Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação dos Protocolos FreeStore de Reconfiguração de Sistemas de Quóruns

Mateus Antunes Braga

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Eduardo Adilio Pelinson Alchieri

Brasília
2014

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Homero Luiz Piccolo

Banca examinadora composta por:

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador) — CIC/UnB
Prof. Dr. André Costa Drummond — CIC/UnB
Prof. Marcos Fagundes Caetano — CIC/UnB

CIP — Catalogação Internacional na Publicação

Braga, Mateus Antunes.

Implementação dos Protocolos FreeStore de Reconfiguração de Sistemas de Quóruns / Mateus Antunes Braga. Brasília : UnB, 2014.
135 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2014.

1. tolerância a falhas, 2. sistemas distribuídos, 3. reconfiguração,
4. freestore, 5. sistemas distribuídos dinâmicos, 6. sistemas de quórum,
7. segurança de funcionamento, 8. memória compartilhada distribuída

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Implementação dos Protocolos FreeStore de Reconfiguração de Sistemas de Quóruns

Mateus Antunes Braga

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador)
CIC/UnB

Prof. Dr. André Costa Drummond
CIC/UnB

Prof. Marcos Fagundes Caetano
CIC/UnB

Prof. Dr. Homero Luiz Piccolo
Coordenador do Bacharelado em Ciência da Computação

Brasília, 11 de junho de 2014

Dedicatória

Dedico à minha mãe, Maria José, ao meu pai, João Braga, aos meus irmãos, Marcelo e Murilo.

Agradecimentos

Agradeço à minha família pelo amor, apoio e presença em minha vida; Ao professor Eduardo, por me apresentar à área de sistemas distribuídos e pelo seu trabalho como orientador; à CJR Empresa Júnior, pelas experiências e oportunidades essenciais ao meu desenvolvimento profissional; a todos meus amigos, que deixaram tudo mais feliz.

Resumo

Sistemas de quóruns são úteis na implementação consistente e confiável de sistemas de armazenamento de dados em presença de falhas. Estes sistemas geralmente compreendem um conjunto estático de servidores que implementam um registrador acessado através de operações de leitura e escrita. Este trabalho de conclusão de curso propõe uma implementação para o FreeStore, um conjunto de protocolos tolerantes a falhas capazes de emular um registrador em sistemas dinâmicos, onde processos podem entrar e sair, durante sua execução, através de reconfigurações. Um conjunto detalhado de experimentos avalia o desempenho dos protocolos implementados e possibilita uma maior compreensão a respeito do processo de reconfiguração de memória compartilhada.

Palavras-chave: tolerância a falhas, sistemas distribuídos, reconfiguração, freestore, sistemas distribuídos dinâmicos, sistemas de quórum, segurança de funcionamento, memória compartilhada distribuída

Abstract

Quorum systems are useful tools for implementing consistent and available storage in the presence of failures. These systems usually comprise a static set of servers that provide a fault-tolerant read/write register accessed by a set of clients. This paper proposes an implementation for FreeStore, a set of fault-tolerant protocols that emulates a register in dynamic systems in which processes are able to join/leave the servers set by reconfigurations. A set of experiments analyses the performance of the implementation and brings some light to the shared memory reconfiguration procedure.

Keywords: fault tolerance, distributed systems, reconfiguration, freestore, dynamic distributed systems, Quorum Systems, Dependability, distributed shared memory

Sumário

Lista de Figuras

Capítulo 1

Introdução

O desenvolvimento na área da tecnologia da comunicação e da computação vem mudando a maneira que as aplicações são projetadas e executadas. Uma dessas mudanças é o número e a variedade cada vez maior de sistemas distribuídos existentes. Um sistema distribuído é o resultado da integração de sistemas computacionais autônomos combinados de forma a atingir um objetivo comum. A evolução na forma de como estes sistemas computacionais são integrados trouxe novos desafios para a Ciência da Computação, dentre eles os relacionados com os sistemas distribuídos dinâmicos (SDD), cujas soluções são grandes oportunidades para desenvolvimentos antes inviáveis, como por exemplo os que envolvem redes móveis *ad hoc* (MANETs) [21, 34], redes de sensores sem fio (RSSF) [50], redes entre pares (*Peer-to-Peer* - P2P) [46], e grades computacionais [23].

A característica principal de um sistema distribuído dinâmico, a qual o difere significativamente dos sistemas convencionais (sistemas distribuídos estáticos), está relacionada com a possibilidade de recursos componentes (especialmente processos) entrarem e saírem do sistema em qualquer momento. Assim, pode-se definir informalmente um sistema distribuído dinâmico como um sistema que executa continuamente, sendo que seus modelos computacionais devem considerar um número variável de processos (ou recursos) fazendo parte do sistema a cada instante ou intervalo de tempo. Desta forma, as topologias e composições de processos podem mudar arbitrariamente durante uma execução da aplicação [3]. Essas mudanças também são chamadas de reconfigurações do sistema.

Os sistemas distribuídos dinâmicos em geral precisam lidar com muitas incertezas e acabam sendo sistemas de grande complexidade. Essas características fazem parte dos desafios no desenvolvimento de aplicações distribuídas e levam nossa atenção aos aspectos de segurança de funcionamento (*dependability*). Um sistema com segurança de funcionamento possui um comportamento que evolui segundo suas especificações, mesmo diante da ocorrência de falhas em componentes do sistema. Uma das técnicas para garantir segurança de funcionamento é tolerância a falhas, que utiliza redundância e permite mascarar falhas em um ou mais dos componentes do sistema.

