicações peer-to-peer já foram desenvolvidas, cada uma com seu próprio foco, entre as quais podemos citar o JXTA, o Chord, o CAN e o Pastry.

2.5.1 JXTA

JXTA (do inglês juxtapose) é uma especificação independente de linguagem e plataforma para comunicação entre dispositivos sem considerar sua localização física e tecnologia de rede no qual se encontram instalados [JXTA, 2007]. Segue os conceitos, protocolos e serviços do JXTA Core, que é a plataforma básica do JXTA. Sobre essa

plataforma são construídos outros serviços, alguns dos quais muito importantes para o desenvolvimento de aplicações peer-to-peer, porém, não trata de replicação de dados.

Aplicações desenvolvidas para suportar o trabalho em grupo devem, obviamente, organizar seus usuários em grupos. Os grupos podem ser organizados segundo diferentes critérios e esta atividade faz parte do projeto da aplicação colaborativa. Para tornar esta aplicação realidade, a plataforma JXTA possui as seguintes entidades mapeadas para os principais conceitos de computação distribuída: Peer, Peer Group, Edge Peers, Minimal Peers, Proxy Peers, Rendez-vous Peers e Relay Peers.

Todas as entidades da plataforma JXTA são representadas usando advertisements; documentos XML bem formatados contendo informação a respeito dessas entidades. A plataforma JXTA define seis advertisements básicos: Peer, Peergroup, Pipe (canal virtual de comunicação entre peers), Service (abstração para serviço oferecido por um Peer ou Peergroup), Content (abstração para conteúdo publicado) e Endpoint (pontos de conexão de um pipe).

Peers transmitem mensagens apenas através de pipes, canais virtuais que são, em geral, unidirecionais e não-confiáveis, e que se conectam a um ponto de entrada e outro de saída (end points). Mensagens são documentos XML bem formatados, que possuem roteamento baseado no ID da fonte, carregando em seu cabeçalho (header) a informação de roteamento necessária, tal como a seqüência de peers a ser percorrida.

Baseado em trocas de mensagens definidas pelos protocolos da plataforma, sete serviços básicos fornecidos pela plataforma JXTA são providos por qualquer peergroup criado: Rendezvous, Resolver, Discovery, Access, Pipe, Membership e Monitoring. Os protocolos da plataforma são: Rendez-vous Protocol, Peer Resolver Protocol, Peer Discovery

Protocol, Peer Information Protocol, Pipe Binding Protocol, Endpoint Routing Protocol e Membership Protocol.

2.5.2 Chord

Chord é uma infra-estrutura para localização e roteamento em redes peer-to-peer. Para isso ela mapeia identificadores dos dados para identificadores de nós. O identificador do nó é calculado aplicando-se uma função de hash sobre seu endereço IP. Para cada dado é calculada uma chave e um par (chave, dado) é armazenado no nó referente à aplicação da função hash sobre sua chave.

Os nós são ordenados em um espaço circular que varia de 0 a 2^m. A Figura 1 mostra um exemplo com m=3 e três nós conectados (pontos cinza na Figura 1), consequentemente existem cinco nós desconectados (pontos pretos na Figura 1). Uma chave k é atribuída ao primeiro nó conectado cujo identificador é igual ou superior a k. Esse nó é então denominado sucessor da chave k (Successor na Figura 1). O uso de uma função de hash consistente causa um balanceamento de carga, dado que cada nó deve receber o mesmo número de chaves.

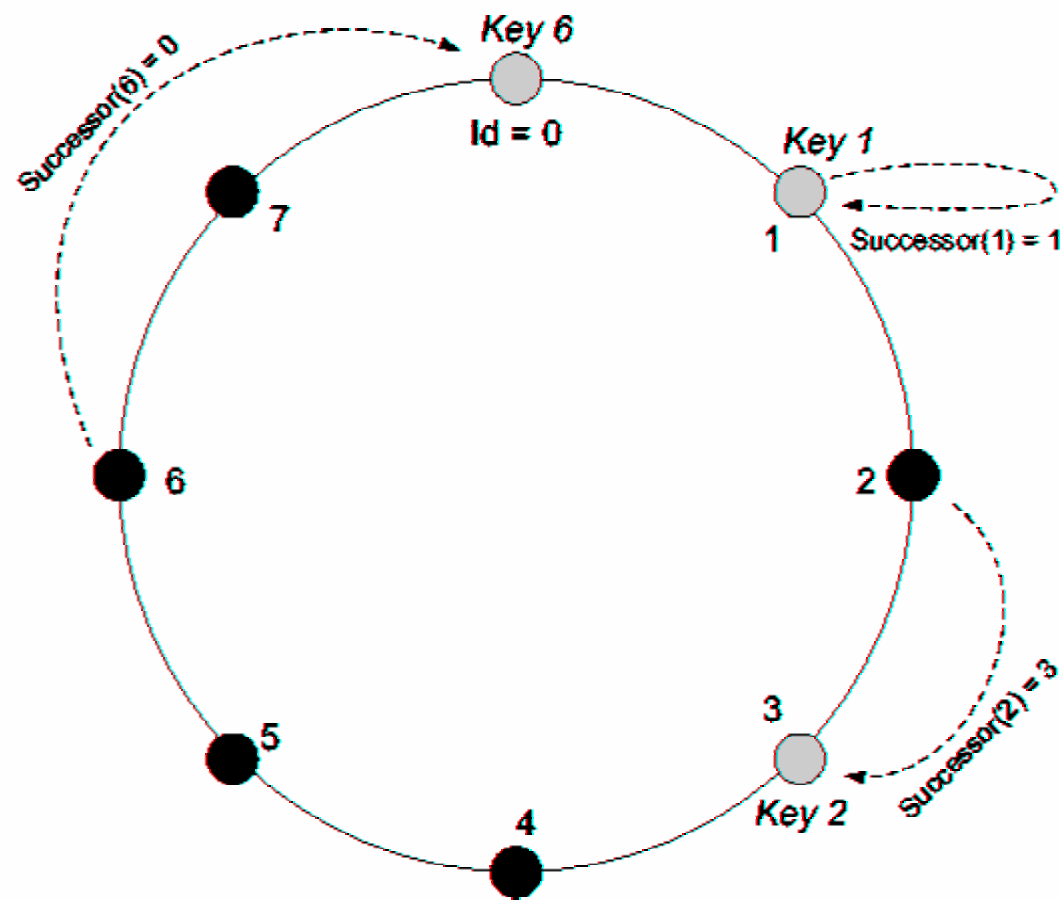

Figura 1 Exemplo de Chord com m=3.

A única informação necessária para efetuar o roteamento de mensagens é cada nó saber de seu sucessor. Com isso, qualquer mensagem é repassada ao seu sucessor até atingir um nó que possua sua chave. Para aumentar a eficiência de localização cada nó mantém informações adicionais de roteamento em forma de uma finger table.

Chord não implementa replicação de dados, deixando isso a cargo da aplicação. Há, porém, uma proposta para que cada dado seja armazenado sob diferentes chaves vindas de informações do dado no nível da aplicação. Outra limitação do Chord é não permitir ao usuário o controle da localização dos dados.

2.5.3 CAN

CAN é a sigla para Content Addressable Network. Ela forma uma rede estruturada peer-to-peer semelhante a uma hash table. Essa tabela é denominada espaço de coordenadas Cartesianas d-dimensional. Esse espaço de coordenadas é completamente lógico, não tendo relação nenhuma com as coordenadas físicas do sistema.

Esse espaço é particionado em zonas e cada nó é responsável por uma zona, mantendo também informações sobre zonas adjacentes. A Figura 2 mostra exemplos de espaço de coordenadas bidimensionais divididos em cinco nós e seis nós, e suas zonas adjacentes (neighbour set). Qualquer chave k é mapeada de forma determinística por uma função hash para um ponto P no espaço de coordenadas. Um par (chave, valor) é armazenado no nó responsável pela zona que contém o ponto P para o qual a chave foi mapeada.

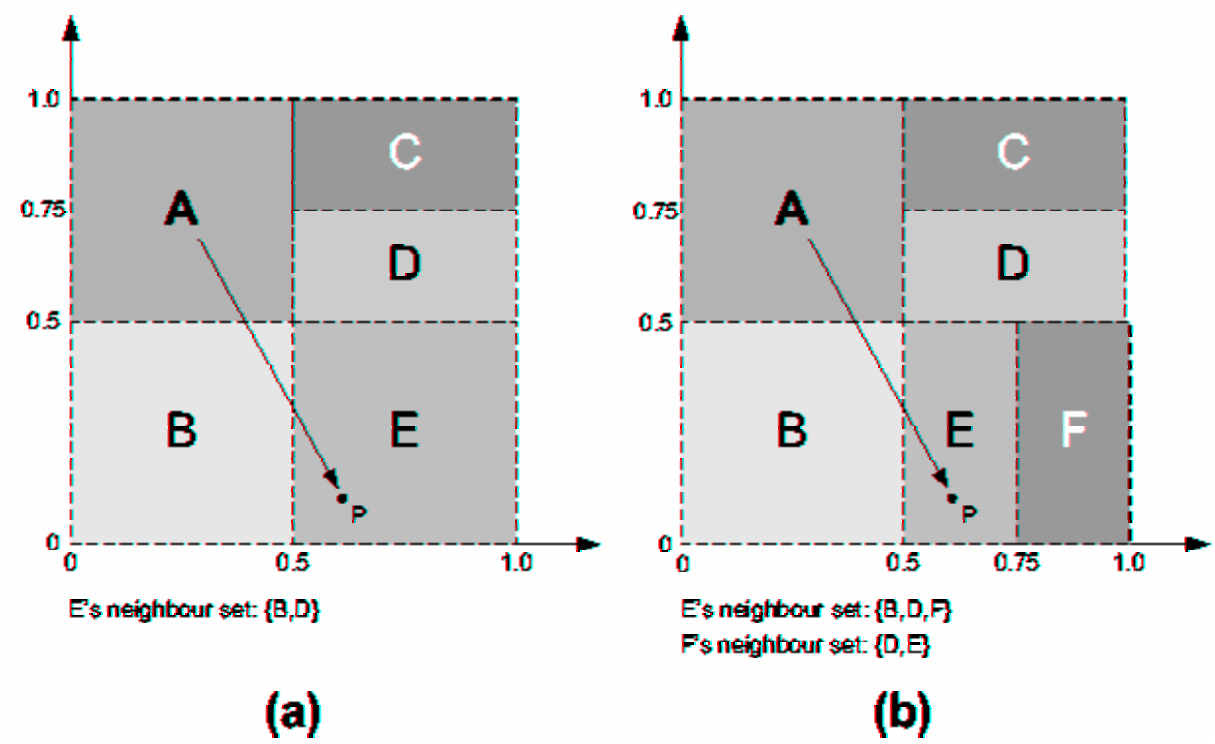

Figura 2 Exemplo de espaço de coordenadas CAN, (a) com cinco nós e (b) com seis nós.

Qualquer nó pode recuperar esse par (chave, valor) executando a mesma função hash sobre a chave para obter o ponto. Se o nó não for responsável pela zona que contém esse ponto, a mensagem deve ser roteada pela infra-estrutura CAN até que atinja o nó responsável. Intuitivamente, o roteamento segue uma linha reta através do espaço de coordenadas.

2.5.4 Pastry

Pastry é um arcabouço de localização e encaminhamento de mensagens peer-to-peer. Ele forma uma rede sobreposta a Internet, mais robusta e que se organiza automaticamente. Para isso, cada nó recebe um identificador único de 128 bits, denominado nodeId. Os nós ficam uniformemente distribuídos na rede, como pode ser visto na Figura 3, formando um espaço circular que varia de 0 a (2^{128}-1).

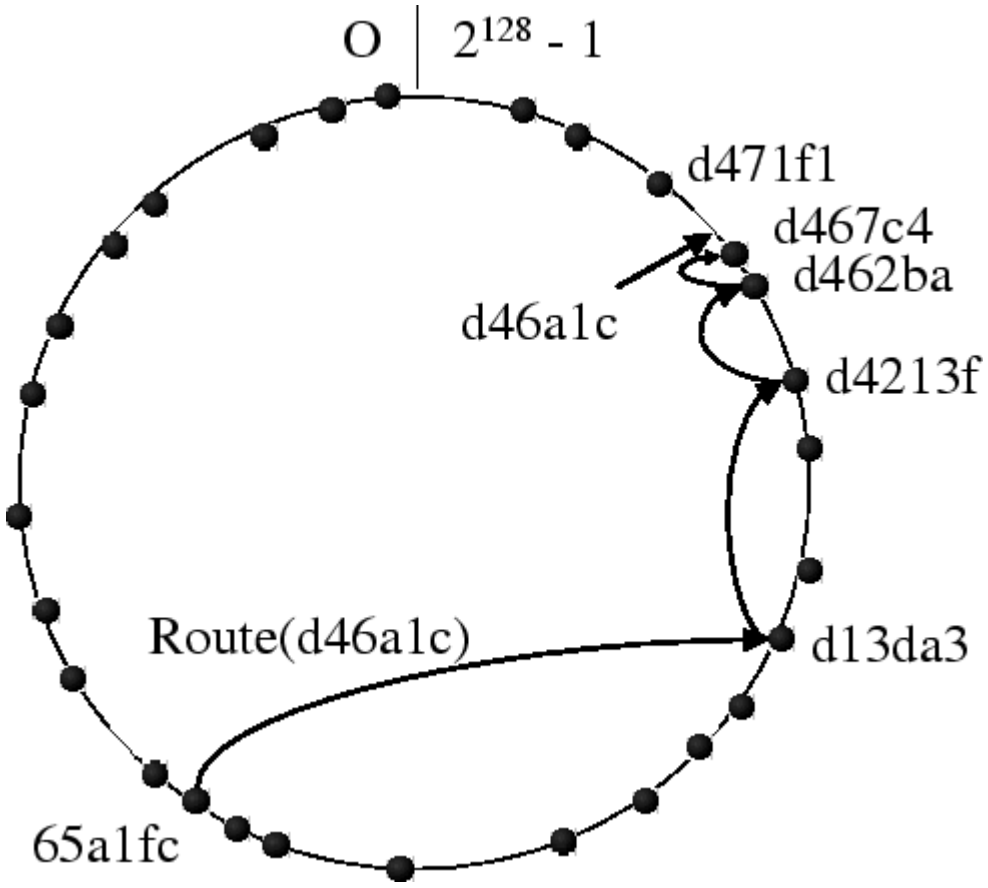

Figura 3. Exemplo de endereçamento de mensagem, de 65a1fc para d46a1c, pelo Pastry.

Diversas aplicações relacionadas com esse arcabouço são objetos de estudo, como por exemplo, Gnutella, Tapestry e OceanStore.

O arcabouço garante a entrega de uma mensagem para o nó vivo com nodeId mais próximo numericamente da rede. As mensagens são encaminhadas em média seguindo uma função logarítmica de retransmissões. Para atingir isso cada nó armazena três conjuntos de informações: uma tabela de roteamento, um conjunto de vizinhos e um conjunto de folhas. A tabela de roteamento armazena endereços IP dos nós para possibilitar a retransmissão das mensagens. O conjunto de vizinhos armazena endereços IP dos nós mais próximos e o conjunto de folhas armazena os nós com nodeID adjacentes, menores e maiores. Um exemplo das três tabelas, de nós folha (Leaf set), de roteamento (Routing table) e de vizinhos (Neighbourhood set) pode ser visto na Figura 4.

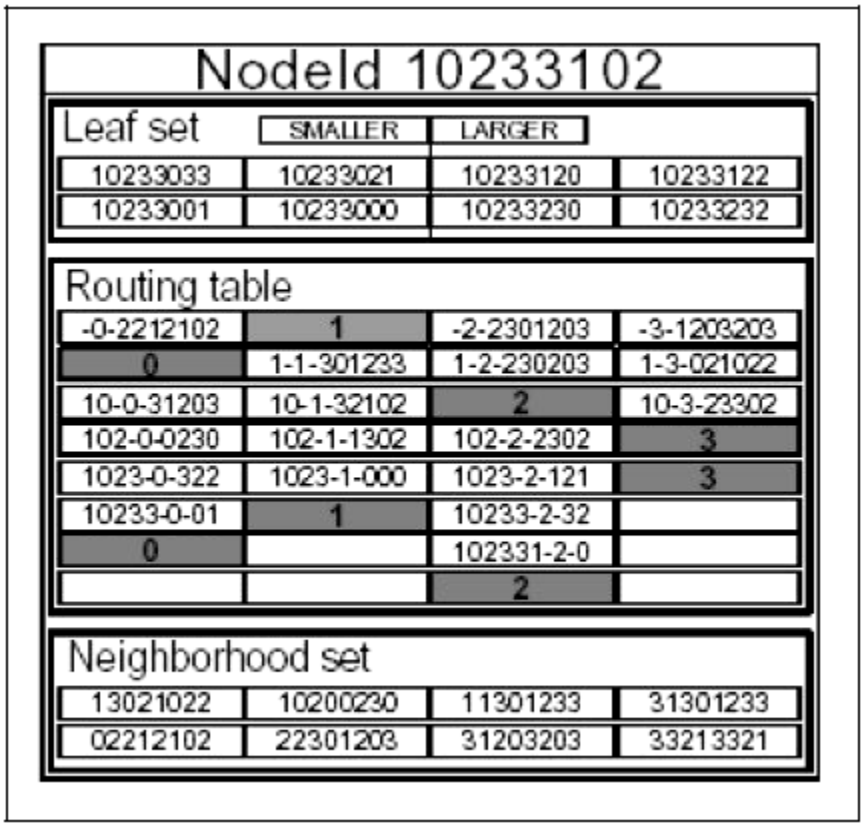

Figura 4. Exemplo de tabelas de nós folha, roteamento e nós vizinhos.

2.6 Conclusão

Sistemas peer-to-peer têm sido bastante pesquisados devido ao sucesso de algumas aplicações comerciais. Com isso surgiram muitas opções de sistemas de infra-estrutura e aplicações nessa área. Foram citados nesse capítulo apenas alguns dos mais importantes arcabouços para desenvolvimento de aplicações existentes.

Dos arcabouços citados acima, o JXTA e o Pastry têm recebido mais atenção nas pesquisas ultimamente. Os dois oferecem implementações públicas e gratuitas. O JXTA utiliza o conceito de peers diferenciados para certas funções. O Pastry trabalha com redes completamente descentralizadas e é mais simples que o JXTA.

A descentralização é uma característica muito importante na arquitetura peer-to-peer. E a simplicidade do arcabouço facilita o desenvolvimento de protótipos e exemplos, como pode ser visto em seus trabalhos relacionados. Por isso, optou-se pela utilização do arcabouço Pastry para a implementação de referência. O arcabouço apesar de simples oferece todos os subsídios necessários para o desenvolvimento de aplicações nesta arquitetura. Porém, os conceitos apresentados nesse trabalho não dependem da infra-estrutura utilizada.

Capítulo 3 Replicação de Dados

Replicação de dados consiste em manter múltiplas cópias de certos dados, denominados réplicas. É uma funcionalidade muito importante para sistemas distribuídos, melhorando seu desempenho e sua disponibilidade [BHAGWAN, 2003]. Sistemas peer-to-peer já utilizam replicação de dados, principalmente em aplicações de compartilhamento de conteúdo. Porém, esta replicação normalmente trata as informações como não estruturadas e imutáveis.

3.1 Formas de replicação de dados imutáveis

Esta replicação pode acontecer de três formas. Uma replicação natural acontece ao copiar um conteúdo de um nó para outro. A partir deste momento, os dois nós possuem cópias da informação.

Uma replicação em forma de cachê acontece durante a transferência das informações, onde os nós, através dos quais as informações passam para ir do nó origem para o nó destino, armazenam uma cópia das informações para posteriores consultas.

Formas de replicação pró-ativas podem ser obtidas analisando introspectivamente a troca de mensagens para definir quando e onde réplicas devem ser criadas. Outra opção é definir parâmetros, como quantidade mínima de réplicas ou distância máxima até uma réplica, para que o sistema crie as mesmas. Ainda é possível que os usuários determinem quantas e onde vão estar as réplicas de determinadas informações.

3.2 Replicação de dados mutáveis

Para algoritmos de replicação em sistemas distribuídos que tratam de informações mutáveis, ou seja, aceitam operações de escrita, existem duas classificações. A primeira determina qual(is) nó(s) recebe(m) as operações de escrita e como e quando essas operações são transferidas para as réplicas [LUNG, 2004], determinando as opções passiva e ativa. Outra forma de classificação de algoritmos de replicação em sistemas distribuídos determina como lidar com operações de escrita e leitura concorrentes, determinando as opções pessimista e otimista.

3.2.1 Replicação Passiva

Nesta técnica, somente um nó, denominando primário, recebe, executa e responde as invocações dos clientes. Para assegurar que as réplicas do primário, denominadas backup, se mantenham consistentes, o primário envia periodicamente mensagens de checkpoint para os backups. Os backups não recebem requisições de clientes, sendo sua função substituir o primário em caso de falha. Nesse caso, um algoritmo de detecção de falha determina o estado do primário e um algoritmo de eleição elege o backup que assumirá seu papel. A desvantagem desta técnica é que há possibilidade de o novo primário não estar completamente atualizado em relação ao antigo [PASIN, 2003].

Uma variação da técnica passiva é denominada coordinator-cohort. Nesta variação todas as réplicas recebem a mensagem do cliente. Porém, o coordinator, que equivale a um primário, executa o comando e então envia mensagens para os cohorts, que são equivalentes a backups. Em caso de falha do coordinator, um cohort é capaz de assumir a requisição do cliente sem comprometer o estado dos dados.

Outra variação da técnica passiva que evita a necessidade de determinar uma falha em qualquer uma das réplicas foi proposta na pesquisa [DÉFAGO, 2004].

3.2.2 Replicação Ativa

Nesta abordagem todas as réplicas recebem, executam e respondem as invocações dos clientes. A desvantagem desta técnica é a exigência maior de recursos do sistema e um fluxo mais elevado de mensagens nos meios de comunicação. Porém, as falhas podem ser mascaradas quase que instantaneamente. Isso torna essa técnica interessante para aplicações real-time.

3.2.3 Replicação Pessimista

Na replicação pessimista, apesar de existirem várias réplicas, o usuário tem a ilusão de estar sempre trabalhando com uma única cópia dos dados, com uma alta disponibilidade. Isso pode ser obtido de várias formas, porém, em todas elas tem-se que bloquear o acesso a cada uma das réplicas até que a mesma seja atualizada.

Técnicas pessimistas funcionam bem em redes locais, onde a latência de rede é pequena e falhas não são comuns. Porém, a Internet ainda é lenta e pouco confiável. E redes formadas por computadores móveis, com conectividade intermitente, estão se tornando mais populares. Algoritmos pessimistas também não escalam bem redes distribuídas [HAMADI, 2005]. E algumas atividades requerem distribuição de dados de forma assíncrona por natureza.

3.2.4 Replicação Otimista

Na replicação otimista não há a necessidade de sincronização durante o acesso as réplicas. A replicação otimista permite que a leitura e escrita de dados sejam feitas assumindo de forma “otimista” que conflitos raramente irão acontecer. E se acontecerem, os conflitos podem ser resolvidos.

A replicação otimista oferece vantagens sobre a pessimista para determinados tipos de aplicação, principalmente sobre redes peer-to-peer.

- Maior disponibilidade, pois o sistema permite acesso às informações mesmo sem que todas as réplicas estejam acessíveis.
- Flexibilidade com relação à topologia da rede, não havendo necessidade de conhecimento da localização das réplicas.
- Melhor escalabilidade, permitindo um maior número de réplicas.
- Maior autonomia aos nós, reduzindo a comunicação entre as réplicas.

Estas vantagens, porém, vêm com um custo. Qualquer sistema distribuído enfrenta um equilíbrio entre disponibilidade e consistência. Na tentativa de obter este equilíbrio o desafio é manter réplicas divergentes e resolver os conflitos de operações concorrentes.

Exemplos de replicação otimista podem ser encontrados nos sistemas de sincronização de PDAs - Personal Digital Assistants, no serviço da Internet DNS - Domain Name System ou no sistema de controle de versões CVS - Concurrent Versions System.

3.3 Características de replicação otimista

Ao especificar e desenvolver um sistema com replicação otimista é necessário fazer algumas escolhas de projeto [SAITO, 2005]. Segue uma análise sucinta das mais importantes.

3.3.1 Número de Réplicas para Escrita

Os sistemas que só possuem uma réplica que aceita operações de atualização das informações, ou seja, comandos de escrita são denominados single-master. Sistemas onde mais de uma réplica aceita operações de escrita são denominados multi-master.

Sistemas single-master são mais simples, porém têm sua disponibilidade para operações de escrita reduzida, principalmente em sistemas com operações de escrita freqüentes. A escalabilidade do sistema single-master está limitada ao desempenho da única réplica que recebe operações de escrita.

Sistemas multi-master oferecem maior disponibilidade, mas com um aumento significativo de complexidade. E apesar da escalabilidade ser maior que em um sistema single-master, ao aumentar o número de réplicas que aceitam operações de escrita aumenta-se também a taxa de conflitos entre operações, determinando que esta vantagem não seja normalmente muito grande.

3.3.2 Ordenação de Comandos

Como um sistema com replicação otimista pode aceitar comandos de escrita criados independentemente, é necessário que o sistema ordene e eventualmente detecte conflitos entre estes comandos.

Existem duas formas de ordenar mensagens em um sistema distribuído. A forma sintática ordena as mensagens baseada em quem, quando e onde elas foram criadas. É uma forma simples e genérica, mas pode indicar concorrência entre mensagens desnecessariamente. A forma semântica é mais complexa, pois se baseia em outras informações específicas da mensagem e da aplicação, dificultando a criação de algoritmos reutilizáveis.

Uma maneira simples de ordenar mensagens de forma sintática foi proposta por Lamport em 1978. Outra forma sintática foi proposta por Parker em 1983. Esta solução é denominada Vector Clocks ou vetor de versões [SAITO, 2005].

3.3.3 Detecção de Conflitos

Políticas de detecção de conflitos também são classificadas em sintáticas e semânticas. E suas características também são semelhantes, as sintáticas são mais simples e genéricas, porém podem detectar falsos conflitos, enquanto as semânticas são mais flexíveis, mas dependentes da aplicação.

Uma forma sintática de detectar conflitos é determinar que operações concorrentes sejam conflitantes, ou seja, aquelas que o mecanismo de ordenação não conseguiu determinar a precedência.

Análise semântica das operações pode permitir que operações concorrentes não sejam conflitantes. Determina-se assim que estas operações possuem a propriedade da comutatividade, ou seja, podem ser executadas em qualquer ordem que a consistência do sistema é mantida. Exemplos de políticas semânticas:

- Um sistema operacional distribuído que determina que operações em arquivos diferentes sejam comutáveis
- Um aplicativo de controle de versão de arquivos que determina que alterações em linhas diferentes de um mesmo arquivo não são conflitantes.

3.3.4 Resolução de Conflitos

A maior parte dos sistemas não resolve conflitos. Muitos deles seguem a Thomas´s Write Rule, onde a informação considerada mais recente sobrepõe todas as outras réplicas desta informação. Mas, idealmente, resolver conflitos elimina a possibilidade de um usuário do sistema perder uma operação de escrita já executada sobre determinada informação.

Porém, sempre que possível, é interessante prevenir conflitos antes de eles acontecerem. Algoritmos pessimistas usam lock e abort de operações. Sistemas single-master evitam conflitos aceitando operações somente em uma réplica, porém reduzindo a disponibilidade. Conflitos também podem ser reduzidos agilizando a propagação de operações e dividindo as informações em unidades menores e independentes [BHAGWAN, 2002].

A resolução de um conflito pode ser manual ou automática. Na manual, a intervenção do usuário é necessária e são apresentadas a ele duas ou mais versões da informação para que ele decida e crie a versão do objeto que deve ser armazenada. Exemplos desta forma de

resolução podem ser encontrados em sistemas de sincronização de PDAs e no controle de versionamento de arquivos CVS.

A resolução automática de conflitos é mais complexa e dependente da aplicação. Para isso podem ser utilizadas pré-condições e procedimentos para mesclar informações conflitantes vinculadas às operações de escrita do sistema. Alguns sistemas de arquivos distribuídos mesclam informações em conflito automaticamente.

Um mecanismo de reconciliação com resolução de conflitos para o ambiente APPA foi proposto na pesquisa [MARTINS, 2006]. Esse mecanismo é multi-master e independente do tipo de dado replicado. Para minimizar o número de conflitos é feita uma análise semântica das operações oferecendo garantia eventual.

3.3.5 Consenso de Resolução

O consenso sobre operações garante que todas as réplicas que aceitam operações de escrita concordem e apliquem a mesma seqüência definida. Uma vantagem é permitir o controle de quais operações já estão disponíveis em todas as réplicas e consequentemente não podem ser desfeitas. Também permite definir um limite menor de informações sobre os comandos de escrita que ainda precisam ser mantidas por cada réplica, pois informações sobre operações em consenso podem ser descartadas.

O consenso pode ser obtido automaticamente em sistemas que conseguem obter uma ordenação de operações totalmente determinística. Exemplos são sistemas single-master e sistemas que seguem a regra Thomas's Write Rule. O consenso também pode ser obtido através de algoritmos específicos envolvendo as réplicas participantes.

3.3.6 Transferência de Estado

Existem duas maneiras de transmitir alterações em dados para suas réplicas. A primeira é transferir todas as informações novamente, sobrepondo às informações existentes.

Esse mecanismo é denominado transferência de estado. Outra forma é enviar somente a operação que causou a alteração, denominada transferência de operações.

A maneira mais simples de transferência de estado é denominada Thomas's Write Rule e foi proposta em 1976 [SAITO, 2005]. Nele cada réplica guarda o timestamp da última alteração. Periodicamente réplicas trocam os timestamps de determinado dado e os comparam. O dado na réplica com timestamp mais recente é sobreposta ao outro. Este método não detecta conflitos podendo causar perda de operações.

Uma maneira de minimizar o problema de não detecção de alguns conflitos do método acima é cada réplica armazenar os dois últimos timestamps dos comandos de escrita em cada dado. Mas esta nova forma pode detectar falsos conflitos quando o número de réplicas for maior que dois.

Uma outra forma de determinar uma transferência de estado é a partir da comparação de Vector Clocks. Quando utilizado desta forma normalmente é denominado de vetor de versões.

Os métodos de transferência de estado requerem um tratamento especial para deleção de dados. Para cada dado a ser deletado é necessário armazenar uma marcação sobre a deleção, para que ela seja distribuída para as outras réplicas. Porém, se não houver um controle sobre estas marcações, elas podem causar problemas de espaço e desempenho.

3.3.7 Transferência de Operação

Muitos sistemas usam Vector Clocks para controlar a transferência de operações. Como cada entrada do vetor é referente a uma operação, por comparação dos vetores determinam-se exatamente quais operações precisam ser enviadas para cada réplica.

Dependendo do tamanho das operações e dos dados, algoritmos híbridos podem melhorar o desempenho da aplicação, ora enviando operações ora enviando estados. A divisão

das informações em pedaços também ajuda, como por exemplo, quebrar um sistema de arquivos em pastas ou um arquivo binário em blocos.

Algumas aplicações que usam transferência de operações usam detecção de conflitos semântica, pois a forma como o sistema descreve operações facilita essa análise.

3.3.8 Topologia de Rede

Para transferir operações ou estados entre as réplicas, elas devem estar conectadas respeitando uma topologia. Topologias fixas como, por exemplo, em estrela ou hierárquica, podem ser muito eficientes, mas não se aplicam em redes dinâmicas e com muitas falhas.

Se alguns nós tiverem um papel mais importante nas transferências, a topologia pode ser considerada semi-estruturada. Topologias ad-hoc, apesar de usualmente mais lentas, atingem uma disponibilidade alta e um balanceamento de carga entre nós de redes dinâmicas.

3.3.9 Momento de transferência

Para definir quando transferir uma operação ou estado pode-se optar pela técnica Push. Nesta técnica o nó que recebe uma operação inicia imediatamente a propagá-la para os outros nós. Isso é recomendado, pois ajuda a diminuir a divergência entre réplicas e evita buscas (pull) desnecessárias.

As técnicas Push incluem envio de mensagens por inundação, que é a forma mais simples e envio de mensagens por multicast, que apesar de reduzir as mensagens necessárias não é confiável.

Técnicas Pull não são recomendadas para redes dinâmicas por aumentar o número de mensagens sobre um ambiente dinâmico e mais propenso à falhas. Mas em sistemas single-master é possível usar uma técnica híbrida onde usualmente a réplica master propaga (push) as novas operações, mas também uma réplica que está se conectando a rede pode buscar (pull)

na réplica master as últimas alterações. A técnica de Matriz de Timestamps, que é uma evolução de Vector Clocks pode ser usada para este fim [SAITO, 2005].

3.3.10 Garantia de Consistência

Apesar de na replicação otimista não ser possível garantir que todas as réplicas estejam com as informações idênticas, não é interessante que um usuário veja dados divergentes ao acessar réplicas diferentes.

Uma maneira de minimizar os problemas causados por diferenças entre réplicas é garantir a ordem de escrita e leitura. Uma das formas é através de garantia por sessão. Por exemplo, um usuário que altera sua senha em uma réplica, ao tentar se autenticar em outra réplica, que ainda não recebeu a operação de alteração, não é autenticado até que esta réplica tenha a nova senha. Esta garantia, que é denominada RYW - Read Your Writes - é apenas uma das possibilidades [SAITO, 2005].

Outra técnica de garantia de consistência é impor limites de divergência. O limite pode ser por tempo, por número de operações ou por diferença máxima de determinados valores.

Alguns sistemas ainda analisam probabilisticamente suas trocas de operações na tentativa de minimizar a divergência das réplicas e o número de mensagens necessárias para isso.

3.4 Projetos com replicação em peer-to-peer

3.4.1 Projeto Rumor

Este projeto utiliza replicação otimista de arquivos em computadores móveis. Esses computadores possuem uma conectividade mais fraca que computadores conectados em redes com fio. A latência é consideravelmente maior, a banda é limitada e políticas de conservação de energia desencorajam a comunicação. Algumas transferências de dados geram um custo

extra ao usuário do computador móvel e permanecer mais tempo desconectado do que conectado é o padrão [GUY, 1998].

Portanto o projeto opta pela replicação otimista, pois isso reduz os requisitos de banda e conectividade utilizados para propagar as operações de escrita. Apesar de a replicação otimista permitir conflitos nos dados replicados, simulações e experiência de uso indicam que esses conflitos são raros e fáceis de resolver.

Outra opção deste projeto é a descentralização dos dados. Todos os nós executam funções semelhantes dentro do sistema. A pouca previsibilidade da conexão dos nós torna isso interessante. A terceira opção foi por reconciliações periódicas ao invés de propagação imediata das operações de escrita. Isso foi motivado pelo custo da comunicação entre os nós.

A técnica de reconciliação utilizada é pull. E acontece somente entre duas réplicas, diminuindo a necessidade de réplicas estarem disponíveis simultaneamente. A detecção de conflitos é feita usando vetor de versões, descrito acima. E um algoritmo distribuído, de duas fases, sem coordenador é utilizado para tratar da deleção de informações.

3.4.2 Projeto Osiris

Este projeto apresenta uma arquitetura distribuída e descentralizada de gerência de processos [SCHULER, 2003]. OSIRIS é a sigla para Open Service Infrastructure for Reliable and Integrated process Support. Ele combina vantagens de outras infra-estruturas como:

- Encontrar e invocar Web Services.
- Suporte para execução de processos com garantias semelhantes a sistemas de gerência de workflow.
- Ligação tardia a serviços e balanceamento de carga.
- Execução de processos em redes peer-to-peer, indicando uma arquitetura descentralizada e com escalabilidade.

Para isso, as meta-informações sobre os provedores de serviços e sua carga são replicados entre os nós. Porém, cada nó se subscreve para a parte das meta-informações de interesse, reduzindo o número de mensagens de atualização.

A garantia de consistência das réplicas também foi relaxada para obter um melhor desempenho. A meta-informação de carga de cada nó, que varia muito rapidamente, só é atualizada em uma réplica se estiver defasada em mais de 10% em relação ao nó master, por exemplo.

A replicação segue técnicas de publish / subscribe. E as informações são armazenadas de forma semi-estruturada, em arquivos XML. Então cada cliente pode se subscrever para um documento. A cada alteração deste documento o repositório publica as alterações para os clientes inscritos. As replicações podem ser parciais, caso não haja necessidade de replicação completa, para minimizar a quantidade de dados transmitidos. E as alterações de dados não considerados como vitais acontecem com menos freqüência.

3.4.3 Projeto OceanStore

Esse projeto oferece uma infra-estrutura para armazenamento de dados projetada para ter nível global e permitir acesso constante às informações persistidas [KUBIATOWICZ, 2003]. Ela assume que a rede é formada por servidores não confiáveis e oferece redundância e criptografia para minimizar possíveis problemas. Para melhorar o desempenho, um cachê dos dados é utilizado extensivamente.

Cada objeto a ser persistido no OceanStore recebe um identificador único, denominado GUID. O objeto é então persistido em múltiplos servidores. Cada réplica é independente do servidor onde está armazenada. Objetos podem ser alterados por operações de escrita. A cada operação de escrita o objeto ganha um novo número de versão. Para

permitir operações de escrita concorrentes o OceanStore utiliza um modelo de resolução de conflitos, que apesar de flexível, requer computação sobre os dados em cada servidor.

A gerência de réplicas ajusta o número e a localização das réplicas para aumentar a eficiência do acesso às mesmas. Existem monitores de requisições de clientes e de carga do sistema analisando acessos excessivos as réplicas. Quando esses acessos ultrapassam um limite aceitável, o sistema cria réplicas adicionais em nós próximos para balancear a carga. Da mesma forma, réplicas são eliminadas quando caem em desuso.

3.4.4 Projeto ManP2P

A gerência de redes de computadores é essencial para grande parte das grandes empresas. As soluções atuais de gerência de redes suportam a monitoração e controle de dispositivos e serviços localizados dentro de um domínio administrativo de rede. Porém, com o aumento de conectividade entre empresas, a utilização da Internet como meio de transmissão e a adoção de computação em grade, surge a necessidade de um novo modelo distribuído de gerência de redes de computadores [PANISSON, 2006].

O modelo de comunicação da arquitetura peer-to-peer, difere das soluções atuais, pois foi construído sobre protocolos de comunicação de Internet para operar com sistemas altamente distribuídos. Uma rede peer-to-peer é formada por nós e conexões lógicas entre eles, operando de forma independente de endereços Internet e seu roteamento.

O projeto ManP2P propõe um modelo de gerência de redes de computadores que estende as soluções atuais, incorporando características da arquitetura peer-to-peer, obtendo auto-organização, tolerância à faltas, escalabilidade e suporte a conectividade intermitente. Essas características proporcionam uma melhora nos sistemas de gerência de redes oferecendo suporte para gerência inter-domínios, troca de mensagens confiável e balanceamento de carga.

A solução forma uma rede peer-to-peer sobreposta à rede gerenciada. Ela foi implementada utilizando o arcabouço JXTA e utiliza Peer Groups para agrupar nós oferecendo os serviços necessários.

3.4.5 Projeto LogMiddle

A evolução dos computadores móveis e dos celulares cria uma crescente demanda por aplicações distribuídas que enfocam colaboração em redes móveis ad-hoc. Os modelos de comunicação de redes fixas não atende corretamente esse novo ambiente. As redes peer-to-peer têm se tornado comuns em aplicações distribuídas e, apesar de ter sido desenvolvida inicialmente em redes fixas, pode ser facilmente estendido para redes sem fio [DIÓGENES, 2006].

O LogMiddle é um middleware peer-to-peer para computação móvel. Ele utiliza replicação otimista como forma de compartilhamento de dados. Ele permite execução de operações de escrita em réplicas desconectadas e os conflitos que ocorram são resolvidos por um processo de reconciliação que dissemina as alterações de forma epidêmica para outras réplicas. Para isso é utilizado o conceito de log de operações. Esse log armazena não só os dados, mas também os metadados utilizados no processo de versionamento e reconciliação das réplicas.

3.5 Conclusão

É comum aos projetos que trabalham sobre redes ad-hoc ou peer-to-peer a utilização de replicação otimista. Essa opção normalmente traz economia de recursos de máquinas e redes importantes para esses projetos.

Ainda assim, essa replicação otimista pode ser feita de muitas formas diferentes, como pode ser visto nos projetos acima. Pode-se optar por receber comandos de escrita em somente

uma réplica, ou em várias réplicas de forma potencialmente concorrente. Pode-se optar por transferência imediata de alterações, ou por reconciliações posteriores. Pode-se optar por transferir somente os comandos de escrita, ou substituir a informação inteira nas réplicas. Mas nenhum trabalho relacionado oferece replicação otimista de dados estruturados sobre redes peer-to-peer como proposto nesse trabalho.

Todas essas opções apresentam vantagens e desvantagens em relação às outras. Cabe ao desenvolvedor analisar os requisitos de cada aplicação para definir qual o melhor conjunto de definições para cada projeto.

Capítulo 4 Decisões de projeto

Com a evolução das aplicações peer-to-peer surge a necessidade de replicação de dados estruturados e mutáveis. Os metadados destas aplicações são exemplos comuns de dados estruturados que precisam ser replicados. Em aplicações de compartilhamento de conteúdo, a replicação de metadados dos arquivos compartilhados pode oferecer consultas mais detalhadas e pesquisas em nós desconectados da rede. Sistemas de computação em grade podem replicar metadados sobre as tarefas em execução ou sobre os nós participantes e com isso melhorar a distribuição de tarefas aos nós. Em aplicações de comunicação e colaboração, informações de autenticação e de contatos podem ser replicadas para evitar a necessidade de um servidor central.

4.1 Replicação de metadados

Os exemplos de metadados citados acima são apenas alguns dos vários tipos de dados em redes peer-to-peer usualmente disponibilizados e mantidos somente por seus proprietários. Porém, é interessante que esses dados possam ser consultados por outros nós ou usuários. Neste domínio de aplicações a replicação single-master é indicada.

Essa decisão também simplifica o mecanismo, pois a ordenação dos comandos de escrita acontece somente no nó que recebe o comando, eliminando problemas típicos de replicação multi-master tais como detecção e resolução de conflitos. Por essas razões o escopo desse trabalho fica limitado à replicação single-master.

A característica de estruturação dos dados facilita o desenvolvimento de um mecanismo baseado em comandos de escrita e leitura aplicados sobre esses dados. Com isso

parece ser interessante optar pela técnica de transferência de operações. Esta escolha possibilita a análise semântica das operações facilitando a ordenação dos comandos e permitindo que alguns comandos semanticamente diferentes sejam executados simultaneamente, atingindo comutatividade de operações. Mas será avaliada também a possibilidade de transferência de estados.

Para transmitir as operações entre as réplicas, pode-se utilizar um mecanismo híbrido. No momento em que um nó master recebe um comando de escrita, ele imediatamente propaga o comando para as réplicas daquela informação, atuando de forma push. Se alguma réplica não recebeu o comando por não estar presente na rede naquele momento, ao retornar a rede e perceber a divergência, ela requisita ao nó master os comandos faltantes ou o estado atual, atuando da forma pull.

4.2 Exemplo de aplicação com replicação de metadados

Um exemplo das escolhas acima pode ser demonstrado evoluindo um sistema de comunicação e colaboração. Atualmente a maioria dos sistemas existentes utiliza um servidor central que autentica os usuários e mantém suas listas de contatos. Esta lista de contatos é um exemplo de metadados que poderiam estar replicados evitando a necessidade de um servidor central.

Como somente o usuário pode alterar sua lista de contatos, o mecanismo de replicação pode ser single-master. O nó onde o usuário está conectado no momento é o master. Exemplos de comandos incluem inserção de um novo contato, remoção de um contato e edição de um contato.

Ao receber um comando de escrita, o nó master propaga para todas as réplicas o comando. Todas as réplicas que estiverem vivas executam imediatamente o comando

também. Se alguma réplica não estiver viva, ao voltar a esse estado ela obtém os comandos faltantes no master.

Na ocorrência de comandos não comutativos, como por exemplo, uma inserção e uma deleção, a ordenação é feita no nó master. No caso de uma réplica receber comandos fora de ordem, ela deve ordená-los antes de executar ou obter novamente os comandos no master.

4.3 Padrão de projeto Command

Para a representação dos comandos de escrita foi utilizado o padrão de projeto Command [GAMMA, 1995]. Esse padrão especifica como encapsular uma solicitação em um objeto permitindo fazer registro (log) das solicitações.

Esse padrão tem como vantagem o isolamento entre o emissor da solicitação e os receptores da mesma. Além disso, o objeto Command pode ter um tempo de vida independente da solicitação original. Permite-se assim que um Command seja executado em um processo diferente de onde surgiu a solicitação, e em um tempo diferente.

A utilização dos registros de solicitações (log) permite uma recuperação de sistemas a partir da reexecução das solicitações armazenadas. A Figura 5 apresenta a estrutura desse padrão.