Geralmente, os participantes de um sistema distribuído interagem através de trocas de mensagens entre si em uma rede de computadores, mas isso não impede que existam outros modelos de comunicação entre processos em aplicações distribuídas. Uma alternativa muito utilizada é simular uma memória compartilhada distribuída onde os processos possam se coordenarem através de escritas e leituras nessa memória, trocando assim o paradigma de troca de mensagem para o de leitura/escrita na memória.

Assim, uma memória compartilhada apresenta uma abstração familiar à programação de aplicações não distribuídas e possibilita o acesso direto aos dados compartilhados. No entanto, essa abordagem adiciona um custo na realização de leituras e escritas, o que pode diminuir o desempenho do sistema.

Uma das formas de prover memória compartilhada em sistemas distribuídos é através de Sistemas de Quóruns. Sistemas de Quóruns [25] são ferramentas usadas para garantir consistência e disponibilidade de dados, que são armazenados de forma replicada em um conjunto de servidores. Além de seu uso como blocos básico de construção (*building blocks*) para protocolos de sincronização (ex.: consenso), o grande atrativo destes sistemas está relacionado com seu poder de escalabilidade e balanceamento de carga, uma vez que as operações não precisam ser executadas por todos os servidores que compõem o sistema, mas apenas por um quórum dos mesmos. A consistência de um sistema de quóruns é assegurada pela propriedade de intersecção dos subconjuntos de servidores (quóruns).

1.1 Motivação

A área de sistemas distribuídos dinâmicos apresenta vários desafios interessantes, como por exemplo as propriedades de auto-organização, de autoadaptação e de execução descentralizada de tarefas. Aplicações projetadas para atender essas propriedades devem ser capazes de detectar e tratar mudanças que ocorrem na composição de uma aplicação distribuída, de forma a permitir que a mesma possa se reconfigurar em tempo de execução com deterioração mínima da qualidade de serviço. Essas são características que provê uma flexibilidade desejável para os usuários e fornecedores da aplicação e que motiva o trabalho.

Apesar de existirem algumas propostas teóricas para memória compartilhada através de sistemas de quóruns em sistemas distribuídos dinâmicos [1, 38, 39], não são conhecidas implementações e análises das mesmas. Por isso, esse trabalho busca estudar, implementar e analisar os protocolos do *FreeStore* [2], que representa um conjunto de protocolos com tolerância a falhas para se realizar a reconfiguração de sistemas de quóruns.

Outra motivação vem das exigências da maioria das aplicações distribuídas relacionadas com a segurança de funcionamento. Exigências que ganham mais peso devido à crescente necessidade de se confiar nos sistemas computacionais a nossa volta. Essa realidade motiva os estudos desse trabalho em tolerância a falhas como técnica para atender essas exigências.

1.2 Objetivos

São dois os objetivos gerais da realização desse trabalho:

1. Implementar os protocolos de reconfiguração de sistemas de quóruns *FreeStore*.
2. Realizar comparações e análises sobre as implementações realizadas.

Dentro destes objetivos gerais, são cinco os objetivos específicos da realização desse trabalho:

1. Estudar a área de tolerância a falhas em sistemas distribuídos dinâmicos.

2. Projetar a implementação dos protocolos do *FreeStore*.
3. Implementar o projeto.
4. Preparar o ambiente de coleta de dados.
5. Coletar e analisar os dados.

1.3 Organização do Texto

A organização deste texto reflete as diversas etapas cumpridas para alcançar os objetivos específicos listados na seção anterior.

O Capítulo 2 apresenta o conceito de segurança de funcionamento, discutindo sobre seus atributos, e sobre falhas, erros e defeitos em um sistema computacional. Além disso, este capítulo apresenta técnicas para aumentar a segurança de funcionamento em sistemas computacionais em geral e depois com foco nos sistemas distribuídos.

O Capítulo 3 apresenta desafios na implementação de aplicações distribuídas, abordando as características mais marcantes dos sistemas distribuídas e suas consequências. Este capítulo também apresenta os principais problemas encontrados na busca por uma maior segurança de funcionamento em aplicações distribuídas.

O Capítulo 4 descreve os protocolos do *FreeStore* e a implementação realizada neste trabalho. Nesse capítulo são apresentados vários detalhes da implementação, como a linguagem de programação, a especificação, a estrutura e as soluções para os desafios encontrados no desenvolvimento do sistema. Por fim, é apresentado o resumo de outras duas implementações utilizadas para fins de comparação com o *FreeStore*.

O Capítulo 5 apresenta os experimentos realizados com a implementação do *FreeStore* e os resultados. Comparações com outras implementações similares também são feitas. Um experimento em especial mostra como o *FreeStore* se comporta mediante falhas e reconfigurações.

Por último, o Capítulo 6 concluí o trabalho, apresentando uma visão geral dos estudos realizados e abordando os possíveis trabalhos futuros.

Capítulo 2

Segurança de Funcionamento em Sistemas Distribuídos

Sistemas computacionais podem ser diferenciados em relação ao quanto podemos depender nos serviços do sistema. Por exemplo, se o serviço vai estar disponível quando precisamos, ou se podemos confiar na resposta recebida para tomar qualquer tipo de decisão. Chamamos a propriedade que diz respeito a essas características do sistema de segurança de funcionamento.