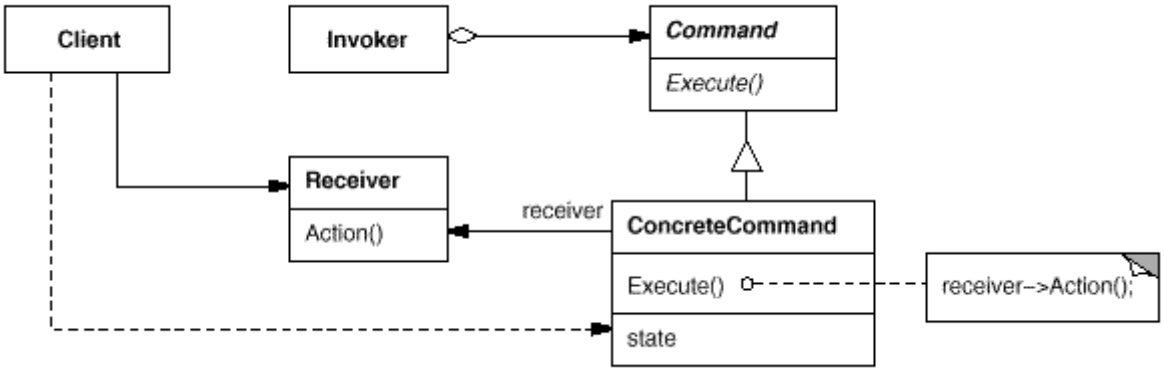

Figura 5. Estrutura do padrão de projeto Command

4.4 Análise do mecanismo

Tendo sido definido que esse trabalho terá seu escopo limitado a sistemas single-master, resta para avaliação o modelo de transferência de dados, que contém as opções de transferência de operações ou estados; e a avaliação do momento em que essas transferências serão feitas, contendo as opções de modelos push e pull.

Tentar transferir as alterações o mais rápido possível para as réplicas é interessante em aplicações sobre a arquitetura peer-to-peer. Isso usualmente minimiza o número de mensagens, evitando que as réplicas tenham que ficar consultando nós master para verificar se houve alguma alteração nos dados. Essa técnica também minimiza a divergência entre as réplicas. Porém, em sistemas com alta intensidade de escrita, pode ser interessante agrupar operações de escrita em lotes a serem enviados para as réplicas, para que o número de mensagens não cresça de forma indesejada.

Surgem então os dois primeiros algoritmos do projeto, utilizando a técnica push:

- Push com transferência de operações
- Push com transferência de estados

Porém, com a característica da arquitetura peer-to-peer de os nós estarem constantemente entrando e saindo da rede, é comum que os comandos enviados não atinjam as réplicas conforme esperado. É necessário então avaliar algoritmos com reconciliação. Optou-se nesse trabalho por executar a reconciliação sempre que um nó entra na rede, para acelerar a convergência das réplicas e consequentemente melhorar a garantia de consistência oferecida.

Surgem então os dois próximos algoritmos do projeto, utilizando a técnica pull para reconciliação:

- Push com transferência de operações e pull com transferência de estados
- Push com transferência de operações e pull com transferência de comandos

4.5 Projeto do mecanismo

Baseado na análise do mecanismo proposto define-se o modelo de classes da Figura 6.

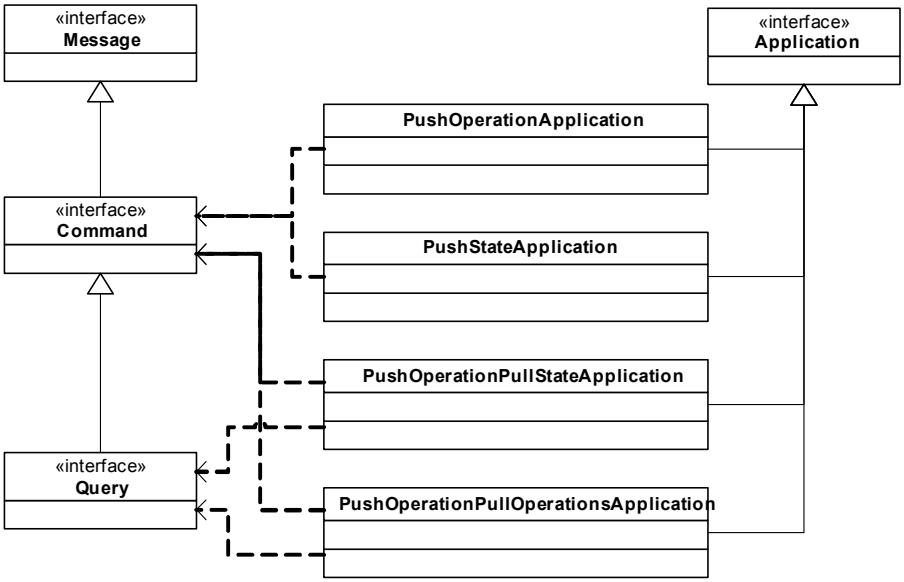

Figura 6. Diagrama de classes do projeto.

Neste modelo podemos encontrar os seguintes componentes:

Interface Message: Define qualquer mensagem que possa ser trocada entre os nós da aplicação.

Interface Command: Define qualquer comando de escrita que possa ser aplicado nos dados de uma aplicação. Segue o padrão de projeto homônimo citado acima. Essa interface estende a interface Message.

Interface Query: Define qualquer consulta, ou seja, comandos de leitura que gerem resultados. Essa interface estende a interface Command acima.

Interface Application: Define as funcionalidades comuns entre as aplicações que implementam um dos quatro algoritmos de replicação descritos nesse capítulo.

Classes Application: Definem as funcionalidades específicas de cada aplicação. Implementam a interface Application.

Capítulo 5 Simulações

Para avaliar as opções discutidas no Capítulo 4 foram desenvolvidas simulações de uma aplicação de comunicação e colaboração.

5.1 Ambiente

A aplicação foi desenvolvida sobre a plataforma Java. Essa plataforma foi escolhida por apresentar a característica de orientação a objetos, que tem se mostrado adequada para o desenvolvimento de sistemas distribuídos por sua capacidade de representar entidades complexas do mundo real e suas relações. Outra característica importante dessa plataforma é a portabilidade, obtida com a execução sobre máquinas virtuais, que é um dos motivos de seu sucesso.

O arcabouço utilizado foi o Pastry. Ele executa localização de nós e encaminhamento de mensagens em redes peer-to-peer. Esse arcabouço tem uma implementação na plataforma Java. Ele apresenta características interessantes como descentralização e tolerância à faltas no encaminhamento de mensagens. Ele ainda oferece algumas facilidades para a replicação e persistência de objetos de negócio, porém, essas funcionalidades não foram utilizadas nesse projeto.

Um protótipo utilizando comunicação por sockets foi desenvolvido para validar a aplicação. Os testes foram realizados simulando uma rede de 100 nós através de um mecanismo do arcabouço Pastry em um computador com processador Athlon 2600 e 1 GB de memória. Esse mecanismo suporta a simulação de latência na comunicação entre os nós,

porém, resultados em uma rede física real podem sofrer pequenas variações em relação aos simulados.

5.2 Projeto sobre o arcabouço Pastry

A conexão entre o projeto proposto e o arcabouço Pastry se dá através de duas das principais interfaces deste. A interface Message identifica qualquer mensagem a ser trocada entre os nós através do mecanismo de encaminhamento do arcabouço. A interface Application identifica as classes que conterão a funcionalidade de cada aplicação em cada nó.

A interface Message define somente um método, que indica qual será a prioridade dessa mensagem no mecanismo de encaminhamento. A interface Application define três métodos: forward(), deliver() e update(). O método deliver(Id, Message) é chamado quando o mecanismo de encaminhamento entrega uma mensagem para o nó rodando essa instância da aplicação, e portanto será o método onde estará concentrada boa parte desse trabalho. Os outros dois métodos são para um controle mais detalhado do arcabouço e conterão as implementações padrão.

Assim, no projeto da Figura 7, as interfaces Message e Application serão as próprias interfaces do arcabouço Pastry, e seus métodos foram incluídos nas hierarquias respectivas.

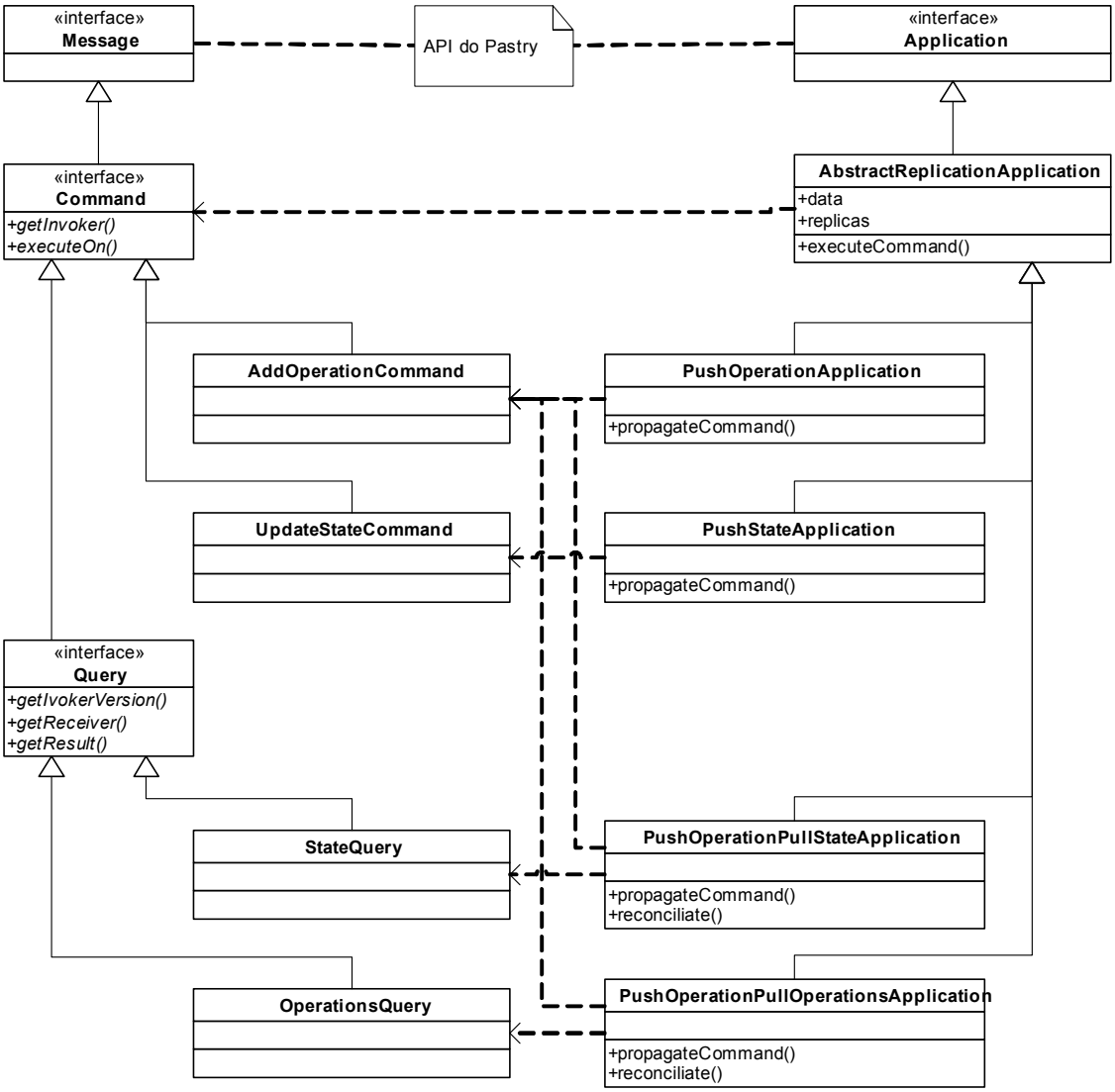

Figura 7. Diagrama de classes da implementação das simulações.

5.3 Implementação do mecanismo

Como pôde ser visto na Figura 7, o projeto é composto por duas interfaces e treze classes. Segue a estrutura de diretórios e arquivos do projeto:

- simulation
    - applications
        - AbstractReplicationOperation
        - PushOperationApplication
        - PushStateApplication
        - PushOperationPullStateApplication
        - PushOperationPullOperationsApplication
    - command
        - AddOperationCommand

        - UpdateStateCommand
    - query
        - OperationsQuery
        - StateQuery
    - Command
    - MessageCounter
    - Query
    - ReplicaCounter
    - SimulationMain
    - TextMessage

As interfaces Command e Query formam junto com a interface Message do arcabouço Pastry a hierarquia das mensagens que podem ser trocadas entre os nós da rede. A classe TextMessage é a implementação de Message, para simular mensagens comuns que são trocadas entre os nós em uma aplicação de comunicação. A classe AddOperationCommand é a implementação de Command que define uma operação de escrita, utilizada para aplicar alterações nos dados do master e na transferência de operações para as réplicas. A classe UpdateStateCommand é a implementação de Command que atualiza todos os dados do master, utilizado na transferência por estado para as réplicas.

A classe StateQuery é a implementação da interface Query que define a funcionalidade de consulta utilizada durante a reconciliação por pull de estados. A classe OperationsQuery é a implementação de Query que define a funcionalidade de consulta utilizada durante a reconciliação por pull de operações.

A classe AbstractReplicationApplication é a primeira implementação da interface Application do arcabouço Pastry. Ela contém todas as funcionalidades comuns à implementação dos quatro algoritmos para as simulações. As classes PushOperationApplication, PushStateApplication, PushOperationPullStateApplication e PushOperationPullOperationsApplications são as implementações das funcionalidades específicas dos quatro algoritmos de replicação respectivamente.

A classe SimulationMain é o ponto inicial das simulações, responsável por inicializar o ambiente e executar cada simulação. Ela utiliza as classes MessageCounter e ReplicaCounter para obter e gravar os resultados.

5.4 Nomenclatura da simulação

Para facilitar o entendimento da simulação será utilizada a seguinte nomenclatura nesse trabalho:

Mensagem: instância da interface Message, podendo ser uma mensagem comum da aplicação, uma consulta a um dado estruturado ou um comando de escrita em dados estruturados.

Comando: instância da interface Command. Contém uma operação de escrita com informações sobre o nó origem da alteração e as informações necessárias para executar a operação.

Consulta: instância da interface Query. Contém informações do requisitante e do nó destino para consulta.

Resposta: instância da interface Query. Contém as informações do requisitante da consulta, do nó onde a consulta foi executada, o resultado da consulta.

Intensidade de escrita: parâmetro da simulação que determina a chance de um usuário executar um comando ao invés de enviar uma mensagem comum ou uma consulta. Por exemplo, se a intensidade de escrita é de 10%, cada nó enviará aproximadamente 10 comandos e 90 mensagens comuns a cada 100 mensagens.

Chance de vida: parâmetro da simulação que determina a chance de um peer estar vivo, ou seja, receber e, se necessário, responder as mensagens que lhe sejam enviadas. Por exemplo, se a chance de vida for 10%, cada nó receberá aproximadamente 10 em cada 100 mensagens enviadas para ele.

5.5 Características da simulação

Para cada algoritmo testado foram feitas 64 simulações. Em cada simulação foram utilizados 100 nós, cada um enviando 100 mensagens. Os nós possuíam um número pré-determinado de réplicas esperadas. Os nós onde as réplicas esperadas estariam, para cada nó master, foram escolhidos aleatoriamente entre o conjunto completo dos nós no início da simulação. Por exemplo, se o número de réplicas esperadas é configurado em quatro, existem 500 dados replicados, (1 master + 4 réplicas) x 100 nós. Cada nó é master de seu próprio dado e quatro outros nós são escolhidos para armazenar suas réplicas. Outro exemplo pode ser visto na Figura 8.

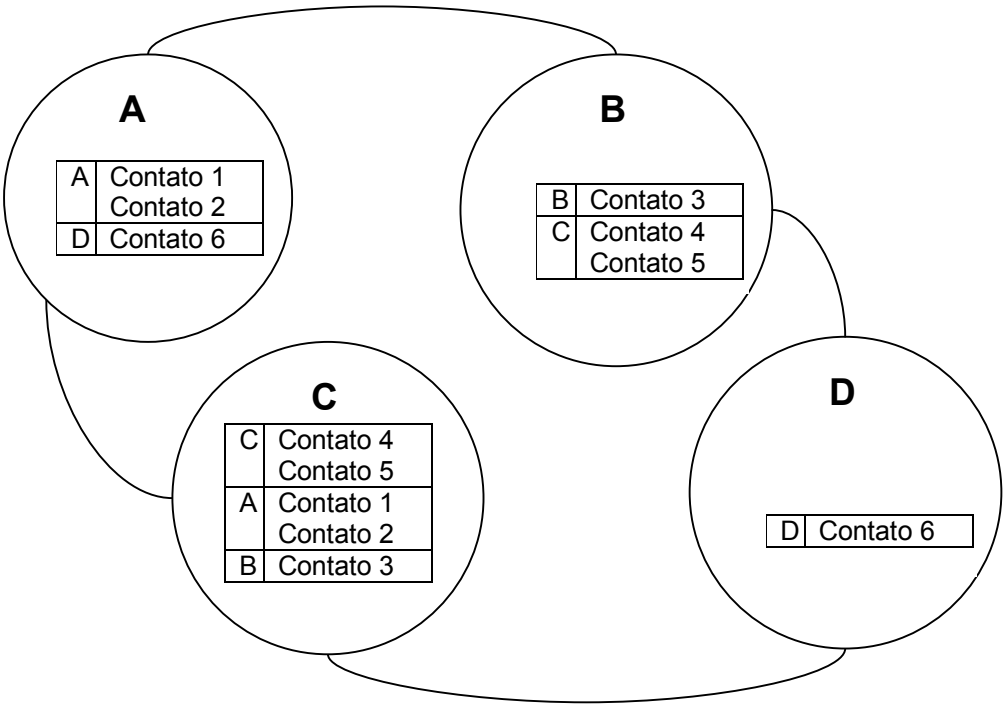

Figura 8 Exemplo de configuração com quatro nós e suas tabelas de contatos com uma réplica para cada lista de contatos.

A cada mensagem enviada, um tipo de mensagem era escolhido de acordo com o parâmetro intensidade de escrita. O recebimento das mensagens pelos nós dependia da chance de vida especificada na simulação.

Foram feitas quatro medidas durante as simulações:

- Número de mensagens trocadas entre os nós.
- Tamanho das mensagens em relação ao tamanho dos dados.
- Quantidade de réplicas efetivas, medida através da percentual em relação ao número de réplicas esperadas.
- Qualidade das réplicas efetivas, medida através da percentual de comandos aplicados no master, que foram aplicados corretamente nas réplicas.

Uma réplica é considerada efetiva quando recebe pelo menos um comando executado no master. Isso permite ao nó que contém a réplica estar ciente que dados de um outro nó estão replicados nele. Assim é possível ao nó responder consultas sobre esses dados e executar reconciliações com o nó master desses dados.

A qualidade das réplicas efetivas é a média do percentual de comandos aplicados em todas as réplicas de todos os dados. Por exemplo, se uma aplicação contém 3 nós, cada um com 1 réplica e cada um executa 5 comandos sobre seus dados, temos que cada réplica deveria ter recebido 5 comandos. Se a primeira réplica recebe 2 comandos, a segunda recebe 3 comandos e a terceira recebe 4 comandos, temos uma qualidade média das réplicas de 60%, pois cada réplica recebeu 40, 60 e 80% dos comandos aplicados nos respectivos nós master.

Essas medidas foram tomadas variando-se:

- Número de réplicas esperadas no conjunto 1, 2, 4 e 8.
- Chance de vida no conjunto 10, 20, 40 e 80%.

- Intensidade de escrita no conjunto 10, 20, 40 e 80%.

As medidas de quantidade e qualidade de réplicas efetivas foram feitas imediatamente após os ciclos de envio de mensagens, evitando a convergência das réplicas nos algoritmos com reconciliação, permitindo assim uma melhor comparação entre os algoritmos com e sem reconciliação.

5.6 Simulação com Push de Operações

Na primeira simulação o algoritmo de replicação experimentado era push de operações. Ou seja, a cada execução de um comando em um master, esse comando era enviado pelo master para a lista de réplicas.

Foi considerado que os comandos são comutáveis, ou seja, podem ser aplicados em qualquer ordem. Isso é necessário, pois nesse algoritmo não há envio ou reconciliações de estado. Assim, há a possibilidade de perda da execução de algumas operações de escrita. Para possibilitar a comparação com os outros algoritmos, que não sofrem dessa desvantagem, a consideração de comandos comutáveis foi assumida.

Assim, as funcionalidades no master e na réplica ficam assim:

Master, ao receber uma requisição de operação de escrita:

```
executarComando(Comando comando) {
        executarComandoLocalmente(comando);
        para cada réplica na lista de réplicas {
                encaminharComandoParaRéplica(comando, réplica);
        }
}
```

Réplicas, ao receber uma transferência de operação de escrita:

```
receberComando(Comando comando) {
        executarComandoLocalmente(comando);
}
```

Comando de operação, de execução no master e de transferência para as réplicas:

```
No dono;
```

```
Objeto item;
Objeto novoItem;
executar(ConjuntoDeDados dados) {
        dadosDono = dados.buscaDadosDono(dono);
        itemDadosDono = dadosDono.buscaItemDadosDono(item);
        itemDadosDono.alterar(novoItem);
}
```

As classes utilizadas nessa simulação foram PushOperationApplication e AddOperationCommand que estão descritas nos apêndices A.6 e A.2.

5.7 Simulação com Push de Estados

Na segunda simulação o algoritmo de replicação experimentado foi o push de estados. Ou seja, a cada execução de um comando em um master, um comando de atualização de estado era criado e enviado para a lista de réplicas. Esse mecanismo aceita comandos não comutativos.

A funcionalidade de execução de um comando de operação de escrita no master sofre uma pequena alteração, com a criação de um novo comando de atualização de estado que será enviado para a lista de réplicas. Essa alteração e o novo comando estão descritos abaixo, a funcionalidade de recebimento da transferência na réplica e o comando de operação não mudam em relação ao primeiro algoritmo.

Master, ao receber uma requisição de operação de escrita:

```
executarComando(Comando comando) {
        executarComandoLocalmente(comando);
        comandoEstado = criaComandoEstado();
        para cada réplica na lista de réplicas {
                encaminharComandoParaRéplica(comandoEstado, réplica);
        }
}
```

Comando de estado, para transferência para as réplicas:

```
No dono;
Objeto dadosDono;
executar(ConjuntoDeDados dados) {
        dadosDono = dados.buscaDadosDono(dono);
        dadosDono.substituir(dadosDono);
}
```

As classes utilizadas nessa simulação foram PushStateApplication, AddOperationCommand e UpdateStateCommand que estão descritas nos apêndices A.9, A.2 e A.15 respectivamente.

5.8 Simulação com Push de Operações e Pull de Estados

Na terceira simulação o algoritmo de replicação experimentado foi push de operações e pull de estados. Ou seja, a cada execução de uma operação de escrita em um master, um comando contendo essa operação era enviado para a lista de réplicas. E toda a vez que um nó contendo réplicas voltasse à vida, ele consultaria nos respectivos nós master de cada réplica mantida nele o estado completo dos dados replicados nele. Esse mecanismo também aceita comandos não comutativos.

Assim, a funcionalidade no master é idêntica à da primeira simulação. A réplica tem sua funcionalidade evoluída para reconciliar. Essa funcionalidade está descrita a seguir, bem como a funcionalidade da consulta utilizada.

Réplicas, funcionalidade de receber comando e de reconciliar:

```
receberComando(Comando comando) {
        executarComandoLocalmente(comando);
}
reconciliar() { // Metodo executado ao voltar a vida
        para cada replicacao no conjunto de replicações neste nó {
                Resposta resposta = consultarMaster(replicacao.master());
                Comando comandoEstado = novo Comando(resposta);
                executarComandoLocalmente(comandoEstado);
        }
}
```

Consulta, executada durante a reconciliação:

```
No dono;
Objeto resposta;
executar(ConjuntoDeDados dados) {
        resposta = dados.buscaDadosDono(dono);
}
resposta(No dono) {
        retorna resposta;
}
```

As classes utilizadas nessa simulação foram PushOperationPullStateApplication, AddOperationCommand e StateQuery que estão descritas nos apêndices A.8, A.2 e A.13 respectivamente.

5.9 Simulação com Push de Operações e Pull de Operações

Na quarta simulação o algoritmo de replicação utilizado foi push de operações e pull de operações. Ou seja, a cada execução de um comando em um nó master, esse comando era enviado para a lista de réplicas dele. E toda a vez que um nó contendo réplicas voltasse à vida, ele consultaria nos respectivos nós master de cada réplica mantida nele os comandos faltantes para os dados replicados nele. Esse mecanismo também aceita comandos não comutativos.

Nesse algoritmo, cada master precisa armazenar a lista de comandos executados sobre ele. E as réplicas ao reconciliar precisam saber reaplicar os comandos faltantes sobre os dados replicados. As outras funcionalidades são idênticas as já apresentadas.

Master, ao receber uma requisição de operação de escrita:

```
executarComando(Comando comando) {
        executarComandoLocalmente(comando);
        armazenarComando(comando);
        para cada réplica na lista de réplicas {
                encaminharComandoParaRéplica(comando, réplica);
        }
}
```

Réplicas, funcionalidade de receber comando e de reconciliar:

```
reconciliar() { // Metodo executado ao voltar a vida
        para cada replicacao no conjunto de replicações neste nó {
                Resposta resposta = consultarMaster(replicacao.master());
                Para cada comando na resposta {
                        executarComandoLocalmente(comandoEstado);
                }
        }
}
```

As classes utilizadas nessa simulação foram PushOperationPullOperationsApplication, AddOperationCommand e OperationsQuery que estão descritas nos apêndices A.7, A.2 e A.5 respectivamente.

Capítulo 6 Resultados da Simulação

6.1 Simulação com Push de Operações

A primeira simulação tem um algoritmo muito simples. Porém, apresenta a desvantagem de suportar somente comandos comutáveis, pois não executa qualquer atualização das suas replicações ao voltar à vida.

6.1.1 Tabelas de resultados

Seguem as quatro tabelas com os resultados completos da simulação com push de operações. Na Tabela 3 encontra-se as quantidades de mensagens, na Tabela 4 o tamanho médio das mensagens, na Tabela 5 as quantidades de réplicas efetivas e na Tabela 6 a qualidade das réplicas efetivas.

Tabela 3. Número de mensagens para a primeira simulação.

	Escrita 10%				**Escrita 20%**			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	10000	10000	10000	10000	10000	10000	10000	10000
2 réplicas	10981	10994	10964	11000	11985	11955	11943	11982
4 réplicas	12786	13185	12930	12925	15901	15743	15853	16007
8 réplicas	16594	16822	16462	16581	23553	23487	23694	23454
	Escrita 40%				**Escrita 80%**			
1 réplica	10000	10000	10000	10000	10000	10000	10000	10000
2 réplicas	13953	13964	13850	13975	17885	17883	17816	17694
4 réplicas	21784	21638	21998	21710	33167	33333	33637	33551
8 réplicas	36391	37362	36854	36678	63140	62882	63848	63939

Tabela 4. Tamanho médio das mensagens para a primeira simulação.

	Escrita 10%				Escrita 20%			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	1	1	1	1	1	1	1	1
2 réplicas	1	1	1	1	1	1	1	1
4 réplicas	1	1	1	1	1	1	1	1
8 réplicas	1	1	1	1	1	1	1	1
	Escrita 40%				**Escrita 80%**			
1 réplica	1	1	1	1	1	1	1	1
2 réplicas	1	1	1	1	1	1	1	1
4 réplicas	1	1	1	1	1	1	1	1
8 réplicas	1	1	1	1	1	1	1	1

Tabela 5. Número de réplicas efetivas por esperadas para a primeira simulação.

	Escrita 10%				Escrita 20%			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	65,00%	91,00%	98,00%	100,00%	86,00%	98,00%	100,00%	100,00%
2 réplicas	64,00%	85,00%	98,00%	100,00%	90,00%	99,50%	100,00%	100,00%
4 réplicas	59,00%	88,50%	97,75%	100,00%	86,75%	98,00%	100,00%	100,00%
8 réplicas	64,25%	88,00%	97,50%	100,00%	87,13%	98,50%	100,00%	100,00%
	Escrita 40%				**Escrita 80%**			
1 réplica	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
2 réplicas	99,50%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
4 réplicas	98,75%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
8 réplicas	98,88%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%

Tabela 6. Qualidade das réplicas efetivas para a primeira simulação.

	Escrita 10%				Escrita 20%			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	18,41%	23,28%	43,91%	80,97%	12,72%	21,58%	41,48%	80,38%
2 réplicas	16,82%	23,85%	41,30%	80,89%	13,01%	20,94%	40,90%	79,85%
4 réplicas	16,42%	24,29%	39,35%	79,03%	13,33%	20,92%	39,29%	79,40%
8 réplicas	17,34%	23,31%	40,22%	77,38%	12,03%	20,14%	39,41%	77,71%
	Escrita 40%				**Escrita 80%**			
1 réplica	12,49%	23,20%	41,89%	80,27%	12,02%	21,38%	39,82%	80,36%
2 réplicas	11,72%	21,08%	39,42%	79,80%	10,73%	19,62%	40,77%	78,72%
4 réplicas	10,94%	19,55%	39,54%	79,01%	10,22%	20,42%	39,98%	78,90%
8 réplicas	10,44%	20,37%	39,70%	77,47%	10,35%	19,56%	39,60%	77,80%

6.1.2 Gráficos de resultados

Os dados coletados durante a simulação indicam que esse algoritmo apresenta o menor número possível de mensagens trocadas entre os nós (Figura 9), pois não há reconciliações que envolvem novas trocas de mensagens. O número de mensagens é independente da chance de vida dos nós, mas aumenta com o número de réplicas, e principalmente, com a intensidade de escrita. Isso ocorre, pois cada comando executado em um nó master é enviado para todas as suas réplicas.

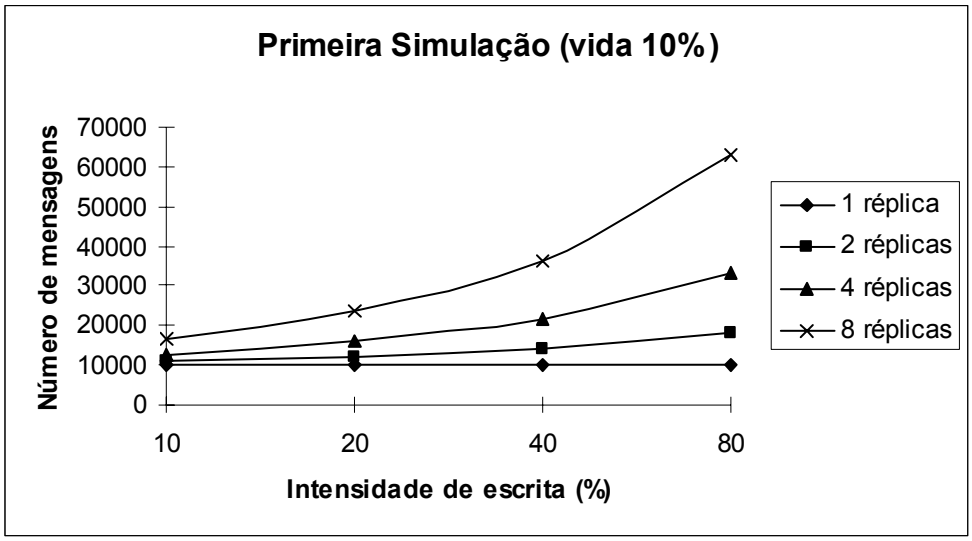

Figura 9. Número de mensagens, por intensidade de escrita e número de réplicas, para a primeira simulação com chance de vida de 10%.

Esse algoritmo também apresenta menor tamanho médio de mensagens, pois todas as mensagens têm exatamente o tamanho dos dados da aplicação (Tabela 4). A quantidade de réplicas efetivas é boa, e não depende do número esperado de réplicas. Mas, como era esperado, aumenta com a chance de vida dos nós, pois para uma réplica se tornar efetiva ela precisa receber pelo menos um comando de seu master. Pode-se perceber também que a quantidade de réplicas aumenta com a intensidade de escrita. Isso acontece, pois com mais

comandos sendo trocados entre os nós, aumenta-se a chance de um nó ser informado que é réplica de outro. (Figura 10).

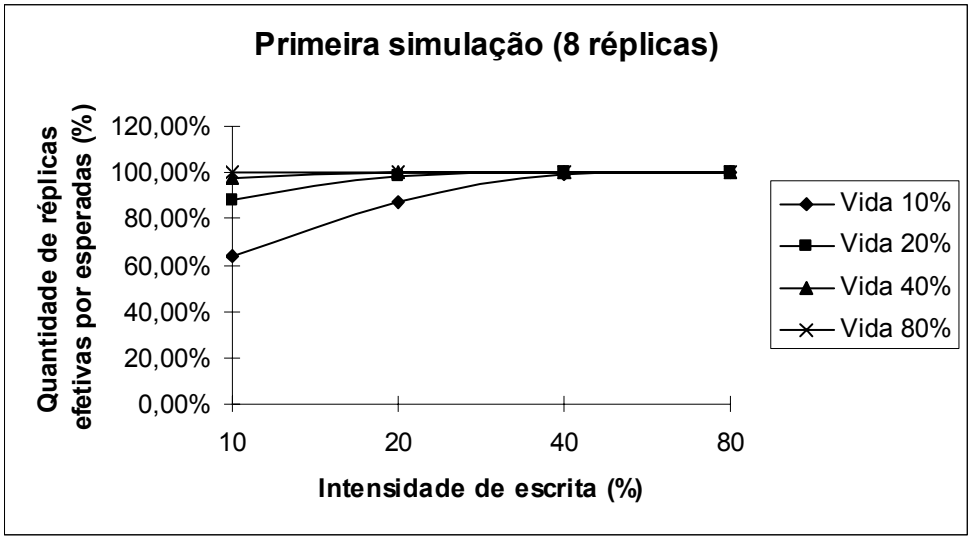

Figura 10. Quantidade de réplicas efetivas por esperadas, por chance de vida e intensidade de escrita, para a primeira simulação com 8 réplicas.

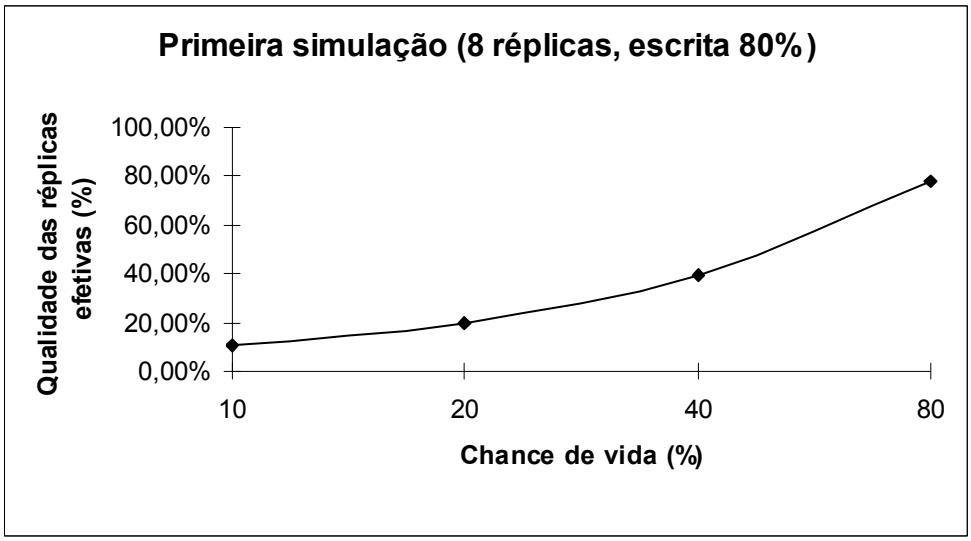

Figura 11. Qualidade das réplicas efetivas por número de réplicas, para a primeira simulação com 8 réplicas e intensidade de escrita de 80%.

Porém a qualidade das réplicas é muito dependente da chance de vida dos nós (Figura 11), pois cada comando carrega dados parciais. A qualidade é independente do número de

réplicas e da intensidade de escrita. Isso mostra que esse algoritmo só deve ser utilizado em situações onde todos os comandos são comutáveis e os nós têm uma chance de vida muito elevada.

6.2 Simulação com Push de Estados

A segunda simulação tem uma implementação simples também. E apresenta a vantagem de suportar a execução de comandos comutáveis no master, mesmo não executando atualizações das replicações.

6.2.1 Tabelas de resultados

Seguem as quatro tabelas com os resultados completos da simulação com push de operações. Na Tabela 7 encontra-se as quantidades de mensagens, na Tabela 8 tamanho médio das mensagens, na Tabela 9 as quantidades de réplicas efetivas e na Tabela 10 a qualidade das réplicas efetivas.

Tabela 7. Número de mensagens para segunda simulação

	Escrita 10%				**Escrita 20%**			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	10000	10000	10000	10000	10000	10000	10000	10000
2 réplicas	10956	11018	11021	10997	11987	12007	11908	12010
4 réplicas	12875	12948	13072	12912	16091	15808	15883	16055
8 réplicas	16640	16677	16963	16416	23600	23509	23775	23800
	Escrita 40%				**Escrita 80%**			
1 réplica	10000	10000	10000	10000	10000	10000	10000	10000
2 réplicas	14006	14022	13971	13948	17967	17972	17973	17965
4 réplicas	21692	21851	21465	21975	33160	33364	33934	33344
8 réplicas	36227	36758	37149	37232	63519	63779	63722	63133

Tabela 8. Tamanho médio das mensagens para a segunda simulação.

	Escrita 10%				Escrita 20%			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	1,4820	1,4843	1,4971	1,5055	2,8351	2,9168	3,0673	2,8755
2 réplicas	1,8384	1,9348	1,9335	1,9057	4,3100	4,3799	4,1851	4,3593
4 réplicas	2,4410	2,5192	2,6528	2,5355	6,2147	5,8511	5,8872	6,1209
8 réplicas	3,2161	3,2413	3,3395	3,1221	7,5960	7,4822	7,7277	7,7648
	Escrita 40%				**Escrita 80%**			
1 réplica	8,9348	8,8877	8,4471	8,8220	32,3718	33,0086	32,5220	32,7009
2 réplicas	12,3330	12,5991	12,2243	12,1919	35,9683	36,0130	36,0123	35,9541
4 réplicas	15,4157	15,6408	15,1141	15,8046	38,1279	38,4033	38,2883	38,0660
8 réplicas	17,1181	17,2939	17,7396	17,6652	39,1125	39,2735	39,3759	38,9465

Tabela 9. Número de réplicas efetivas por esperadas para a segunda simulação.

	Escrita 10%				Escrita 20%			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	67,00%	85,00%	100,00%	100,00%	89,00%	100,00%	100,00%	100,00%
2 réplicas	63,50%	85,50%	99,50%	99,50%	87,50%	98,00%	100,00%	100,00%
4 réplicas	65,50%	89,25%	98,25%	100,00%	85,25%	97,75%	100,00%	100,00%
8 réplicas	60,88%	85,75%	99,38%	100,00%	88,75%	97,75%	100,00%	100,00%
	Escrita 40%				**Escrita 80%**			
1 réplica	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
2 réplicas	97,50%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
4 réplicas	98,25%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
8 réplicas	97,38%	99,88%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%

Tabela 10. Qualidade das réplicas efetivas para a segunda simulação.

	Escrita 10%				Escrita 20%			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	60,89%	66,48%	86,19%	96,96%	67,89%	82,38%	91,33%	98,42%
2 réplicas	65,43%	70,82%	85,07%	97,60%	66,83%	82,13%	90,42%	98,58%
4 réplicas	65,73%	75,25%	84,35%	94,79%	65,33%	79,37%	91,91%	97,14%
8 réplicas	60,82%	70,48%	83,83%	93,94%	68,81%	80,03%	89,38%	95,67%
	Escrita 40%				**Escrita 80%**			
1 réplica	77,23%	89,00%	95,14%	99,44%	87,93%	95,13%	97,81%	99,62%
2 réplicas	79,91%	89,29%	95,49%	98,79%	89,47%	94,86%	97,97%	99,71%
4 réplicas	78,98%	89,33%	93,42%	97,73%	86,86%	92,63%	98,02%	98,02%
8 réplicas	76,54%	87,29%	93,10%	96,53%	86,86%	92,63%	98,02%	98,02%

6.2.2 Gráficos de resultados

Os dados coletados durante essa simulação indicam que esse algoritmo mantém o mesmo número de mensagens trocadas entre os nós que a primeira simulação (Figura 12), pois também não ocorrem reconciliações.

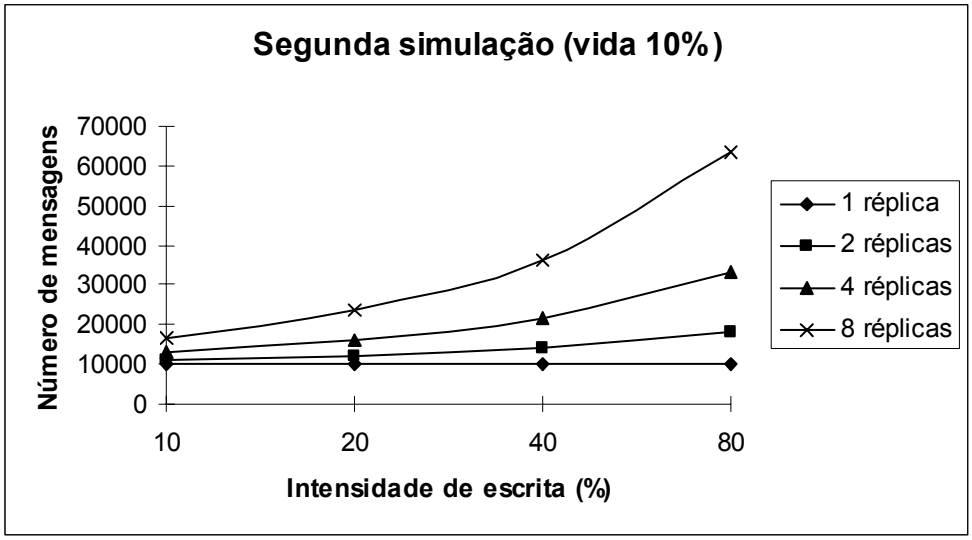

Figura 12. Número de mensagens por intensidade de escrita e número de réplicas, para a segunda simulação com chance de vida de 10%.

Porém, há uma diferença significativa no tamanho médio das mensagens. Ele aumenta com a intensidade de escrita, pois mais comandos são enviados e o tamanho dos comandos é usualmente maior que o tamanho das mensagens comuns nesse algoritmo. A implementação do segundo algoritmo nesse trabalho mostra que uma mensagem pode atingir até 40 vezes o tamanho dos dados (Figura 13).

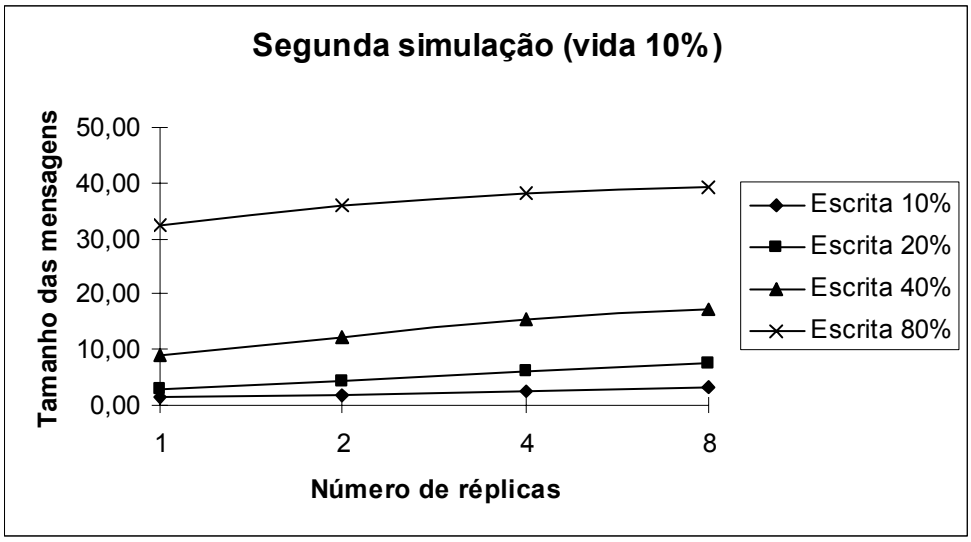

Figura 13. Tamanho médio das mensagens, por número de réplicas e intensidade de escrita, para a segunda simulação com chance de vida 10%.

A quantidade de réplicas efetivas também permanece a mesma em relação à primeira simulação (Figura 14).

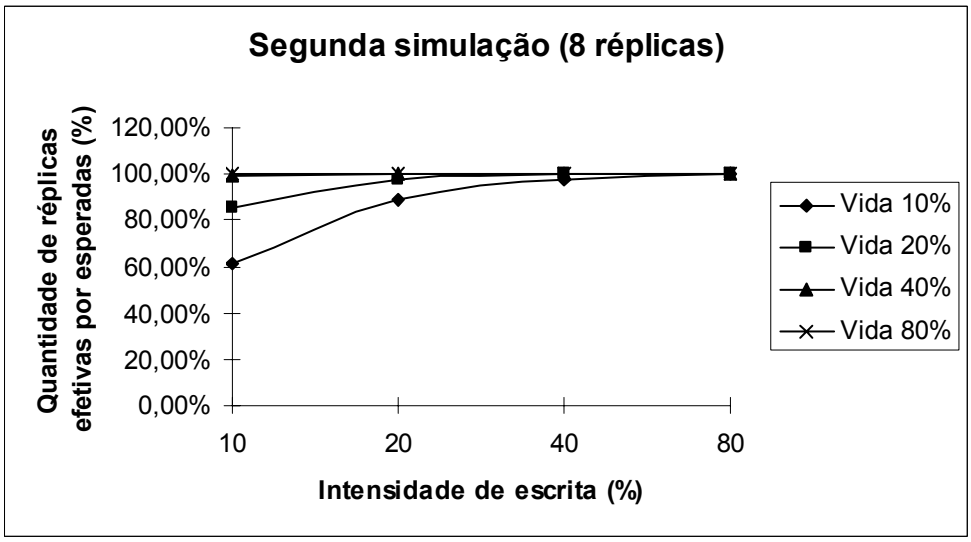

Figura 14. Quantidade de réplicas efetivas por esperadas, por intensidade de escrita e chance de vida, para a segunda simulação com 8 réplicas.