Na Seção 2.1 descrevemos vários conceitos relacionados à segurança de funcionamento, como seus atributos e o que são falhas, erros e defeitos. Apresentamos também técnicas para aumentar a segurança de funcionamento de um sistema em geral na Seção 2.2, e por último focamos nas técnicas de redundância mais comuns destinadas aos sistemas distribuídos na Seção 2.3.

2.1 Conceitos Básicos

Alguns conceitos são fundamentais para tratar sistemas de comunicação e computação. Para começar, um sistema é caracterizado por propriedades fundamentais: funcionalidade, performance, segurança de funcionamento (e segurança contra agentes maliciosos) e custos. A função de um sistema é descrita pela sua especificação funcional em termos de funcionalidade e performance, e o seu comportamento é a maneira que o sistema implementa sua função e pode ser descrita por uma sequência de estados. Para funcionar, um sistema possui uma certa estrutura e pode ser visto como um conjunto de componentes, onde cada componente é um outro sistema. O serviço fornecido por um sistema é o comportamento percebido pelos seus usuários [8].

A segurança de funcionamento é a propriedade que mais recebe atenção nesse trabalho devido a sua forte relação com tolerância a falhas. A segurança de funcionamento de um sistema pode ser definida da seguinte forma [8]:

- **Segurança de Funcionamento** (*Dependability*): a habilidade de fornecer serviço que pode ser justificadamente confiável;

Essa definição coloca ênfase na necessidade de justificativas para confiar no sistema. Uma definição alternativa que provê um critério para decidir se um serviço é confiável é a seguinte:

- **Segurança de Funcionamento** (*Dependability*): a habilidade de evitar defeitos no serviço que são mais frequentes e mais severos do que o aceitável;

Essa definição, por sua vez, coloca ênfase no ato de evitar defeitos. Definiremos o que são defeitos a seguir na Seção 2.1.1.

A segurança de funcionamento possui os seguintes atributos [8]:

- **Confiabilidade** (*Reliability*): o sistema mantém o correto fornecimento de seu serviço;
- **Disponibilidade** (*Availability*): o sistema sempre está pronto para fornecer seu serviço;
- **Segurança** (*Safety*): impossibilidade de ocorrerem consequências catastróficas com o usuário ou o ambiente;
- **Integridade** (*Integrity*): impossibilidade de ocorrerem alterações impróprias no sistema, i.e., o estado do sistema só pode ser alterado através da correta execução de suas operações;
- **Manutenibilidade** (*Maintainability*): facilidade de modificar e reparar o sistema.

Esses atributos estão presentes nos sistemas em níveis diferentes. A escolha dos níveis de cada atributo não é simples pois alguns dos atributos são conflitantes. O que normalmente se busca é uma combinação balanceada apropriada dos atributos para cada sistema. Essa escolha se encontra, implicitamente ou explicitamente, na especificação do sistema.

O estudo da segurança de funcionamento está fortemente ligado a outras características de um sistema, como a segurança do sistema a ataques maliciosos. Nesse trabalho, não nos preocupamos com falhas causadas por agentes maliciosos. Essa simplificação é útil para diminuir a carga sobre esse trabalho e não tira a sua praticidade. Simplificações desse tipo já foram utilizadas para sistemas reais que executam apenas dentro de uma rede privada que possui os devidos mecanismos de proteção contra agentes maliciosos. Um exemplo é o sistema Dynamos da empresa americana de comércio eletrônico Amazon [20], um sistema de armazenamento que suporta boa parte dos serviços da empresa.

Com o amadurecimento da indústria da Tecnologia da Informação, os sistemas computacionais estão fazendo cada vez mais parte das nossas vidas. Aos poucos, a automatização se insere em contextos cada vez mais críticos e que exigem uma maior confiança sobre os sistemas envolvidos. Esse aumento é refletido numa maior expectativa de sistemas computacionais com maior segurança de funcionamento e é desse aumento que vem a importância de se estudar essa propriedade. Por isso, um dos objetivos do projetista e desenvolvedor de sistemas distribuídos é desenvolver um sistema que evita defeitos de maneira racional.

2.1.1 Falhas, Erros e Defeitos

Para facilitar a discussão sobre tolerância a falhas, vamos definir o que são falhas, erros, defeitos e como eles se relacionam.