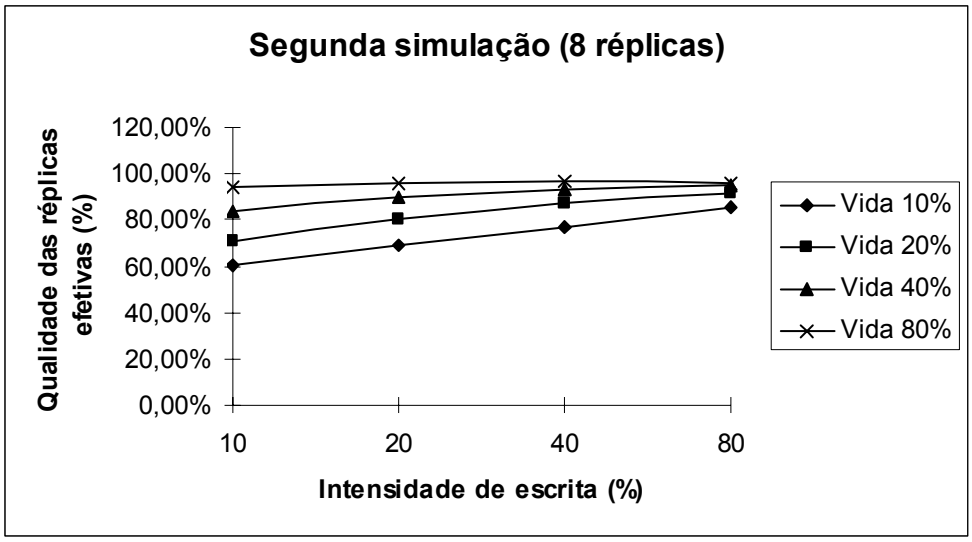

Figura 15. Qualidade das réplicas efetivas, por intensidade de escrita e chance de vida, para a segunda simulação com 8 réplicas.

Porém a qualidade das réplicas melhora substancialmente (Figura 15). Mesmo o pior caso desse algoritmo é significativamente melhor que o primeiro algoritmo, pois a cada comando recebido por uma réplica o dado completo é atualizado. E a qualidade das réplicas efetivas aumenta com a chance de vida dos nós e com a intensidade de escrita da aplicação, pois assim aumentam-se as chances de uma réplica receber um comando enviado por seu master.

Conclui-se que esse algoritmo pode ser utilizado ao invés do primeiro para obter-se uma qualidade maior das réplicas efetivas se o aumento do tamanho médio das mensagens não for um problema. Isso pode ser obtido em sistemas com tamanho total dos dados de cada nó pequeno ou com pouca intensidade de escrita.

6.3 Simulação com Push de Operações e Pull de Estados

A terceira simulação é mais complexa que as duas primeiras e continua com o suporte a comandos não comutáveis.

6.3.1 Tabelas de resultados

Seguem as quatro tabelas com os resultados completos da simulação com push de operações. Na Tabela 11 encontra-se as quantidades de mensagens, na Tabela 12 tamanho médio das mensagens, na Tabela 13 as quantidades de réplicas efetivas e na Tabela 14 a qualidade das réplicas efetivas.

Tabela 11. Número de mensagens para a terceira simulação.

	Escrita 10%				**Escrita 20%**			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	10496	11397	13145	13304	10687	11825	13956	13457
2 réplicas	12132	14309	18438	17820	13945	16626	21340	20569
4 réplicas	14814	18729	28652	27716	20499	27532	38937	36795
8 réplicas	22617	33806	57117	54517	35559	52634	84908	83286
	Escrita 40%				**Escrita 80%**			
1 réplica	11180	12547	14874	14316	11670	13348	16352	15928
2 réplicas	16540	20433	26895	24865	22632	28405	38326	35084
4 réplicas	29099	40650	57756	53083	48836	64814	93818	84860
8 réplicas	60529	96347	142974	132985	113683	170943	259580	221662

Tabela 12. Tamanho médio das mensagens para a terceira simulação.

	Escrita 10%				**Escrita 20%**			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	1,0287	1,1042	1,2491	1,1506	1,0579	1,2704	1,6953	1,5161
2 réplicas	1,0539	1,2196	1,4367	1,2282	1,1338	1,5097	2,0445	1,7168
4 réplicas	1,0670	1,2601	1,5395	1,2525	1,2377	1,7476	2,3158	1,8194
8 réplicas	1,1417	1,4154	1,6600	1,2901	1,3292	1,9233	2,3831	1,7782
	Escrita 40%				**Escrita 80%**			
1 réplica	1,2309	1,6456	2,6641	2,6859	1,5757	2,6613	5,4168	6,4043
2 réplicas	1,2973	2,0878	3,3637	3,0889	1,7335	3,5178	6,5499	6,6386
4 réplicas	1,5224	2,5701	3,9622	3,0198	2,2560	4,2239	7,4071	6,0424
8 réplicas	1,8045	3,0395	4,0850	2,8424	2,6888	5,1094	7,8844	5,3001

Tabela 13. Número de réplicas efetivas por esperadas para a terceira simulação.

	Escrita 10%				Escrita 20%			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	65,00%	88,00%	99,00%	100,00%	86,00%	98,00%	100,00%	100,00%
2 réplicas	67,50%	92,00%	97,50%	100,00%	92,00%	98,00%	100,00%	100,00%
4 réplicas	63,00%	83,00%	98,75%	100,00%	86,75%	99,00%	99,75%	100,00%
8 réplicas	67,38%	87,00%	98,88%	100,00%	88,50%	97,88%	100,00%	100,00%
	Escrita 40%				**Escrita 80%**			
1 réplica	99,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
2 réplicas	97,50%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
4 réplicas	98,75%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
8 réplicas	98,25%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%

Tabela 14. Qualidade das réplicas efetivas para a terceira simulação.

	Escrita 10%				Escrita 20%			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	8,57%	32,85%	83,17%	98,71%	7,54%	29,62%	81,79%	99,02%
2 réplicas	7,76%	37,63%	86,08%	97,65%	8,24%	38,22%	83,13%	97,73%
4 réplicas	8,78%	31,98%	84,64%	97,55%	9,22%	39,42%	87,02%	97,78%
8 réplicas	10,08%	37,45%	84,42%	96,40%	9,92%	50,00%	88,47%	96,18%
	Escrita 40%				**Escrita 80%**			
1 réplica	8,91%	35,94%	84,12%	98,64%	7,79%	37,92%	89,05%	99,12%
2 réplicas	8,58%	44,84%	86,16%	98,49%	8,68%	55,31%	90,68%	98,46%
4 réplicas	10,38%	53,40%	90,25%	97,81%	17,76%	64,65%	93,58%	98,23%
8 réplicas	15,08%	64,43%	90,57%	96,65%	27,49%	77,26%	94,37%	95,23%

6.3.2 Gráficos de resultados

Os resultados mostram que o tamanho das mensagens é intermediário, quando comparado com os outros dois algoritmos (Figura 16). Mesmo no pior caso, que acontece com uma chance de vida intermediária e uma grande intensidade de escrita, o tamanho médio das mensagens não ultrapassa oito vezes o tamanho dos dados de um nó. Com chances de vida menores, os nós fazem poucas reconciliações e por isso o tamanho médio das mensagens permanece pequeno. Com chances de vida maiores, os nós também fazem poucas reconciliações, diminuindo o tamanho médio das mensagens, porém a diferença aparece na qualidade das réplicas.

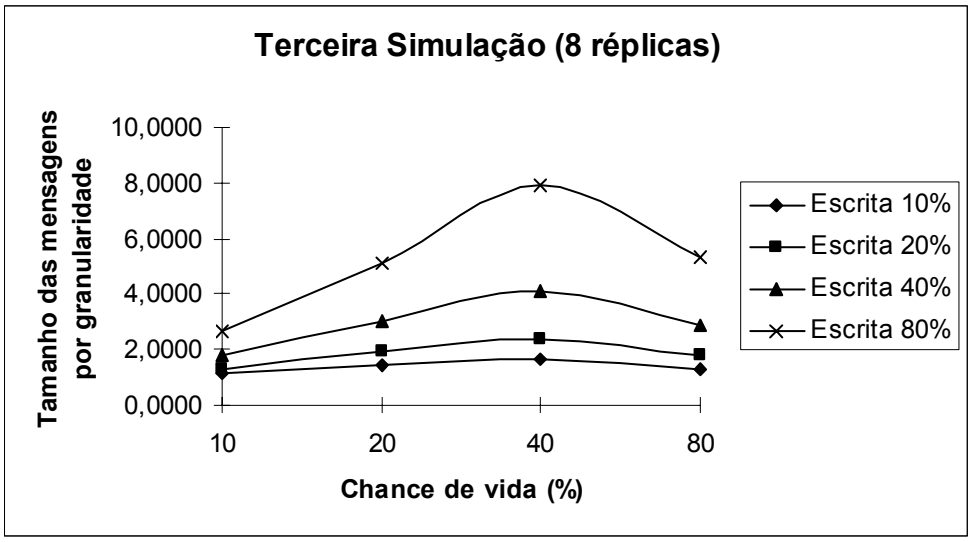

Figura 16. Tamanho médio das mensagens, por chance de vida e intensidade de escrita, para a terceira simulação com 8 réplicas.

Apesar dos ganhos significativos no tamanho das mensagens, há um aumento considerável do número de mensagens (Figura 17). Esse aumento é acentuado pelo número de réplicas e pela intensidade de escrita. Uma chance de vida intermediária é o pior caso para o número de mensagens por questões semelhantes às descritas acima na explicação de tamanho de mensagens.

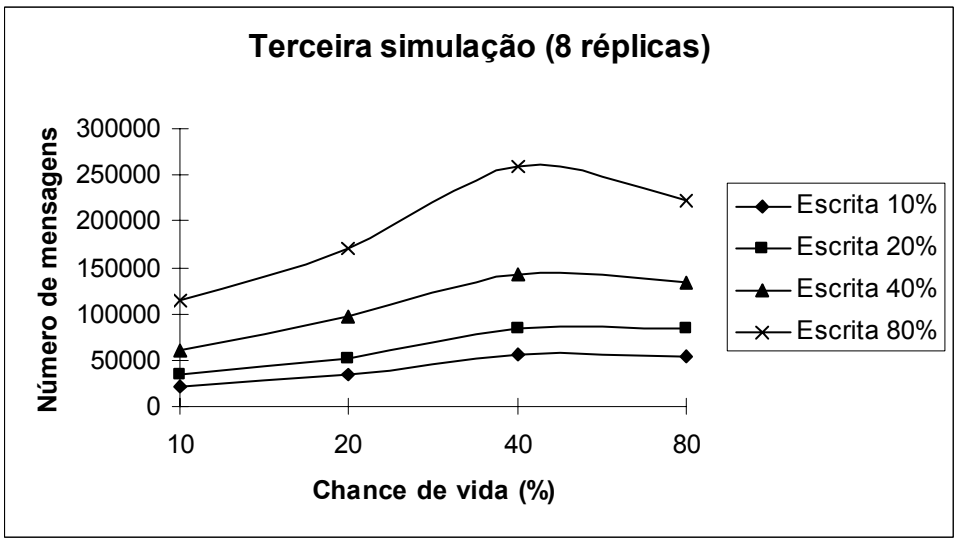

Figura 17. Número de mensagens por chance de vida e intensidade de escrita, para a terceira simulação com 8 réplicas.

A quantidade de réplicas efetivas continua inalterada (Figura 18).

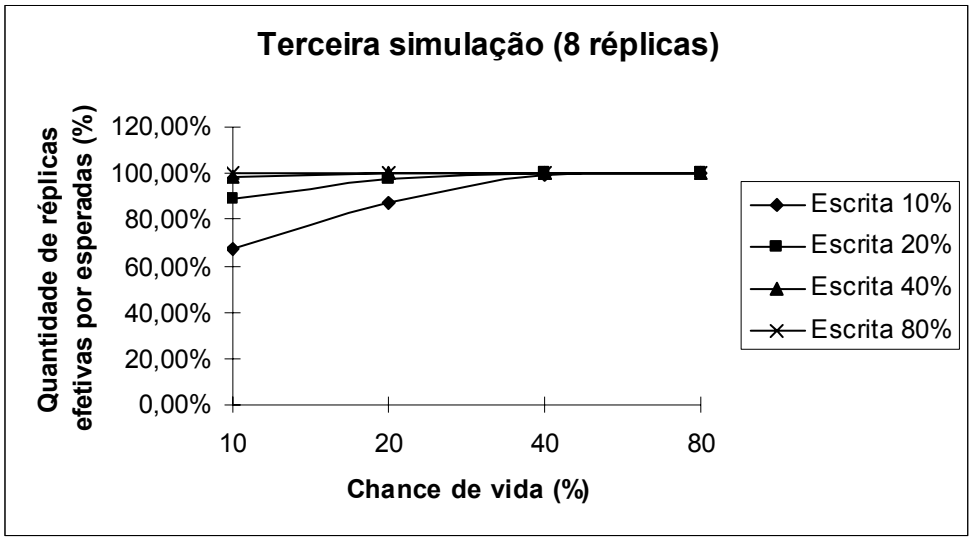

Figura 18. Quantidade de réplicas efetivas por esperadas, por chance de vida e intensidade de escrita, para a terceira simulação com 8 réplicas.

A qualidade das réplicas efetivas com uma chance de vida pequena é semelhante ao primeiro algoritmo (Figura 19), porém com o aumento da chance de vida a qualidade melhora, igualando o segundo algoritmo a partir de uma chance de vida próxima a 40%.

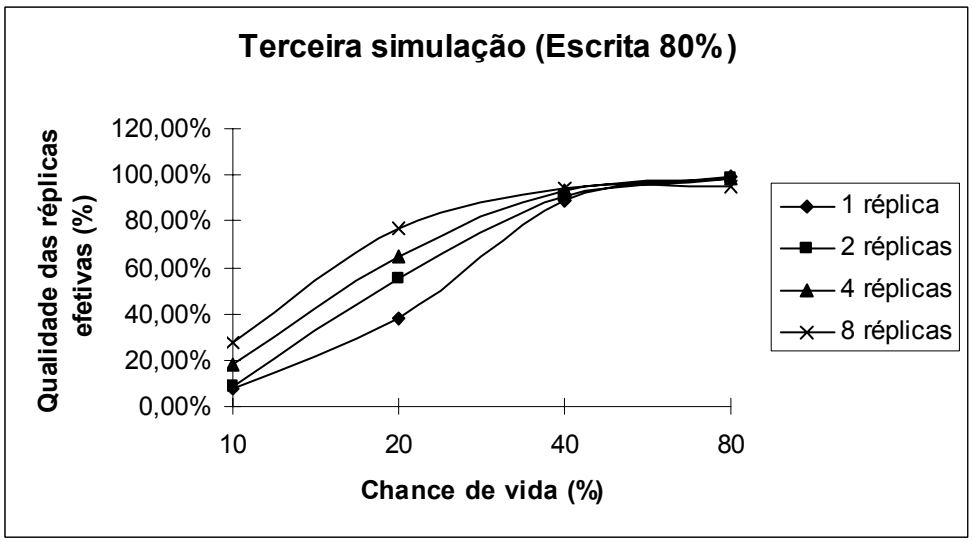

Figura 19. Qualidade das réplicas efetivas, por chance de vida e número de réplicas, para a terceira simulação com intensidade de escrita de 80%.

6.4 Simulação com Push de Operações e Pull de Operações

A quarta simulação é a mais complexa desse trabalho. Ela também oferece suporte a comandos não comutáveis.

6.4.1 Tabelas de resultados

Seguem as quatro tabelas com os resultados completos da simulação com push de operações. Na Tabela 15 encontra-se as quantidades de mensagens, na Tabela 16 tamanho médio das mensagens, na Tabela 17 as quantidades de réplicas efetivas e na Tabela 18 a qualidade das réplicas efetivas.

Tabela 15. Número de mensagens para a quarta simulação.

	Escrita 10%				**Escrita 20%**			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	10445	11277	13410	13265	10713	11995	14001	13626
2 réplicas	12116	14038	17678	17642	13655	16493	20950	21001
4 réplicas	15390	19336	28955	28121	20465	26861	39566	36358
8 réplicas	22779	33222	54061	53211	34870	54021	88399	78586
	Escrita 40%				**Escrita 80%**			
1 réplica	11167	12438	15460	14398	11663	13235	16152	15765
2 réplicas	16745	20172	27262	25284	22433	28339	39948	36224
4 réplicas	29605	39455	57822	51641	49012	64561	89463	84680
8 réplicas	62216	89241	139910	130110	111205	172201	263309	224632

Tabela 16. Tamanho médio das mensagens para a quarta simulação.

	Escrita 10%				**Escrita 20%**			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	1,0215	1,0783	1,0905	1,0109	1,0805	1,2596	1,2341	1,0444
2 réplicas	1,0504	1,1494	1,1338	1,0221	1,1189	1,3534	1,3093	1,0629
4 réplicas	1,0789	1,2250	1,1685	1,0239	1,1863	1,5543	1,2988	1,0670
8 réplicas	1,1394	1,2925	1,1718	1,0280	1,3293	1,6041	1,2873	1,0707
	Escrita 40%				**Escrita 80%**			
1 réplica	1,1854	1,4712	1,4785	1,1254	1,5527	2,0812	1,9455	1,3645
2 réplicas	1,3065	1,6912	1,5558	1,1692	1,6828	2,3074	1,8327	1,3813
4 réplicas	1,5037	1,9232	1,5502	1,1759	2,0650	2,4545	1,7838	1,3346
8 réplicas	1,7422	1,9342	1,4228	1,1321	2,3218	2,3434	1,5569	1,2346

Tabela 17. Número de réplicas efetivas por esperadas para a quarta simulação.

	Escrita 10%				Escrita 20%			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	62,00%	87,00%	100,00%	100,00%	85,00%	99,00%	100,00%	100,00%
2 réplicas	68,50%	88,50%	98,50%	100,00%	87,00%	99,00%	100,00%	100,00%
4 réplicas	63,75%	86,50%	97,75%	100,00%	88,75%	98,00%	100,00%	100,00%
8 réplicas	69,00%	86,63%	98,38%	100,00%	86,63%	98,88%	100,00%	100,00%
	Escrita 40%				**Escrita 80%**			
1 réplica	96,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
2 réplicas	98,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
4 réplicas	98,50%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%
8 réplicas	98,63%	99,88%	100,00%	100,00%	100,00%	100,00%	100,00%	100,00%

Tabela 18. Qualidade das réplicas efetivas para a quarta simulação.

	Escrita 10%				Escrita 20%			
	Vida 10%	**Vida 20%**	**Vida 40%**	**Vida 80%**	**Vida 10%**	**Vida 20%**	**Vida 40%**	**Vida 80%**
1 réplica	7,28%	30,94%	77,83%	93,82%	7,06%	38,66%	78,93%	94,35%
2 réplicas	8,68%	38,57%	76,96%	94,23%	9,15%	34,06%	80,39%	94,39%
4 réplicas	6,49%	34,60%	79,80%	94,06%	6,89%	39,99%	83,80%	92,96%
8 réplicas	7,62%	37,65%	79,36%	91,30%	11,60%	47,78%	85,48%	91,31%
	Escrita 40%				**Escrita 80%**			
1 réplica	4,54%	40,25%	81,59%	94,48%	8,30%	38,94%	85,20%	94,79%
2 réplicas	7,64%	41,40%	82,20%	94,14%	10,75%	51,85%	87,94%	94,53%
4 réplicas	12,14%	51,61%	86,33%	93,21%	20,03%	62,12%	89,16%	95,03%
8 réplicas	15,83%	59,02%	87,06%	92,46%	24,76%	74,11%	91,34%	92,09%

6.4.2 Gráficos de resultados

Os resultados mostram que o número de mensagens é semelhante ao terceiro algoritmo (Figura 20), pois ambos os algoritmos utilizam reconciliações semelhantes.

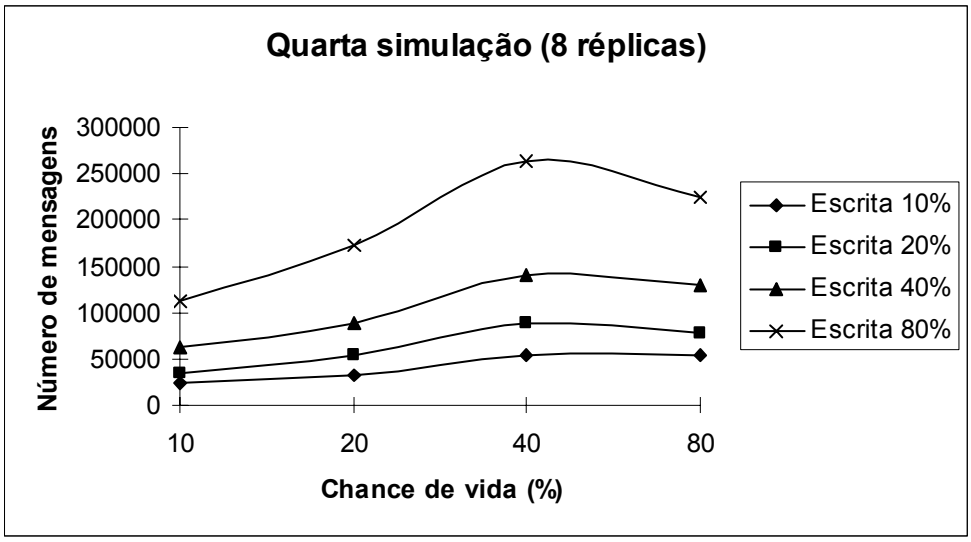

Figura 20. Número de mensagens, por chance de vida e intensidade de escrita, para a quarta simulação com 8 réplicas.

Mas há uma diminuição significativa no tamanho médio das mensagens como se pode perceber na Figura 21, pois as mensagens trocadas durante uma reconciliação são menores por conter somente dados parciais. Essa melhoria é acentuada com o aumento da chance de vida dos nós.