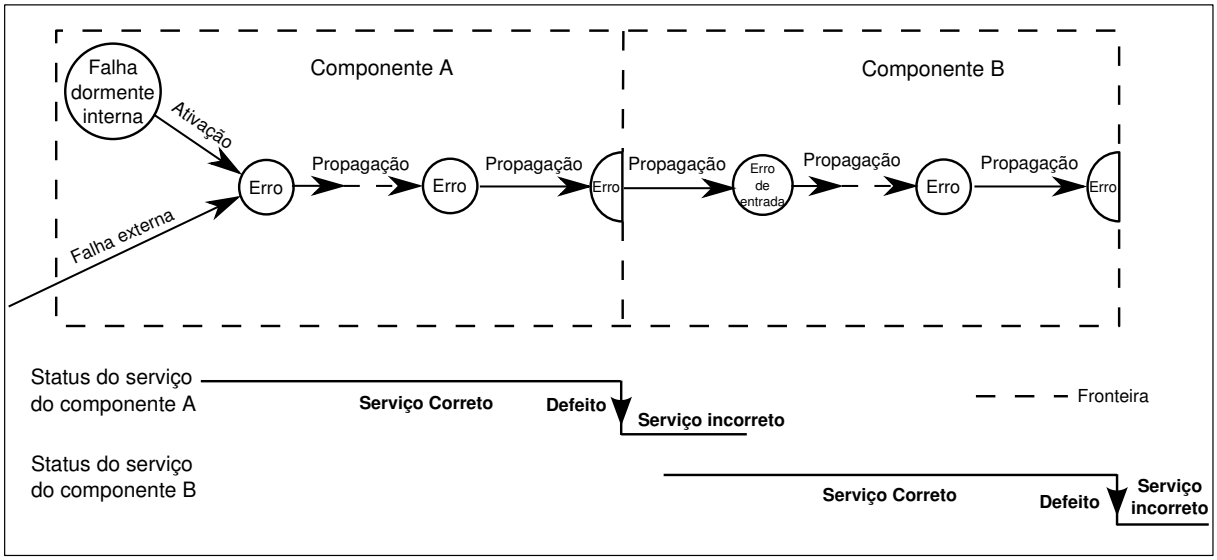


Figura 2.1: Interação entre falhas, erros e defeitos [8].

- **Defeito** (*Failure*): evento que ocorre quando o serviço fornecido desvia do serviço correto;
- **Erro** (*Error*): parte do estado do sistema que desvia do estado correto do serviço;
- **Falha** (*Fault*): causa real ou suposta de um erro.

Falhas são inevitáveis. As causas pelas quais elas surgem nos sistemas computacionais são diversas e, dentre elas, a maior é a grande complexidade desses sistemas. Não podemos evitá-las mas podemos tolerá-las e essa é uma das principais técnicas para aumentar a segurança de funcionamento de um sistema. Uma falha quando ativada faz com que uma parte do estado do sistema fique incorreta, ou seja, gera um erro.

Um erro é parte do estado do sistema que pode conduzir a um defeito. Não necessariamente um erro causa um defeito, é preciso que o erro influencie o serviço fornecido pelo sistema. As vezes a parte do sistema que contém o erro não chega a ser executada e o erro é ignorado. A propagação de um erro dentro de um sistema é a transformação sucessiva desse erro em outros erros.

Um sistema apresenta defeito quando não está fornecendo o serviço esperado mesmo seguindo a sua especificação. Um defeito de segurança de funcionamento ocorre quando o sistema sofre defeitos de serviço mais frequentes ou mais severos do que o aceitável.

A Figura 2.1 ilustra a interação entre falhas, erros e defeitos: uma falha externa ou uma falha dormente interna quando ativada, vira um erro que é propagado dentro do componente. Quando o erro é refletido em sua interface, ou seja, quando o serviço é afetado, dizemos que o erro gerou um defeito no serviço do componente. O serviço é dito correto até o momento em que o erro se torna um defeito.

Existem várias formas diferentes de se categorizar as falhas. Todas as falhas que podem afetar um sistema durante sua vida são classificadas de acordo com 8 pontos de vista básicos [8]:

- Fase da criação ou ocorrência (desenvolvimento/operacionais)

- Fronteira do sistema (interna/externa)
- Causa fenomenológica (natural/causada por humanos)
- Dimensão (hardware/software)
- Objetivo (malicioso/não malicioso)
- Intenção (intencionado/não intencionado)
- Capacidade (acidental/incompetência)
- Persistência (permanente/temporária)

Um outro ponto de vista que pode ser usado para categorizar falhas é a reprodutibilidade da falha. As falhas reproduzíveis são chamadas de falhas sólidas (*solid, or hard faults*), enquanto as falhas em que não são sistematicamente reproduzíveis são chamadas de falhas evasivas (*elusive, or soft faults*).

2.2 Meios para Alcançar a Segurança de Funcionamento

Nessa seção vamos falar sobre os quatro métodos para alcançar segurança de funcionamento. Eles são:

- Prevenção de falhas
- Tolerância a falhas
- Remoção de falhas
- Previsão de falhas

Todos esses métodos de uma forma ou de outra, tem como o objetivo evitar defeitos. As seções a seguir falam sobre cada um deles, mas a ênfase desse trabalho é na tolerância a falhas.

2.2.1 Prevenção de Falhas

Prevenção de falhas é um assunto geral relacionado com a engenharia de sistemas computacionais. Existem diversas formas para prevenir que falhas surjam, dependendo do tipo da falha. Para as falhas no desenvolvimento do software, a engenharia de software provê metodologias de desenvolvimento que trazem boas práticas e convenções para reduzir o número de falhas introduzidas pelos desenvolvedores. Algumas metodologias vão além e realizam a catalogação das falhas nos produtos para melhorar o processo de desenvolvimento e eliminar mais causas de falhas.

2.2.2 Tolerância a Falhas

Tolerância a falhas é realizada através da **detecção de erro** e **recuperação do sistema**.

- **Detecção de erro**: identifica a presença de um erro.
- **Recuperação do sistema**: transforma o estado do sistema que contém um ou mais erros e (possivelmente) falhas em um estado sem erros detectados e sem falhas que podem ser ativadas novamente.

A detecção de erro pode ser realizada de forma preemptiva ou concorrente. Na preemptiva, o sistema suspende o fornecimento do seu serviço para a realização da checagem por erros e falhas dormentes, enquanto que na concorrente a checagem é realizada ao mesmo tempo que o sistema fornece seu serviço.

A recuperação do sistema ocorre através do tratamento de erros ou tratamento de falhas. No tratamento dos erros, eliminamos os erros do estado do sistema, enquanto que no tratamento das falhas prevenimos que as falhas sejam ativadas novamente e assim não geram mais erros. Podemos realizar ambos ao mesmo tempo em um sistema.

O tratamento de falhas pode ser dividida em quatro atividades:

- **Diagnóstico** (*Diagnosis*): identifica e registra a causa(s) dos erro(s) em termos de localização e tipo.
- **Isolamento de falha** (*Isolation*): excluí componentes faltosos de futuras participações no fornecimento do serviço, i.e., torna a falha dormente.
- **Reconfiguração** (*Reconfiguration*): incluí componentes extras ou redistribuí tarefas entre componentes livres de falhas.
- **Reinicialização** (*Reinitialization*): checa, atualiza e registra a nova configuração e atualiza as tabelas e registros do sistema.

O diagnóstico é uma atividade de grande importância pois os dados gerados aqui podem ser utilizados na prevenção e previsão de falhas. Ou seja, gera insumos úteis para se alcançar a segurança de funcionamento em geral.

Um sistema distribuído dinâmico como o desenvolvido nesse trabalho apresenta vantagens quanto à reconfiguração do sistema. Com a possibilidade de adicionar e remover participantes do sistema durante sua execução, um sistema distribuído dinâmico facilita o tratamento de falhas no sistema.

O tratamento de erros, por sua vez, pode ser realizado de três formas:

- **Recuperação por retorno** (*rollback*): trás o sistema de volta a um estado salvo anterior à ocorrência dos erros (*checkpoint*).
- **Recuperação por avanço** (*rollforward*): coloca o sistema em um estado sem os erros detectado (reexecutar operação ou retransmitir mensagem).
- **Compensação** (*Compensation*): o estado incorreto possui **redundância** o suficiente para mascarar o erro.

A recuperação por retorno e a recuperação por avanço podem ser utilizadas de forma complementar. Por exemplo: primeiro se tenta fazer a recuperação por retorno, se o erro persistir, se tenta a recuperação por avanço.

O uso sistemático de compensação através de redundância também é chamado de mascaramento de falhas. A Seção 2.3 apresentará algumas das técnicas mais populares de redundância em sistemas distribuídos.

A eficiência da técnica de compensação é fortemente dependente do quão independente são as falhas nos componentes redundantes. O sucesso na implementação desse método só é possível quando os componentes redundantes não apresentem as mesmas falhas, pois somente dessa forma é possível mascarar os erros. Essa exigência torna o método mais prático para falhas que afetam o hardware do sistema, e é esse o uso mais comum.

Para a implementação de compensação em que se busca a tolerância de falhas de desenvolvimento, a exigência é de que se haja diversidade de design (*design diversity*). Ou seja, é preciso de componentes que implementam a mesma função mas que foram desenvolvidos por diferentes projetos e implementações. Somente dessa forma a redundância executa a função e apresenta tolerância a falhas inseridas no desenvolvimento de um componente em particular. Essa técnica também é chamada de programação N-Versões (*N-Version programming*).

A Figura 2.2 apresenta algumas implementações típicas de tolerância a falhas. A diferença entre mascaramento de erros e recuperação por avanço completa é que na primeira a redundância é utilizada sistematicamente (independentemente da presença ou ausência de erros detectados), enquanto na segunda a redundância é utilizada em demanda, depois da detecção de um erro ter sido realizada.

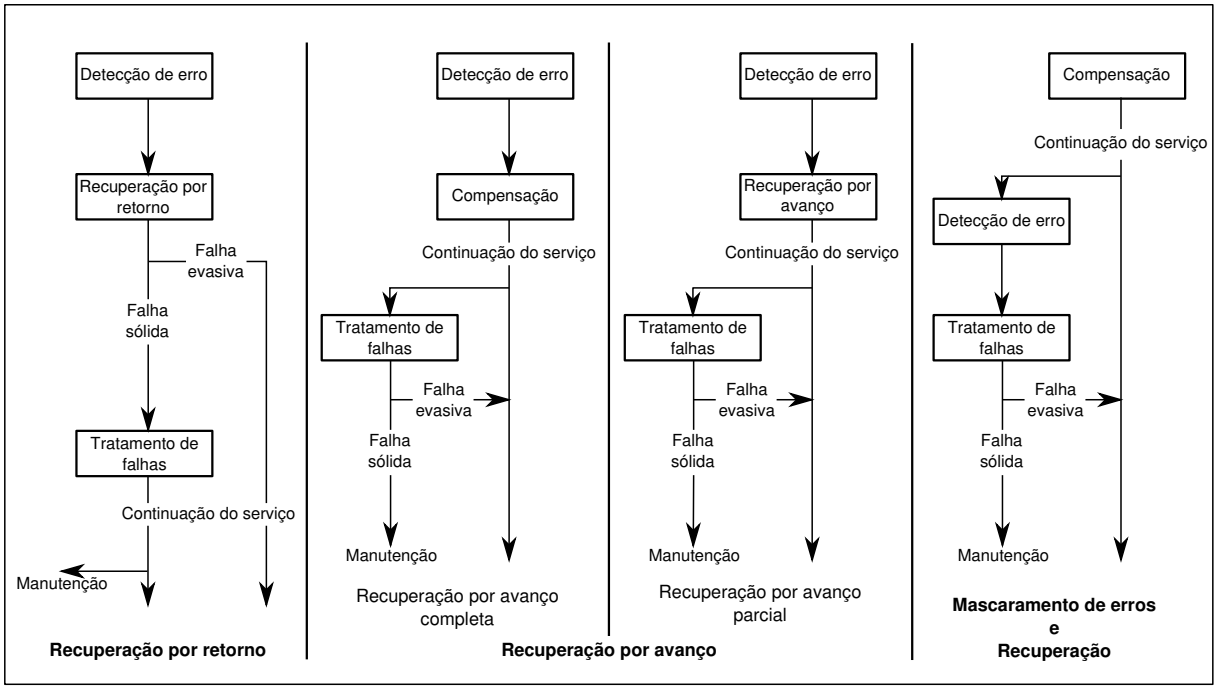


Figura 2.2: Implementações típicas de tolerância a falhas [8].