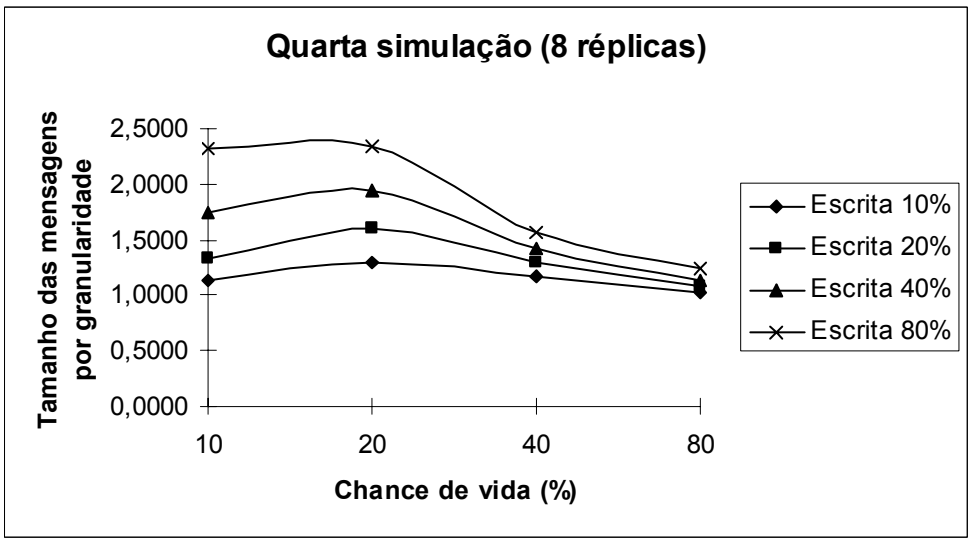

Figura 21. Tamanho médio das mensagens por chance de vida e intensidade de escrita, para a quarta simulação com 8 réplicas.

A quantidade e a qualidade das réplicas ficam semelhantes as da terceira simulação, mesmo com o tamanho das mensagens sendo bem menor. Isso pode ser visto nas Figura 22 e Figura 23.

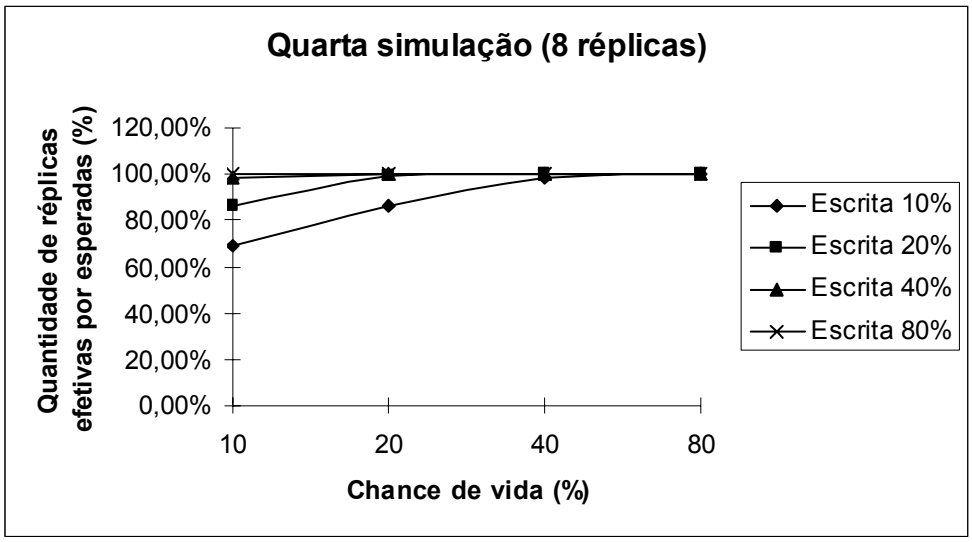

Figura 22. Quantidade de réplicas efetivas por esperadas, por chance de vida e intensidade de escrita, para a quarta simulação com 8 réplicas.

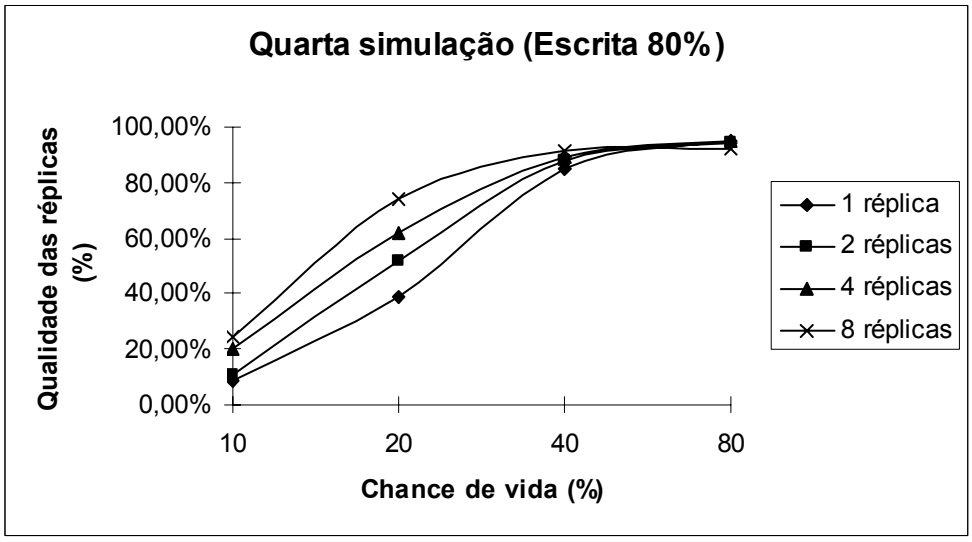

Figura 23. Qualidade das réplicas efetivas por chance de vida e número de réplicas, para a quarta simulação com 8 réplicas.

6.5 Comparações das simulações

Tendo apresentado os resultados individuais de cada simulação, é interessante apresentar gráficos comparativos dos quatro algoritmos. Como o escopo do trabalho se limita a avaliar os algoritmos na arquitetura peer-to-peer, a base utilizada para os gráficos será a chance de vida, que simula a entrada e saída dos nós na rede, característica presente nesta arquitetura.

Baseado nos resultados individuais apresentados acima, conclui-se que o número de réplicas deve ser pequeno, pois esse número está diretamente relacionado com o número de mensagens trocadas entre os nós, mas não apresenta variações significativas na quantidade efetiva e na qualidade das réplicas. Será utilizado para as comparações o número de duas réplicas.

Apesar do número de réplicas ser uma decisão do desenvolvedor da aplicação, ou do mantenedor da mesma, a intensidade de escrita depende da natureza do sistema. Portanto, serão analisadas duas intensidades de escrita, uma indicando sistemas com poucas alterações em seus metadados, com valor de 20%, e outra para sistemas com uma alta taxa de mudanças em metadados, com valor de 80%.

6.5.1 Comparações para intensidade de escrita de 20%

Na Figura 24 se pode perceber que aumentando a complexidade do algoritmo aumentamos também consideravelmente o número de mensagens trocadas entre os nós, devido às mensagens trocadas durante as reconciliações. Mas deve-se ressaltar que o primeiro algoritmo não suporta mensagens não comutáveis e oferece garantia de consistência piores que os outros algoritmos.

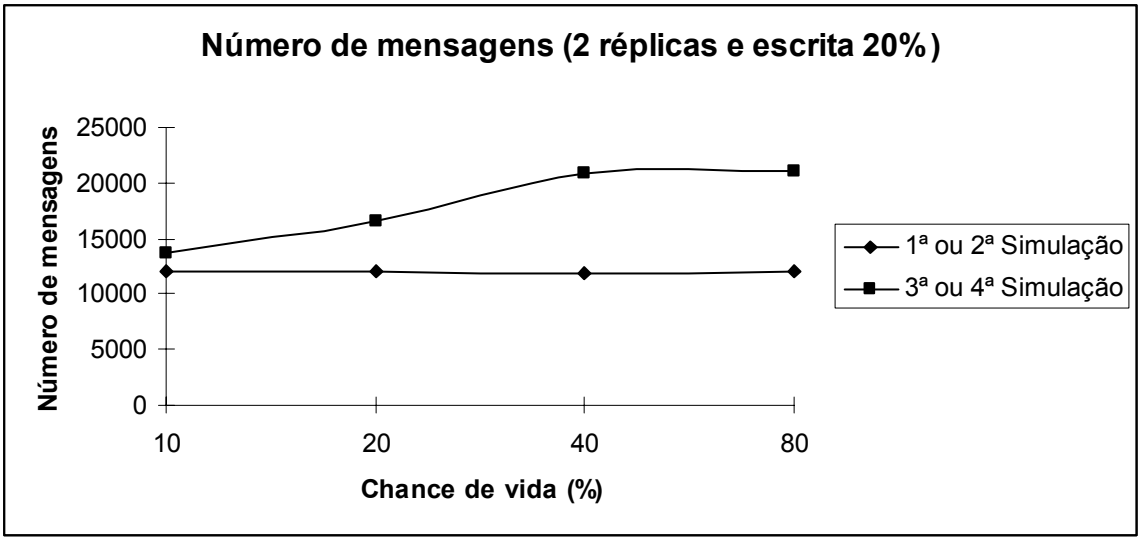

Figura 24. Número de mensagens por chance de vida, para as 4 simulações.

Analisando-se o tamanho médio das mensagens percebe-se que os algoritmos mais complexos acabam por compensar o aumento no número de mensagens (Figura 25), pois só utilizam mensagens maiores durante as reconciliações. O terceiro e o quarto algoritmos dobram o número de mensagens no pior caso, mas o segundo algoritmo tem mensagens que são mais que quatro vezes maior que o tamanho dos dados do nó.

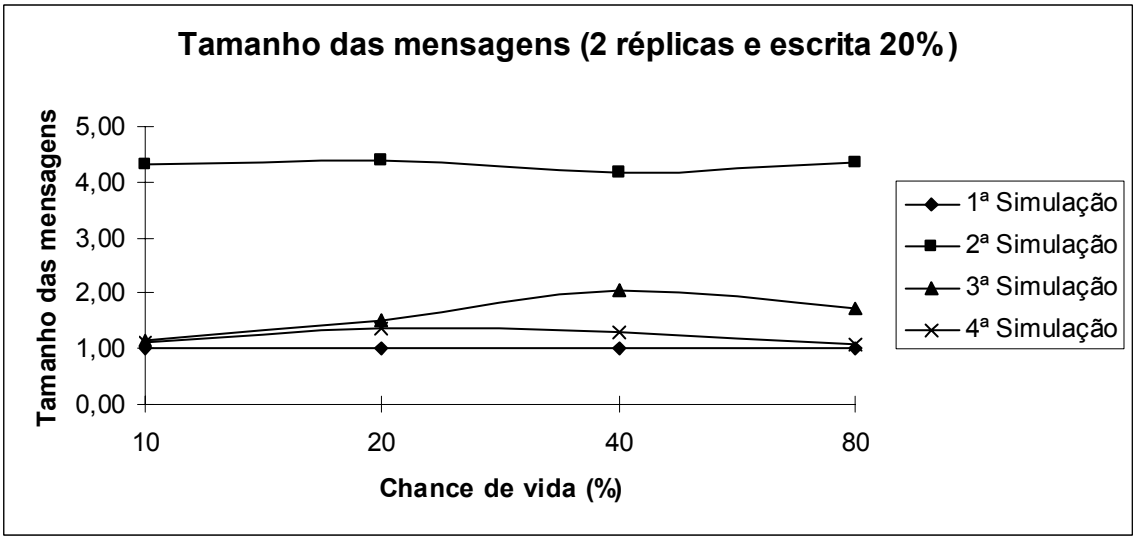

Figura 25. Tamanho médio das mensagens, por chance de vida, para as 4 simulações.

A razão entre réplicas efetivas e esperadas aumenta de acordo com a chance de um nó estar vivo, conforme esperado. Mas é interessante ressaltar que a partir de 20% de chance de vida essa razão já é quase 100%, como pode ser visto no gráfico da Figura 26.

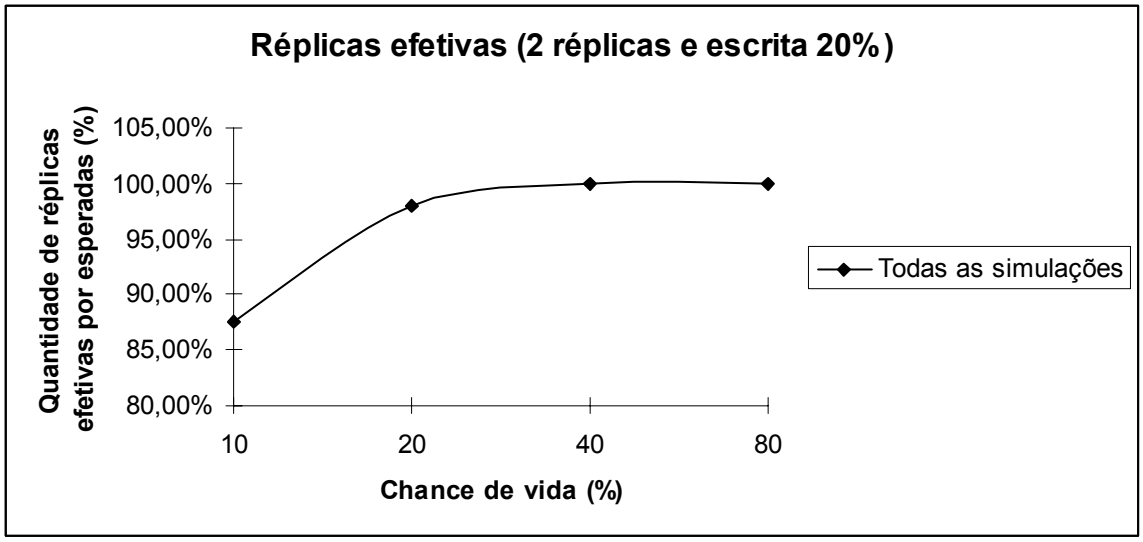

Figura 26. Quantidade de réplicas efetivas por esperadas, por chance de vida, para as 4 simulações.

A qualidade das réplicas efetivas da terceira e da quarta simulação é intermediária em relação às outras duas simulações. Os algoritmos com reconciliação se comportam melhor que o algoritmo de transferência de operações e pior que o algoritmo de transferência de estados para chances de vida inferiores a 40%. Para chances de vida superiores a 40% os algoritmos com reconciliação se comportam de maneira semelhante ao algoritmo de transferência por estados (Figura 27).

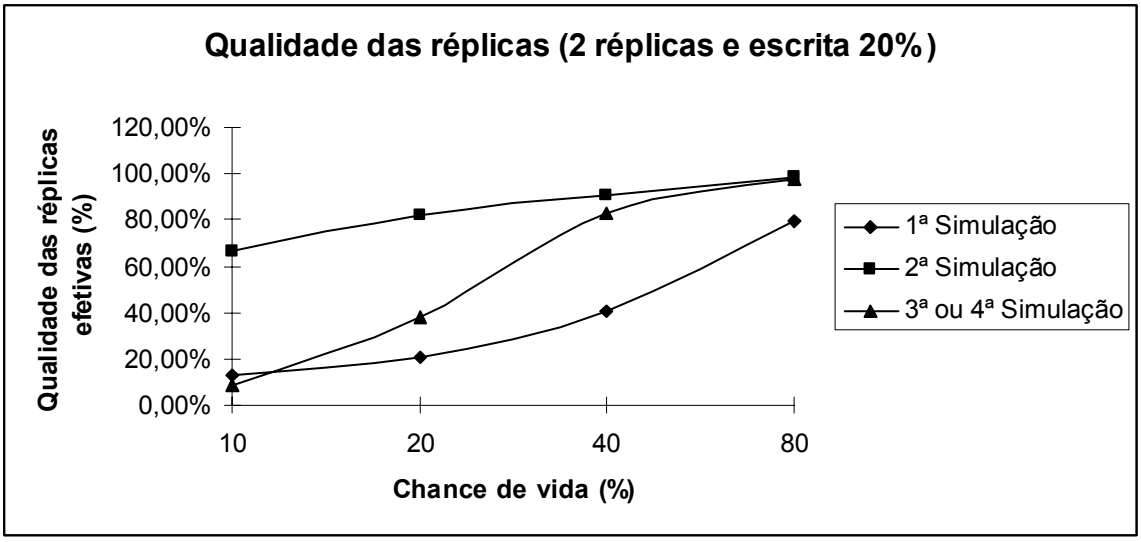

Figura 27. Qualidade das réplicas efetivas, por chance de vida, para as 4 simulações.

6.5.2 Comparações para intensidade de escrita de 80%

O aumento do número de mensagens entre os algoritmos com e sem reconciliação é ainda maior em sistemas com intensidade de escrita elevada, pois mais comandos precisam ser enviados para suas réplicas. Esse resultado pode ser visto na Figura 28.

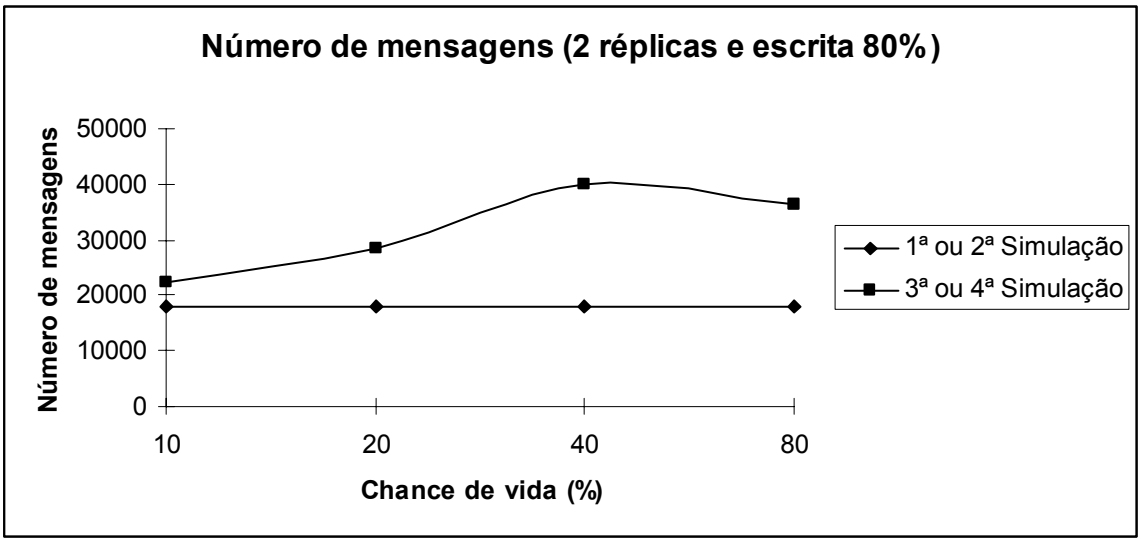

Figura 28. Número de mensagens por chance de vida, para as 4 simulações.

Mas a principal diferença quando comparamos as duas intensidades de escrita acima é no tamanho médio das mensagens trocadas entre os nós para a segunda simulação. Percebe-se no gráfico da Figura 29 que esse tamanho pode ser várias vezes maior que o tamanho médio das mensagens nas outras simulações.

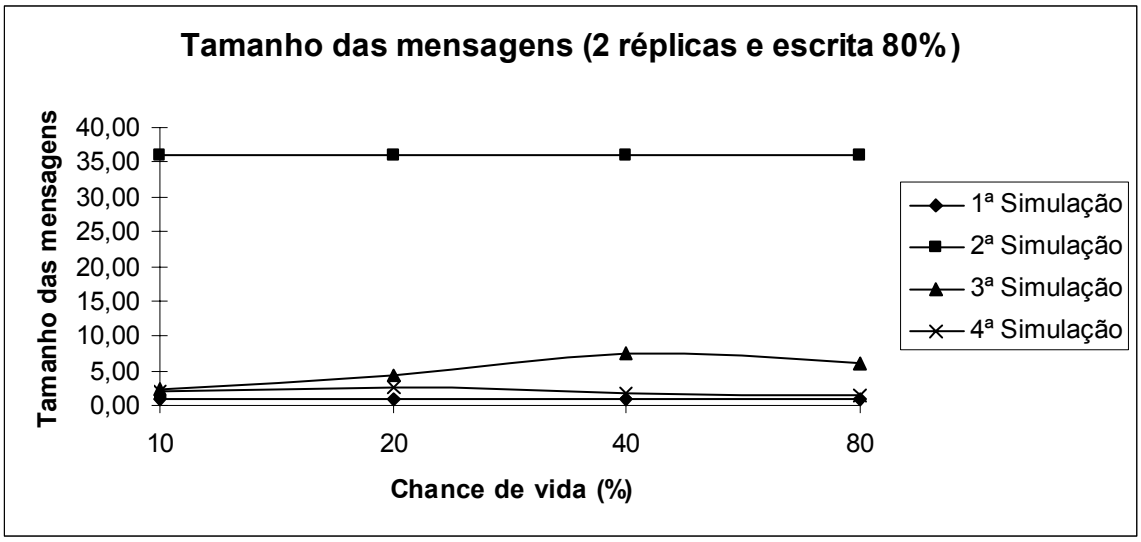

Figura 29. Tamanho médio das mensagens, por chance de vida, para as 4 simulações.

Outro resultado interessante é que para a intensidade de escrita de 80% e para 2 réplicas em todas as quatro simulações a quantidade de réplicas efetivas é igual ao número de réplicas esperadas, ou seja, independente da chance de vida dos nós a quantidade foi de 100%.

A qualidade das réplicas efetivas é semelhante à obtida para intensidade de escrita de 20%, como mostra o gráfico da Figura 30.

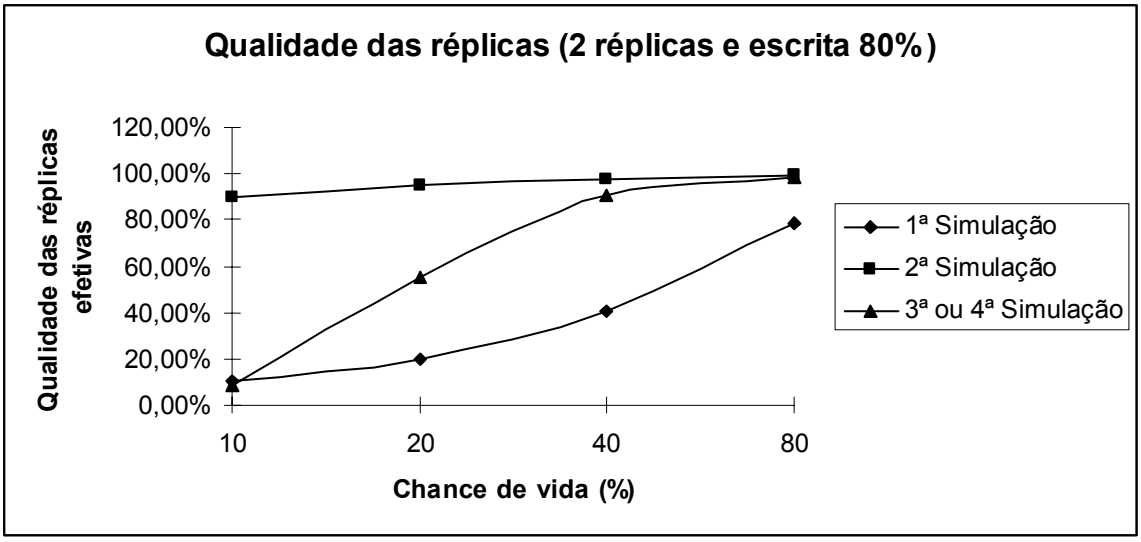

Figura 30. Qualidade das réplicas efetivas por chance de vida, para as 4 simulações.