Os sistemas distribuídos tolerante a falhas normalmente se diferenciam quanto ao número de falhas que toleram e da classe de falhas que toleram. As duas classes mais comuns de falhas são falhas de parada e falhas bizantinas. As falhas de parada são

aquelas que causam a parada do processo, enquanto que as falhas bizantinas são aquelas introduzidas por um agente malicioso no ambiente do processo.

O algoritmo *FreeStore* implementado nesse trabalho utiliza-se extensivamente da técnica de mascaramento de falhas para atingir tolerância a falhas a um nível acima de cada um dos sistemas computacionais participantes do sistema. As falhas de parada inerentes às máquinas individuais de cada processo e aos canais de comunicação do sistema são mascaradas enquanto existe uma maioria de processos que não apresentam erros. Ou seja, existe uma tolerância a $\lfloor \frac{N-1}{2} \rfloor$ processos com erros em várias das operações realizadas pelo sistema, onde N é o número total de processos que formam o sistema.

2.2.3 Remoção de Falhas

A remoção de falhas pode ocorrer durante o desenvolvimento e durante o uso do sistema. Remoção de falhas durante o desenvolvimento envolve três passos: verificação, diagnóstico e correção. A verificação é o processo de checar se um sistema apresenta certas propriedades, e somente nos casos negativos realizar os outros passos. É recomendado que após a correção, a verificação seja realizada novamente para evitar a adição de novas falhas. São várias as técnicas de verificação e muitas delas podem ser utilizadas de forma complementar. A Figura 2.3 apresenta as técnicas de verificação, divididas nas características que diferenciam uma técnica da outra.

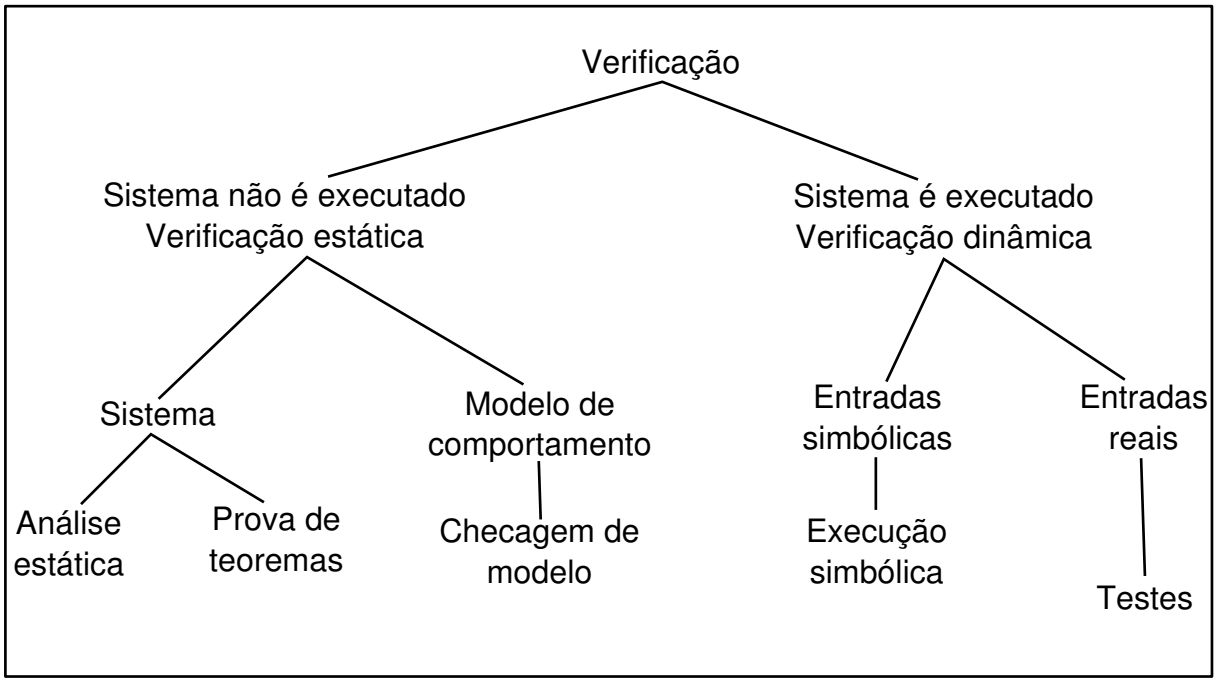


Figura 2.3: Técnicas de verificação [8].

A checagem da especificação é normalmente chamada de validação (*validation*) [12]. A descoberta de falhas na especificação pode ser realizada em qualquer estágio do desenvolvimento.

Remoção de falhas durante o uso pode ser chamada de manutenção preventiva ou de manutenção corretiva. Na preventiva, o objetivo é descobrir e remover falhas que possam

causar erros durante a operação do sistema. Na corretiva, o objetivo é remover falhas reportadas que causaram um ou mais erros. Estas formas de manutenção valem para qualquer sistema , i.e., tanto para sistemas que toleram quanto para sistemas que não toleram falhas. Além disso, esta manutenção pode ser realizada sem ou com a interrupção do seu serviço. O fator que distingue tolerância a falhas de manutenção é que a manutenção é realizada por um agente externo.

2.2.4 Previsão de Falhas

A previsão de falhas é feita através da realização da avaliação do comportamento do sistema com respeito à ocorrência e ativação de falhas. A avaliação pode ser qualitativa ou quantitativa.

A avaliação qualitativa tem como objetivo identificar, classificar, e ranquear os modos de falhas, ou as combinações de eventos (defeitos nos componentes ou condições do ambiente) que podem levar a um defeito no sistema.

A avaliação quantitativa tem como objetivo estimar em termos de probabilidade a extensão em que alguns dos atributos do sistema são satisfeitos, os quais são então vistos como medidas.

2.3 Técnicas de Redundância em Sistemas Distribuídos

As técnicas de redundância em sistemas distribuídos variam em relação aos requisitos exigidos, à transparência da replicação e à consistência dos dados replicados. A transparência diz respeito à percepção que o cliente tem da replicação, normalmente o objetivo é que a replicação seja invisível para o cliente. A consistência diz respeito aos critérios de correção dos dados replicados e podem variar a medida que o grau de consistência é reduzido para melhorar outras propriedades do sistema, como a escalabilidade e o desempenho.

A seguir serão apresentadas as técnicas de replicação passiva, replicação ativa (replicação máquina de estado) e replicação através de sistemas de quóruns. Existem muitas outras maneiras de se replicar dados em um sistema distribuído, alguns com interessantes graus de consistência, mas esses três são os mais conhecidos. Para simplificar, a descrição de cada técnica assume que o número de réplicas não muda, ou seja, que é um sistema distribuído estático.

2.3.1 Replicação Passiva: *Primary-Backup*

A técnica de *primary-backup* utiliza uma réplica, a primária, para executar um papel especial: ela recebe as invocações dos clientes e responde. As outras réplicas são *backups*. *Backups* interagem diretamente apenas com a réplica primária, e não com os clientes. Uma aplicação distribuída replicada com essa técnica realiza a invocação de uma operação $op(arg)$ emitida por um cliente da seguinte forma [30]:

- A invocação $op(arg)$ é enviada apenas para a réplica primária junto com um identificador único de invocação.

- A réplica primária processa a invocação, atualiza seu próprio estado e envia uma mensagem de atualização para os *backups*. Essa mensagem possui o identificador da invocação, a resposta e a atualização no estado. Após processar a mensagem de atualização, os *backups* confirmam a atualização.
- Quando a réplica primária recebe a confirmação de todas as réplicas *backups* corretas (que não falharam), ela envia a resposta para o cliente.

Essa técnica requer uma primitiva de comunicação que garante atomicidade no envio das mensagens de atualização, ou seja, as mensagens de atualização devem ser recebidas por todos ou nenhum dos *backups*. A técnica de comunicação em grupo *view-synchronous* [45] é normalmente usada para satisfazer esse requisito da replicação passiva.

A falha de uma réplica *backup* é transparente para o cliente. A falha da réplica primária requer que o cliente reenvie a requisição para uma nova réplica primária que é escolhida dentre as réplicas *backups*, resultando em um atraso. Com essa técnica, o sistema tolera $N - 1$ falhas, sendo N o número total de réplicas.

2.3.2 Replicação Ativa: Replicação Máquina de Estado

A técnica da replicação máquina de estado consiste em dar a todas as réplicas o mesmo papel, sem um controle centralizado. Uma aplicação distribuída replicada com essa técnica realiza a invocação de uma operação $op(arg)$ emitida por um cliente da seguinte forma [30]:

- A invocação $op(arg)$ é enviada para todas as réplicas do sistema.
- Cada réplica processa a invocação, atualiza seu próprio estado, e retorna a resposta para o cliente.
- O cliente espera até receber a primeira resposta.

Essa técnica requer que todas as operações sejam determinísticas. Determinismo significa que o resultado de uma operação depende apenas do estado inicial e a sequência de operações já realizadas. Processos com várias *threads* normalmente leva ao não determinismo, o que pode ser um problema.

Outro requisito é que as réplicas precisam receber as mesmas invocações dos clientes e na mesma ordem (atomicidade e ordenamento total), a primitiva de comunicação que satisfaz esse requisito é chamada de difusão atômica. Difusão atômica é descrita na Seção 3.3.3. A técnica de comunicação em grupo *view-synchronous* [45] é normalmente usada para satisfazer esse requisito da replicação máquina de estado.

A falha de uma réplica é transparente para o cliente e o cliente não precisa reenviar um pedido. Com essa técnica, o sistema tolera $N-1$ falhas desconsiderando os protocolos necessários para ordenação de mensagens, sendo N o número de réplicas.

2.3.3 Sistemas de Quóruns

Um sistema de quóruns é uma técnica específica para implementar memória compartilhada. Isso significa que esse sistema permite apenas a operação de leitura e escrita sobre um registrador (o registrador pode ser um objeto). Existem várias variações de sistemas de quóruns com diferentes características, a seguir está descrita uma dessas variações.