6.6 Conclusão

Conclui-se que a replicação otimista se torna bastante interessante quando a chance de vida dos nós é superior a 40%, ou seja, que os nós participantes da aplicação tenham uma taxa de entrada e saída da rede que possibilite que pelo menos 40% das mensagens cheguem com sucesso ao seu destino. Apesar de essa condição ser facilmente obtida em aplicações departamentais, algumas aplicações distribuídas mundialmente não apresentam essa característica.

A partir dessa faixa de chance de vida, o número de mensagens é cada vez menor devido ao menor número de reconciliações necessárias. A quantidade de réplicas efetivas é alto mesmo em sistemas com intensidade de escrita baixa. A qualidade das réplicas é boa, partindo de aproximadamente 85% de convergência com o master e tendendo a 100% com o aumento da chance de vida.

O algoritmo de transferência de estados é um algoritmo bastante simples e que oferece bons resultados mesmo com chance de vida inferior a 40%. Porém, esse algoritmo apresenta

uma desvantagem aumentando o tamanho médio das mensagens. Isso pode ser um problema caso o tamanho dos dados em cada nó seja grande, ou crescer rapidamente com a execução de comandos de escrita. Esses problemas ainda podem ser acentuados caso o sistema tenha uma alta intensidade de escrita.

Os algoritmos que usam reconciliação, por estado ou por comandos, diminuem consideravelmente a média de tamanho das mensagens, em contrapartida, aumentam o número de mensagens. Porém, principalmente em sistemas com alta intensidade de escrita, o consumo total de banda é menor nessas configurações. Os dois algoritmos com reconciliação apresentam resultados bastante parecidos, exceto pela diferença a favor da reconciliação por comandos no tamanho médio das mensagens. Porém, esse ganho vem com um aumento da complexidade, pois nesse algoritmo é necessário que cada nó guarde um histórico dos últimos comandos executados.

Em qualquer das opções é interessante manter o número de réplicas pequeno. Para os casos estudados um número interessante estaria entre duas e quatro réplicas.

Capítulo 7 Conclusão

Replicação otimista de dados aumenta a disponibilidade das informações, melhorando o desempenho das aplicações. Aumenta-se também a escalabilidade do número de nós participantes ao relaxar a sincronização necessária entre as réplicas. Porém, para obter essas características é necessário aumentar a complexidade da aplicação.

Um bom equilíbrio entre velocidade de propagação de operações de escrita, número de réplicas e consumo de banda é importante. Quanto mais rápido as alterações forem propagadas para as réplicas, maior a convergência entre as réplicas e o master e menor a probabilidade de um conflito ocorrer.

É importante projetar o mecanismo de replicação para tirar proveito da propriedade da comutatividade de operações. Permitir que operações não conflitantes sejam executadas simultaneamente no sistema reduz o número de conflitos facilitando a sincronização das réplicas.

Uma prática interessante no desenvolvimento de sistemas é manter simplicidade. Essa regra deve ser usada também nas escolhas do projeto de mecanismo de replicação otimista. Para muitos sistemas não é necessário um projeto de replicação otimista muito complexo, facilitando o desenvolvimento e a manutenção da aplicação.

7.1 Contribuição

Levando em consideração as citações acima, esse trabalho apresenta um estudo teórico contendo uma análise das propriedades de algoritmos de replicação sobre redes peer-to-peer. Também foi apresentada uma implementação de referência para simulações de quatro

algoritmos de replicação otimista. Os algoritmos apresentados são single-master e variam de acordo com as seguintes características:

- Push com transferência de operações
- Push com transferência de estados
- Push com transferência de operações com reconciliação pull com transferência de estados
- Push com transferência de operações com reconciliação pull com transferência de operações

Cada algoritmo foi avaliado em simulações que variavam o número de réplicas de cada master, a chance de cada nó responder as mensagens recebidas e a quantidade de operações de escrita.

Os resultados obtidos mostram que mesmo algoritmos simples, como o push com transferência de estados, pode oferecer bons resultados. E que aumentando a complexidade do algoritmo é possível obter um mecanismo de replicação com uma relação consumo de banda versus convergência das réplicas em relação ao master bastante interessante, como nos algoritmos com reconciliação.

Os resultados ainda mostram que o número de réplicas não deve ser grande, mantendo-se em poucas unidades. Conforme esperado, a técnica de transferência push é interessante para sistemas com grande quantidade de operações de escrita, melhorando sensivelmente a convergência das réplicas com o master.

Sobre a arquitetura peer-to-peer o fator preocupante fica com a taxa normalmente elevada de entrada e saída de nós da rede. Isso pode fazer com que muitas mensagens enviadas no sistema não cheguem ao seu destino. Os resultados indicam que a replicação

otimista começa a apresentar bons resultados quando a chance de um nó receber uma mensagem é superior a aproximadamente 40%.

Com isso conclui-se que utilizando replicação otimista de forma consciente é possível aperfeiçoar sistemas peer-to-peer e até mesmo possibilitar funcionalidades que seriam inviáveis sobre um mecanismo de replicação pessimista.

7.2 Trabalhos Futuros

É interessante avaliar as respostas dos algoritmos estudados para aplicações de larga escala, que não estavam no escopo desse trabalho. Se aumentado o número de nós nas simulações é possível efetuar tal avaliação. Outra alteração interessante nos parâmetros das simulações seria utilizar intensidades de escrita pequenas, bastante comuns em aplicações reais, como por exemplo, valores menores que 10%.

Pode-se ainda evoluir a simulação de entrada e saída dos nós da rede, tornando o comportamento da simulação mais próximo da realidade de algumas aplicações atuais, incluindo a necessidade de lidar com a saída permanente de alguns nós.

Outros algoritmos de replicação de dados estruturados podem ser interessantes para outros tipos de informações de sistemas peer-to-peer. Replicação multi-master, com opções de detecção e resolução de conflitos [MARTINS, 2006], apesar de não fazerem parte do escopo desta pesquisa são um tema extenso de estudo.

Referências Bibliográficas

[ANDROUTSELLIS, 2004] Androutsellis-Theotokis, S., Spinellis, D. A survey of peer-to-peer content distribution technologies. ACM Comput. Surv. 36, 4 (Dezembro de 2004), 335-371, 2004.

[BHAGWAN, 2002] Bhagwan, R., Moore, D., Savage, S., Voelker, G. M. Replication strategies for highly available peer-to-peer storage. Proceedings of FuDiCo: Future directions in Distributed Computing, Junho de 2002.

[BHAGWAN, 2003] Bhagwan, R. Savage, S. Voelker, G. Understanding availability. In Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), Berkeley, CA, Fevereiro de 2003.

[BIRMAN, 1993] Birman, K. P. The process group approach to reliable distributed computing. Communications ACM 36, 12, 37-53, Dezembro de 1993.

[BOLOSKY, 2000] Bolosky, W. J., Douceur, J. R., Ely, D., and Theimer, M. 2000. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. Proceedings of ACM SIGMETRICS '00, 34-43, Santa Clara, California, United States, Junho de 2000.

[COHEN, 2002] Cohen, E. Shenker, S. Replication Strategies in Unstructured Peer-to-Peer Networks. SIGCOMM'02, 19-23 de Agosto de 2002.

[CUENCA, 2003] Cuenca-Acuna, Francisco Matias . Martin, Richard P., Nguyen, Thu D. Autonomous Replication for High Availability in Unstructured P2P Systems, srds, p. 99, 22nd International Symposium on Reliable Distributed Systems (SRDS'03), 2003.

[DÉFAGO, 2004] Défago, X. Schiper, A. Semi-passive replication and Lazy Consensus. Journal of Parallel and Distributed Computing 64, 1380-1398, 2004.

[DIÓGENES, 2006] Diógenes, F. A. Mendonça, N. C. LogMiddle: Um Middleware P2P para Replicação de Dados em Redes Móveis Ad Hoc. In II Workshop sobre Redes Peer-to-Peer (WP2P'06), evento integrante do XXIV Simpósio Brasileiro de Redes de Computadores (SBRC'06), 2006, Curitiba - PR. Anais do II Workshop sobre Redes Peer-to-Peer (WP2P'06), 2006.

[GAMMA, 1995] Gamma, E. Helm, R. Johnson, R. Vlissides, J. Design Patterns: Elements of reusable object-oriented software. Addison-Wesley Longman. 1995.

[GRANVILLE, 2005] Granville, L. Z. da Rosa, D. M. Panisson, A. Melchiors, C. Almeida, M. J. B. Tarouco, L. M. R. Managing computer networks using peer-to-peer technologies. Communications Magazine, IEEE, 43, 10, 62-68, Outubro de 2005.

[GUY, 1998] Guy, R. Reicher, P. Ratner, D. Gunter, M. Ma, W. Popek, G. Rumor: Mobile Data Access Through Optimistic Peer-to-peer Replication. Proceedings: ER'98 Workshop on Mobile Data Access, 1998.

[HAMADI, 2005] Hamadi, Y., Shapiro, M. Pushing Log-based Reconciliation. Int. J. on Artif. Intelligence Tools (IJAIT) 14, 3-4, 445-458, Junho de 2005.

[JOSEPH, 1986] Joseph, T. A. Birman, K. P. Low cost management of replicated data in fault-tolerant distributed systems. ACM Trans. Comput. Syst. 4, 1, 54-70, Fevereiro de 1986.

[JXTA, 2007] JXTA Project, http://www.jxta.org, Julho de 2007.

[KUBIATOWICZ, 2000] Kubiatowicz, J. OceanStore: An Architecture for Global-Scale Persistent Storage. In Proceedings of ASPLOS ’00, 190-201, Boston, MA, Novembro de 2000.

[KUBIATOWICZ, 2003] Kubiatowicz, J. Extracting guarantees from chaos. Commun. ACM 46, 2, 33-38, Fevereiro de 2003.

[LIN, 2004] Lin, W. K., Chiu, D. M., and Lee, Y. B. Erasure Code Replication Revisited. Proceedings of the Fourth international Conference on Peer-To-Peer Computing (P2p'04) - Volume 00 (August 25 - 27, 2004). P2P. IEEE Computer Society, Washington, DC, 90-97, 2004.

[LITTLE, 1993] Little, M. C. McCue, D. L. Shrivastava, S. K. Maintaining Information about Persistent Replicated Objects in a Distributed System. Proceedings of the Thirteenth International Conference on Distributed Computing Systems, Maio de 1993.

[LUNG, 2004] Lung, L. C. Bessani, A. N. Fraga, J. S. Programação de Sistemas Distribuídos Confiáveis. In: Claudio Cesar de Sá. (Org.). ERI-SC'04 - XII Escola Regional de Informática. 1 ed. Porto Alegre: SBC - Sociedade Brasileira de Computação - Regional Santa Catarina, 2004, v. 1, p. 1-40.

[MARTINS, 2006] Martins, V. Akbarinia, R. Pacitti, E. Valduriez, P. Reconciliation in the APPA P2P system. IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2006, Minneapolis. Proceedings of ICPADS 2006. Los Alamitos: IEEE Computer Society, 2006. v. 1. p. 401-410.

[ON, 2003] On, G., Schmitt, J., and Steinmetz, R. The Effectiveness of Realistic Replication Strategies on Quality of Availability for Peer-to-Peer Systems. Proceedings of the 3rd international Conference on Peer-To-Peer Computing (01-03 de Setembro de 2003). P2P. IEEE Computer Society, Washington, DC, 57, 2003.

[PANISSON, 2006] Panisson, A. Melchiors, C, Granville, L. Z. Almeida, M. J. B. Tarouco, L. M. R. Designing the Architecture of P2P-Based Network Management Systems. In Proceedings of 11th IEEE Symposium on Computers and Communications, 69-75, Washington, DC, 2006.

[PASIN, 2003] Pasin, M. Weber, T. S. del Fabro, M. D. Riveill, M. A Highly-Available Replicated Component-Based Distributed Architecture. SympAAA'2003. pages 549-560. France, Oct. 2003.

[RANGANATHAN, 2002] Ranganathan, K., Iamnitchi, A., and Foster, I. Improving Data Availability through Dynamic Model-Driven Replication in Large Peer-to-Peer Communities. Proceedings of the 2nd IEEE/ACM international Symposium on Cluster Computing and the Grid (21–24 de Maio de 2002). CCGRID. IEEE Computer Society, Washington, DC, 376, 2002.

[RODRIGUES, 2002] Rodrigues, R., Liskov, B., Shrira, L. The design of a robust peer-to-peer system. SIGOPS European Workshop, 2002.

[SAITO, 2005] Saito, Y., Shapiro, M. Optimistic replication. ACM Comput. Surv. 37, 1 (Março de 2005), 42-81, 2005.

[SCHULER, 2003] Schuler, C., Weber, R., Schuldt, H., Schek, H.J.: Peer-to-Peer Process Execution with Osiris. Proceedings of First International Conference on Service-Oriented Computing ICSOC, 2003.

[TSOUMAKOS, 2003] Tsoumakos, Dimitrios. Roussopoulos, Nick. A Comparison of Peer-to-Peer Search Methods. International Workshop on the Web and Databases (WebDB), 12-13 de Junho de 2003.

Apêndice A Código-fonte do projeto de simulações

A.1 Classe AbstractReplicationApplication

```
package simulation;

import java.util.*;

import rice.p2p.commonapi.*;

public abstract class AbstractReplicationApplication implements Application {

	protected final Endpoint endpoint;
	protected final Id id;
	protected final Set<Id> replicas;
	protected final Map<Id, Set<String>> data;
	protected final MessageCounter messageCounter;
	private final Node node;
	private final float upChance;
	private boolean up;

	public AbstractReplicationApplication(Node node, float upChance, MessageCounter
messageCounter) {
		this.node = node;
		this.id = this.node.getId();
		this.endpoint = this.node.buildEndpoint(this, "Replication");
		this.replicas = new HashSet<Id>();
		this.data = new HashMap<Id, Set<String>>();
		this.initializeNodeData(this.node.getId());
		this.upChance = upChance;
		this.up = true;
		this.messageCounter = messageCounter;

		this.endpoint.register();
	}

	public Node getNode() {
		return this.node;
	}

	public Set<Id> getReplicas() {
		return this.replicas;
	}

	public Map<Id, Set<String>> getData() {
		return this.data;
	}

	public void sendTextMessage(Id to, TextMessage message) {
		this.endpoint.route(to, message, null);
		this.messageCounter.incrementTextMessagesSent();
		System.out.println("Mensagem enviada de "+this.id+" para "+to+": "+message);
	}

	public void addReplica(Id replicaId) {
		this.replicas.add(replicaId);
	}

	public void executeCommand(Command command) {
		command.executeOn(this.data);
		this.propagateCommand(command);
	}

	public boolean forward(RouteMessage arg0) {
```

```
		return true;
	}

	public void deliver(Id id, Message message) {
		Random random = new Random();
		float randomFloat = random.nextFloat();
		if (randomFloat <= upChance) {
			if (this.up) {
				this.deliverMessage(id, message);
			}
			else {
				this.up = true;
				this.recoverWithMessage(id, message);
			}
		} else {
			this.up = false;
			this.messageCounter.incrementRejectedReceived();
			System.out.println("Mensagem rejeitada: " + message);
		}
	}

	public void update(NodeHandle arg0, boolean arg1) {
	}

	@Override
	public String toString() {
		StringBuffer buffer = new StringBuffer(this.id.toString());
		buffer.append("\n");
		for (Id replicationId : data.keySet()) {
			buffer.append("\t");
			buffer.append(replicationId.toString());
			buffer.append("\n");
			for (String dataItem : data.get(replicationId)) {
				buffer.append("\t\t");
				buffer.append(dataItem);
				buffer.append("\n");
			}
		}
		return buffer.toString();
	}

	protected abstract void propagateCommand(Command operationCommand);

	protected abstract void deliverMessage(Id id, Message message);

	protected abstract void recoverWithMessage(Id id, Message message);

	protected void initializeNodeData(Id id) {
		Set<String> nodeData = new HashSet<String>();
		this.data.put(id, nodeData);
	}
}
```

A.2 Classe AddOperationCommand

```
package simulation;

import java.util.Map;
import java.util.Set;

import rice.p2p.commonapi.Id;
import rice.p2p.commonapi.Message;

public class AddOperationCommand implements Command {

	private Id invoker;
	private String newData;

	public AddOperationCommand(Id invoker, String newData) {
		this.invoker = invoker;
		this.newData = newData;
```

```
	}

	public Id getInvoker() {
		return this.invoker;
	}

	public void executeOn(Object _dataCollection) {
		Map<Id, Set<String>> dataCollection = (Map<Id, Set<String>>)_dataCollection;
		Set<String> nodeData = dataCollection.get(this.invoker);
		nodeData.add(this.newData);
	}

	public int getPriority() {
		return Message.LOW_PRIORITY;
	}

	@Override
	public String toString() {
		return "Adição do dado " + newData + " de " + this.invoker;
	}

}
```

A.3 Interface Command

```
package simulation;

import rice.p2p.commonapi.Id;
import rice.p2p.commonapi.Message;

public interface Command extends Message {

	public Id getInvoker();
	public void executeOn(Object _dataCollection);
}
```

A.4 Classe MessageCounter

```
package simulation;

import java.io.FileWriter;
import java.io.IOException;

public class MessageCounter {

	private long textMessagesSent;
	private long commandsSent;
	private long queriesSent;
	private long responsesSent;
	private long messagesSize;
	private long textMessagesReceived;
	private long commandsReceived;
	private long queriesReceived;
	private long responsesReceived;
	private long rejectedReceived;

	private FileWriter writer;
	private int linha;

	public void incrementTextMessagesSent() {
		this.textMessagesSent++;
		this.messagesSize++;
	}

	public void incrementCommandsSent(int commandSize) {
		this.commandsSent++;
		this.messagesSize += commandSize;
	}
```

```
public void incrementQueriesSent() {
        this.queriesSent++;
        this.messagesSize++;
}

public void incrementResponsesSent(int responseSize) {
        this.responsesSent++;
        this.messagesSize += responseSize;
}

public void incrementTextMessagesReceived() {
        this.textMessagesReceived++;
}

public void incrementCommandsReceived() {
        this.commandsReceived++;
}

public void incrementQueriesReceived() {
        this.queriesReceived++;
}

public void incrementResponsesReceived() {
        this.responsesReceived++;
}

public void incrementRejectedReceived() {
        this.rejectedReceived++;
}

public void start(String fileName) throws IOException {
        this.writer = new FileWriter(fileName);
        this.linha = 1;
        this.writer.write("Descrição\t");
        this.writer.write("Mens Env\t");
        this.writer.write("Com Env\t");
        this.writer.write("Quer Env\t");
        this.writer.write("Res Env\t");
        this.writer.write("Tam Mens\t");
        this.writer.write("Mens Rec\t");
        this.writer.write("Com Rec\t");
        this.writer.write("Quer Rec\t");
        this.writer.write("Res Rec\t");
        this.writer.write("Rej Rec\t");
        this.writer.write("Soma Env\t");
        this.writer.write("Soma Rec\t");
        this.writer.write("Media Tam\n");
        this.writer.flush();
}

public void split(String descritpion) throws IOException {
        this.linha++;

        this.writer.write(descritpion);
        this.writer.write("\t");
        this.writer.write(String.valueOf(this.textMessagesSent));
        this.writer.write("\t");
        this.writer.write(String.valueOf(this.commandsSent));
        this.writer.write("\t");
        this.writer.write(String.valueOf(this.queriesSent));
        this.writer.write("\t");
        this.writer.write(String.valueOf(this.responsesSent));
        this.writer.write("\t");
        this.writer.write(String.valueOf(this.messagesSize));
        this.writer.write("\t");
        this.writer.write(String.valueOf(this.textMessagesReceived));
        this.writer.write("\t");
        this.writer.write(String.valueOf(this.commandsReceived));
        this.writer.write("\t");
        this.writer.write(String.valueOf(this.queriesReceived));
        this.writer.write("\t");
        this.writer.write(String.valueOf(this.responsesReceived));
        this.writer.write("\t");
        this.writer.write(String.valueOf(this.rejectedReceived));
```

```
                this.writer.write("\t=Soma(B" + this.linha + ";C" + this.linha + ";D" +
this.linha + ";E" + this.linha + ")");
                this.writer.write("\t=Soma(G" + this.linha + ";H" + this.linha + ";I" +
this.linha + ";J" + this.linha + ";K" + this.linha + ")");
                this.writer.write("\t=F" + this.linha + "/L" + this.linha);
                this.writer.write("\n");
                this.writer.flush();

                this.textMessagesSent = 0;
                this.commandsSent = 0;
                this.queriesSent = 0;
                this.responsesSent = 0;
                this.messagesSize = 0;
                this.textMessagesReceived = 0;
                this.commandsReceived = 0;
                this.queriesReceived = 0;
                this.responsesReceived = 0;
                this.rejectedReceived = 0;
        }

        public void stop() throws IOException {
                this.writer.close();
        }
}
```

A.5 Classe OperationsQuery

```
package simulation;

import java.util.*;

import rice.p2p.commonapi.Id;
import rice.p2p.commonapi.Message;

public class OperationsQuery implements Query {

        private Id invoker;
        private Id receiver;
        private int state;
        private Object result;

        public OperationsQuery(Id invoker, Id receiver, int state) {
                this.invoker = invoker;
                this.receiver = receiver;
                this.state = state;
        }

        public Id getReceiver() {
                return this.receiver;
        }

        public Object getResult() {
                return this.result;
        }

        public Id getInvoker() {
                return this.invoker;
        }

        public void executeOn(Object _commandsList) {
                List<Command> commandsList = (List<Command>)_commandsList;
                if (commandsList.size() > this.state) {
                        this.result = new ArrayList<Command>();
                        for (int i = this.state; i < commandsList.size(); i++) {
                                ((List<Command>)this.result).add(commandsList.get(i));
                        }
                }
        }

        public int getPriority() {
                return Message.LOW_PRIORITY;
```

```
	}

	@Override
	public String toString() {
		return "Consulta dos dados de " + this.receiver + " para " + this.invoker;
	}
}
```

A.6 Classe PushOperationApplication

```
package simulation;

import rice.p2p.commonapi.Id;
import rice.p2p.commonapi.Message;
import rice.p2p.commonapi.Node;

public class PushOperationApplication extends AbstractReplicationApplication {

	public PushOperationApplication(Node node,float upChance,MessageCounter messageCounter)
{
		super(node, upChance, messageCounter);
	}

	@Override
	public void propagateCommand(Command operationCommand) {
		for (Id replicaId : replicas) {
			this.endpoint.route(replicaId, operationCommand, null);
			this.messageCounter.incrementCommandsSent(1);
			System.out.println("Comando enviado de "+this.id+" para "+replicaId+":
"+operationCommand);
		}
	}

	@Override
	public void deliverMessage(Id id, Message message) {
		if (message instanceof Command) {
			this.deliverCommand(id, (Command)message);
		} else {
			this.messageCounter.incrementTextMessagesReceived();
			System.out.println("Mensagem recebida de " + id + ": " + message);
		}
	}

	@Override
	protected void recoverWithMessage(Id id, Message message) {
		// Nao faz reconciliacao
		this.deliverMessage(id, message);
	}

	private void deliverCommand(Id id, Command command) {
		if (!this.data.containsKey(command.getInvoker())) {
			this.initializeNodeData(command.getInvoker());
		}
		command.executeOn(this.data);
		this.messageCounter.incrementCommandsReceived();
		System.out.println("Comando recebido de " + id + ": " + command);
	}
}
```

A.7 Classe PushOperationPullOperationsApplication

```
package simulation;

import java.util.ArrayList;
import java.util.List;

import rice.p2p.commonapi.Id;
import rice.p2p.commonapi.Message;
import rice.p2p.commonapi.Node;
```

```
public class PushOperationPullOperationsApplication extends
		AbstractReplicationApplication {

	private List<Command> commandsList;

	public PushOperationPullOperationsApplication(Node node, float upChance, MessageCounter
messageCounter) {
		super(node, upChance, messageCounter);
		this.commandsList = new ArrayList<Command>();
	}

	@Override
	protected void propagateCommand(Command operationCommand) {
		this.commandsList.add(operationCommand);
		for (Id replicaId : replicas) {
			this.endpoint.route(replicaId, operationCommand, null);
			this.messageCounter.incrementCommandsSent(1);
			System.out.println("Comando enviado de " + this.id + " para " +
replicaId + ": " + operationCommand);
		}
	}

	@Override
	protected void deliverMessage(Id id, Message message) {
		 if (message instanceof Query) {
			this.deliverQuery(id, (Query)message);
		} else if (message instanceof Command) {
			this.deliverCommand(id, (Command)message);
		} else {
			this.messageCounter.incrementTextMessagesReceived();
			System.out.println("Mensagem recebida de " + id + ": " + message);
		}
	}

	@Override
	protected void recoverWithMessage(Id id, Message message) {
		if (message instanceof Query) {
			this.deliverQuery(id, (Query)message);
		} else if (message instanceof Command) {
			Command command = (Command)message;
			if (!this.data.containsKey(command.getInvoker())) {
				this.initializeNodeData(command.getInvoker());
			}
			this.messageCounter.incrementRejectedReceived();
			System.out.println("Mensagem rejeitada: " + message);
			this.reconciliate();
		} else {
			this.messageCounter.incrementTextMessagesReceived();
			System.out.println("Mensagem recebida de " + id + ": " + message);
			this.reconciliate();
		}
	}

	private void deliverCommand(Id id, Command command) {
		if (!this.data.containsKey(command.getInvoker())) {
			this.initializeNodeData(command.getInvoker());
		}
		command.executeOn(this.data);
		this.messageCounter.incrementCommandsReceived();
		System.out.println("Comando recebido de " + id + ": " + command);
	}

	private void deliverQuery(Id id, Query query) {
		if (query.getReceiver().equals(this.id)) {
			query.executeOn(this.commandsList);

			this.messageCounter.incrementQueriesReceived();
			System.out.println("Consulta recebida de " + id + ": " + query);

			this.endpoint.route(query.getInvoker(), query, null);

			int tamanho = 1;
```

```
                        if (query.getResult() != null) tamanho =
((List)query.getResult()).size();
                        this.messageCounter.incrementResponsesSent(tamanho);
                        System.out.println("Resposta enviada de " + query.getReceiver() + " para
" + query.getInvoker());
                } else if (query.getInvoker().equals(this.id)) {
                        if (query.getResult() != null) {
                                List<Command> operationsList = (List<Command>)query.getResult();
                                for (Command command : operationsList) {
                                        command.executeOn(this.data);
                                }
                        }

                        this.messageCounter.incrementResponsesReceived();
                        System.out.println("Resposta recebida de " + id + ": " + query);
                }
        }

        private void reconciliate() {
                OperationsQuery operationsQuery = null;
                for (Id replicationId : this.data.keySet()) {
                        if (this.id.equals(replicationId)) continue;
                        int replicationState = this.data.get(replicationId).size();
                        operationsQuery = new OperationsQuery(this.id, replicationId,
replicationState);
                        this.endpoint.route(replicationId, operationsQuery, null);
                        this.messageCounter.incrementQueriesSent();
                        System.out.println("Consulta enviada de " + this.id + " para " +
replicationId);
                }
        }
}
```

A.8 Classe PushOperationPullStateApplication

```
package simulation;

import java.util.Set;

import rice.p2p.commonapi.Id;
import rice.p2p.commonapi.Message;
import rice.p2p.commonapi.Node;

public class PushOperationPullStateApplication extends
                AbstractReplicationApplication {

        public PushOperationPullStateApplication(Node node, float upChance, MessageCounter
messageCounter) {
                super(node, upChance, messageCounter);
        }

        @Override
        protected void propagateCommand(Command operationCommand) {
                for (Id replicaId : replicas) {
                        this.endpoint.route(replicaId, operationCommand, null);
                        this.messageCounter.incrementCommandsSent(1);
                        System.out.println("Comando enviado de " + this.id + " para " +
replicaId + ": " + operationCommand);
                }
        }

        @Override
        protected void deliverMessage(Id id, Message message) {
                 if (message instanceof Query) {
                        this.deliverQuery(id, (Query)message);
                } else if (message instanceof Command) {
                        this.deliverCommand(id, (Command)message);
                } else {
                        this.messageCounter.incrementTextMessagesReceived();
                        System.out.println("Mensagem recebida de " + id + ": " + message);
                }
```

```
	}

	@Override
	protected void recoverWithMessage(Id id, Message message) {
		if (message instanceof Query) {
			this.deliverQuery(id, (Query)message);
		} else if (message instanceof Command) {
			Command command = (Command)message;
			if (!this.data.containsKey(command.getInvoker())) {
				this.initializeNodeData(command.getInvoker());
			}
			this.messageCounter.incrementRejectedReceived();
			System.out.println("Mensagem rejeitada: " + message);
			this.reconciliate();
		} else {
			this.messageCounter.incrementTextMessagesReceived();
			System.out.println("Mensagem recebida de " + id + ": " + message);
			this.reconciliate();
		}
	}

	private void deliverCommand(Id id, Command command) {
		if (!this.data.containsKey(command.getInvoker())) {
			this.initializeNodeData(command.getInvoker());
		}
		command.executeOn(this.data);
		this.messageCounter.incrementCommandsReceived();
		System.out.println("Comando recebido de " + id + ": " + command);
	}

	private void deliverQuery(Id id, Query query) {
		if (query.getReceiver().equals(this.id)) {
			query.executeOn(this.data);

			this.messageCounter.incrementQueriesReceived();
			System.out.println("Consulta recebida de " + id + ": " + query);

			this.endpoint.route(query.getInvoker(), query, null);

			int tamanho = 1;
			if (query.getResult() != null) tamanho =
((Set)query.getResult()).size();
			this.messageCounter.incrementResponsesSent(tamanho);
			System.out.println("Resposta enviada de " + query.getReceiver() + " para
" + query.getInvoker());
		} else if (query.getInvoker().equals(this.id)) {
			if (query.getResult() != null) {
				this.data.put(query.getReceiver(),
(Set<String>)query.getResult());
			}

			this.messageCounter.incrementResponsesReceived();
			System.out.println("Resposta recebida de " + id + ": " + query);
		}
	}

	private void reconciliate() {
		StateQuery stateQuery = null;
		for (Id replicationId : this.data.keySet()) {
			if (this.id.equals(replicationId)) continue;
			int replicationState = this.data.get(replicationId).size();
			stateQuery = new StateQuery(this.id, replicationId, replicationState);
			this.endpoint.route(replicationId, stateQuery, null);
			this.messageCounter.incrementQueriesSent();
			System.out.println("Consulta enviada de " + this.id + " para " +
replicationId);
		}
	}
}
```

A.9 Classe PushStateApplication

```
package simulation;

import java.util.HashSet;
import java.util.Set;

import rice.p2p.commonapi.Id;
import rice.p2p.commonapi.Message;
import rice.p2p.commonapi.Node;

public class PushStateApplication extends AbstractReplicationApplication {

	public PushStateApplication(Node node, float upChance, MessageCounter messageCounter) {
		super(node, upChance, messageCounter);
	}

	@Override
	public void propagateCommand(Command operationCommand) {
		Set<String> nodeData = new HashSet<String>(this.data.get(this.id));
		UpdateStateCommand stateCommand = new UpdateStateCommand(this.id, nodeData);

		for (Id replicaId : replicas) {
			this.endpoint.route(replicaId, stateCommand, null);
			this.messageCounter.incrementCommandsSent(nodeData.size());
			System.out.println("Comando enviado de " + this.id + " para " +
replicaId + ": " + stateCommand);
		}
	}

	@Override
	public void deliverMessage(Id id, Message message) {
		if (message instanceof Command) {
			this.deliverCommand(id, (Command)message);
		} else {
			this.messageCounter.incrementTextMessagesReceived();
			System.out.println("Mensagem recebida de " + id + ": " + message);
		}
	}

	@Override
	protected void recoverWithMessage(Id id2, Message message) {
		// Nao faz reconciliacao
		this.deliverMessage(id, message);
	}

	private void deliverCommand(Id id, Command command) {
		command.executeOn(this.data);
		this.messageCounter.incrementCommandsReceived();
		System.out.println("Comando recebido de " + id + ": " + command);
	}

}
```

A.10 Interface Query

```
package simulation;

import rice.p2p.commonapi.Id;

public interface Query extends Command {

	public Id getReceiver();
	public Object getResult();

}
```

A.11 Classe ReplicaCounter

```
package simulation;

import java.io.FileWriter;
import java.io.IOException;

public class ReplicaCounter {

	private static final int MAX_NUMBER_OF_REPLICAS = 12;
	private static final char MASTER_COLUMN = 'A';
	private static final char FIRST_REPLICA_COLUMN = 'B';
	private static final char LAST_REPLICA_COLUMN = 'M';
	private static final char REPLICAS_COUNT_COLUMN = 'N';
	private static final char REPLICAS_PERCENT_COLUMN = 'O';
	private static final char REPLICATION_AVERAGE_COLUMN = 'P';
	private static final char REPLICATION_PERCENT_COLUMN = 'Q';

	private FileWriter writer;
	private int linha;

	public void start(String fileName) throws IOException {
		this.writer = new FileWriter(fileName);
		this.linha = 1;
		this.writer.write("Master\t");
		for (int i = 1; i <= MAX_NUMBER_OF_REPLICAS; i++) {
			this.writer.write("Replica " + i + "\t");
		}
		this.writer.write("Rep. Efet.\t");
		this.writer.write("% Rep. Efet.\t");
		this.writer.write("Media Rep.\t");
		this.writer.write("Qlde Rep.\n");
		this.writer.flush();
	}

	public void countReplica(int[] replicas, int numberOfReplicas) throws IOException {
		this.linha++;

		for (int i = 0; i <= numberOfReplicas; i++) {
			if (replicas[i] >= 0) {
				this.writer.write("" + replicas[i]);
			}
			this.writer.write("\t");
		}
		for (int i = numberOfReplicas; i < MAX_NUMBER_OF_REPLICAS; i++) {
			this.writer.write("\t");
		}
		this.writer.write("=CONT.VALORES(" + FIRST_REPLICA_COLUMN + this.linha + ":" +
LAST_REPLICA_COLUMN + this.linha + ")\t");
		this.writer.write("=" + REPLICAS_COUNT_COLUMN + this.linha + "/" +
numberOfReplicas + "\t");
		this.writer.write("=SE(" + REPLICAS_COUNT_COLUMN + this.linha + ">0;MÉDIA(B" +
this.linha + ":" + LAST_REPLICA_COLUMN + this.linha + ");0)\t");
		this.writer.write("=SE(E(" + REPLICAS_COUNT_COLUMN + this.linha + ">0;" +
MASTER_COLUMN + this.linha + ">0);" + REPLICATION_AVERAGE_COLUMN + this.linha + "/" +
MASTER_COLUMN + this.linha + ";\"\")\n");
	}

	public void split(int numberOfNodes, String description) throws IOException {
		this.writer.write(description);
		for (int i = 0; i <= (MAX_NUMBER_OF_REPLICAS + 1); i++) {
			this.writer.write("\t");
		}
		this.writer.write("=Média(" + REPLICAS_PERCENT_COLUMN + (this.linha -
numberOfNodes + 1) + ":" + REPLICAS_PERCENT_COLUMN + this.linha + ")\t\t");
		this.writer.write("=Média(" + REPLICATION_PERCENT_COLUMN + (this.linha -
numberOfNodes + 1) + ":" + REPLICATION_PERCENT_COLUMN + this.linha + ")\n\n");
		this.writer.flush();

		this.linha += 2;
	}
```

```
	public void stop() throws IOException {
		this.writer.close();
	}
}
```

A.12 Classe SimulationMain

```
package simulation;

import java.io.IOException;
import java.util.*;

import rice.environment.Environment;
import rice.p2p.commonapi.Id;
import rice.pastry.NodeHandle;
import rice.pastry.NodeIdFactory;
import rice.pastry.PastryNode;
import rice.pastry.PastryNodeFactory;
import rice.pastry.direct.DirectPastryNodeFactory;
import rice.pastry.direct.EuclideanNetwork;
import rice.pastry.direct.NetworkSimulator;
import rice.pastry.standard.RandomNodeIdFactory;

public class SimulationMain {

	private static final int NUMBER_OF_NODES = 100;
	private static final int NUMBER_OF_CYCLES = 100;
	private static final int[] NUMBER_OF_REPLICAS = {1, 2, 4, 8};
	private static final float[] UP_CHANCE = {0.1f, 0.2f, 0.4f, 0.8f};
	private static final float[] COMMAND_CHANCE = {0.1f, 0.2f, 0.4f, 0.8f};

	private static final int TEMPO_MAIOR = 1000;
	private static final int TEMPO_MENOR = 200;

	private static Environment environment;
	private static NodeIdFactory nodeIdFactory;
	private static NetworkSimulator simulator;
	private static PastryNodeFactory nodeFactory;
	private static List<AbstractReplicationApplication> applications;
	private static MessageCounter messageCounter;
	private static ReplicaCounter replicaCounter;

	public static void main(String[] args) throws Exception {
		messageCounter = new MessageCounter();
		messageCounter.start("POPOMensagens.txt");
		replicaCounter = new ReplicaCounter();
		replicaCounter.start("POPOReplicas.txt");

		for (int i = 0; i < COMMAND_CHANCE.length; i++) {
			for (int j = 0; j < NUMBER_OF_REPLICAS.length; j++) {
				for (int k = 0; k < UP_CHANCE.length; k++) {
					environment = Environment.directEnvironment();
					nodeIdFactory = new RandomNodeIdFactory(environment);
					simulator = new EuclideanNetwork(environment);
					simulator.setMaxSpeed(1.0f);
					nodeFactory = new DirectPastryNodeFactory(nodeIdFactory,
simulator, environment);
					applications = new
ArrayList<AbstractReplicationApplication>();
					createNodes(UP_CHANCE[k]);
					simulator.setFullSpeed();
					environment.getTimeSource().sleep(TEMPO_MAIOR);
					addReplicas(NUMBER_OF_REPLICAS[j]);
					environment.getTimeSource().sleep(TEMPO_MAIOR);
					simulate(COMMAND_CHANCE[i]);
					environment.getTimeSource().sleep(TEMPO_MAIOR);
					messageCounter.split(COMMAND_CHANCE[i] + " " +
NUMBER_OF_REPLICAS[j] + " " + UP_CHANCE[k]);
					print();
					countReplicas(NUMBER_OF_REPLICAS[j]);
```

```
                                        replicaCounter.split(NUMBER_OF_NODES, COMMAND_CHANCE[i] +
" " + NUMBER_OF_REPLICAS[j] + " " + UP_CHANCE[k]);
                                        applications.clear();
                                        environment.destroy();
                                }
                        }
                }

                messageCounter.stop();
                replicaCounter.stop();
        }

        private static void createNodes(float upChance) throws Exception {
                NodeHandle bootHandle = null;
                PastryNode node = null;
                for (int nodeNumber = 0; nodeNumber < NUMBER_OF_NODES; nodeNumber++) {
                        node = nodeFactory.newNode(bootHandle);
                        bootHandle = node.getLocalHandle();
                        synchronized(node) {
                         while(!node.isReady() && !node.joinFailed()) {
                                node.wait(TEMPO_MENOR);
                                if (node.joinFailed()) {
                                        throw new IOException("Não foi possível criar o nó: " +
node.joinFailedReason());
                                }
                         }
                        }
                        AbstractReplicationApplication application = new
PushOperationPullOperationsApplication(node, upChance, messageCounter);
                        applications.add(application);
                        System.out.println("Nó " + nodeNumber + " criado com sucesso");
                }
        }

        private static void addReplicas(int numberOfReplicas) throws Exception {
                Random random = new Random();
                int applicationsSize = applications.size();
                float randomFloat = 0.0f;
                int randomIndex = 0;
                for (AbstractReplicationApplication application : applications) {
                        for (int replicaNumber = 0; replicaNumber < numberOfReplicas;
replicaNumber++) {
                                randomFloat = random.nextFloat();
                                randomIndex = (int)(randomFloat * applicationsSize);
                                Id replicaId = applications.get(randomIndex).getNode().getId();
                                application.addReplica(replicaId);
                                environment.getTimeSource().sleep(TEMPO_MENOR);
                        }
                }
        }

        private static void simulate(float commandChance) throws Exception {
                Random random = new Random();
                int applicationsSize = applications.size();
                int randomIndex = 0;
                for (int i = 0; i < NUMBER_OF_CYCLES; i++) {
                        for (AbstractReplicationApplication application : applications) {
                                float randomFloat = random.nextFloat();
                                if (randomFloat <= commandChance) {
                                        randomFloat = random.nextFloat();
                                        Command command = new
AddOperationCommand(application.getNode().getId(), String.valueOf(randomFloat));
                                        application.executeCommand(command);
                                } else {
                                        randomFloat = random.nextFloat();
                                        randomIndex = (int)(randomFloat * applicationsSize);
                                        Id destinationApplicationId =
applications.get(randomIndex).getNode().getId();
                                        TextMessage textMessage = new TextMessage("Olá mundo!");
                                        application.sendTextMessage(destinationApplicationId,
textMessage);
                                }
                                environment.getTimeSource().sleep(TEMPO_MENOR);
                        }
```

```
                }
        }

        private static void print() {
                for (AbstractReplicationApplication application : applications) {
                        System.out.println(application);
                }
        }

        private static void countReplicas(int numberOfReplicas) throws IOException {
                Map<Id, Map<Id, Set<String>>> allData = new HashMap<Id, Map<Id,
Set<String>>>();
                for (AbstractReplicationApplication application : applications) {
                        allData.put(application.getNode().getId(), application.getData());
                }
                int[] replicasStatus = null;
                for (AbstractReplicationApplication application : applications) {
                        replicasStatus = new int[numberOfReplicas + 1];
                        Id masterId = application.getNode().getId();
                        replicasStatus[0] = allData.get(masterId).get(masterId).size();
                        Set<Id> replicaIds = application.getReplicas();
                        int index = 1;
                        int replicaStatus = -1;
                        for (Id replicaId : replicaIds) {
                                Set<String> replicaData = allData.get(replicaId).get(masterId);
                                replicaStatus = -1;
                                if (replicaData != null) {
                                        replicaStatus = replicaData.size();
                                }
                                replicasStatus[index] = replicaStatus;
                                index++;
                        }
                        replicaCounter.countReplica(replicasStatus, numberOfReplicas);
                }
        }
}
```

A.13 Classe StateQuery

```
package simulation;

import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import rice.p2p.commonapi.Id;
import rice.p2p.commonapi.Message;

public class StateQuery implements Query {

        private Id invoker;
        private Id receiver;
        private int state;
        private Object result;

        public StateQuery(Id invoker, Id receiver, int state) {
                this.invoker = invoker;
                this.receiver = receiver;
                this.state = state;
        }

        public Id getReceiver() {
                return this.receiver;
        }

        public Object getResult() {
                return this.result;
        }

        public Id getInvoker() {
                return this.invoker;
```

```
	}

	public void executeOn(Object _dataCollection) {
		Map<Id, Set<String>> dataCollection = (Map<Id, Set<String>>)_dataCollection;
		Set<String> receiverData = dataCollection.get(this.receiver);
		if (receiverData.size() > this.state) {
			this.result = new HashSet<String>(receiverData);
		}
	}

	public int getPriority() {
		return Message.LOW_PRIORITY;
	}

	@Override
	public String toString() {
		return "Consulta dos dados de " + this.receiver + " para " + this.invoker;
	}
}
```

A.14 Classe TextMessage

```
package simulation;

import rice.p2p.commonapi.Message;

public class TextMessage implements Message {

	private String message;

	public TextMessage(String message) {
		this.message = message;
	}

	public String getMessage() {
		return message;
	}

	public int getPriority() {
		return Message.LOW_PRIORITY;
	}

	@Override
	public String toString() {
		return message;
	}

}
```

A.15 Classe UpdateStateCommand

```
package simulation;

import java.util.Map;
import java.util.Set;

import rice.p2p.commonapi.Id;
import rice.p2p.commonapi.Message;

public class UpdateStateCommand implements Command {

	private Id invoker;
	private Set<String> nodeData;

	public UpdateStateCommand(Id master, Set<String> nodeData) {
		this.invoker = master;
		this.nodeData = nodeData;
	}
```

```
	public Id getInvoker() {
		return this.invoker;
	}

	public void executeOn(Object _dataCollection) {
		Map<Id, Set<String>> dataCollection = (Map<Id, Set<String>>)_dataCollection;
		dataCollection.put(this.invoker, this.nodeData);
	}

	public int getPriority() {
		return Message.LOW_PRIORITY;
	}

	@Override
	public String toString() {
		return "Substituição dos dados de " + this.invoker + " (" +
this.nodeData.size() + ")";
	}
}
```