A técnica consiste em dar a todas as réplicas o mesmo papel, sem um controle centralizado. Na verdade, o controle é realizado pelo cliente e as réplicas normalmente realizam apenas as seguintes operações com controle de concorrência:

- leitura $r(x)$: retorna $<val, ts>$ como resposta para o cliente, onde val é o valor do registrador x e ts o *timestamp* associado.
- escrita $w(x, val, ts)$: atualiza o registrador x com o valor val e *timestamp* ts.

O protocolo de leitura de um registrador x, realizado pelo cliente e essencial para o controle das réplicas, é da seguinte forma:

Fase 1:

1. Invoca $r(x)$ em uma maioria qualquer (quórum) das réplicas do sistema (pode ser todas).
2. O cliente espera até receber uma maioria qualquer de respostas e seleciona aquela com maior *timestamp* associado. Essa contém o valor correto do registrador $<val_h, ts_h>$.
3. Caso alguma das respostas seja diferente, o cliente precisa realizar a fase 2, senão retorna val_h.

Fase 2: (*write-back*)

1. Invoca $w(x, val_h, ts_h)$ em uma maioria qualquer das réplicas do sistema (pode ser todas).
2. O cliente espera até receber uma maioria qualquer de confirmações.
3. O cliente retorna val_h.

O protocolo de escrita de um valor $newval$ em um registrador x, realizado pelo cliente, é da seguinte forma:

Fase 1: (*read*)

1. Invoca $r(x)$ em uma maioria qualquer das réplicas do sistema (pode ser todas).
2. O cliente espera até receber uma maioria qualquer de respostas e seleciona aquela com maior *timestamp* associado. Essa contém o valor atual do registrador $<val_h, ts_h>$.
3. O cliente vai para a fase 2.

Fase 2: (*write*)

1. Invoca $w(x, newval, ts_h + 1)$ em uma maioria qualquer das réplicas do sistema (pode ser todas).
2. O cliente espera até receber uma maioria qualquer de confirmações para concluir a operação.

A técnica de sistemas de quóruns utiliza-se do fato de que em duas maiorias quaisquer de um conjunto, sempre terá pelo menos um elemento em comum. Isso significa que, com um meio de distinguir qual valor é o mais atual (o *timestamp*), um sistema de quóruns pode escrever em apenas uma maioria que qualquer leitura em uma maioria a seguir verá a atualização realizada.

Na operação de escrita, o cliente precisa ler de uma maioria para descobrir qual será o *timestamp* do novo valor do registrador. Na operação de leitura, o cliente precisa escrever em uma maioria para garantir que qualquer leitura a seguir (mesmo com novas falhas) também lerá o valor lido. Esses requisitos garantem consistência sequencial às operações do sistema (Seção 3.2.2).

Em um sistema de quóruns, as comunicações são realizadas apenas entre clientes e servidores que implementam o registrador, i.e., os servidores não comunicam-se entre si. Devido a este fato, um sistema de quóruns não exige nenhuma primitiva de comunicação especial como as outras técnicas.

Nesses sistemas, a falha de uma réplica é transparente para o cliente e o cliente não precisa reenviar um pedido. Com essa técnica, o sistema tolera $\lfloor \frac{N-1}{2} \rfloor$ falhas (uma minoria), sendo N o número de réplicas.

Capítulo 3

Desafios na Implementação de uma Aplicação Distribuída com Segurança de Funcionamento

A implementação de aplicações distribuídas apresenta vários novos desafios relacionados às características específicas dos sistemas distribuídos. Na Seção 3.1 descrevemos essas características, em seguida na Seção 3.2 apresentamos alguns dos desafios mais relevantes para esse trabalho, e por último na Seção 3.3, falamos sobre os problemas fundamentais que surgem quando se quer alcançar uma maior segurança de funcionamento em aplicações distribuídas.

3.1 Introdução

Um sistema distribuído é o resultado da integração de sistemas computacionais autônomos combinados de forma a atingir um objetivo comum. Interligados através de redes de comunicação, os computadores se comunicam e coordenam suas ações por meio de mensagens [16]. Isso significa que uma aplicação distribuída se utiliza fortemente de trocas de mensagem para atingir seus objetivos e é composto por diversos protocolos que regem as regras de comunicação entre os sistemas da aplicação.

A definição de sistemas distribuídos possui certas características significantes que diferencia o desenvolvimento de aplicações distribuídas dos outros tipos de aplicações. Essas características sãos [16]:

- Concorrência: Em uma rede de computadores, programas concorrentes é a norma. O sistema precisa lidar com várias coisas sendo realizadas ao mesmo tempo e em máquinas diferentes. Mecanismos para realizar a coordenação dos processos que compartilham recursos é de extrema importância nessas aplicações.
- Inexistência de um relógio global: Para aplicações distribuídas cooperarem, elas precisam coordenar suas ações através de trocas de mensagens. Coordenação normalmente depende de uma ideia compartilhada do tempo em que cada ação acontece. Infelizmente, devido ao fato de que a comunicação acontece apenas por trocas de mensagens através de um rede, não existe uma noção global de qual é o tempo correto.

- Falhas independentes: Todo sistema computacional pode falhar, e é parte do projeto do sistema planejar as consequências das possíveis falhas. Os sistemas distribuídos podem falhar de novas maneiras: falhas na rede pode resultar no isolamento de computadores que estão conectados a ela, mas isso não significa que eles pararam de executar. O fato é que os processos de uma aplicação distribuída podem não conseguir detectar se ocorreu uma falha na rede ou a mesma apenas está lenta. Componentes também podem falhar e não notificar os outros componentes do sistema.

Essas características originam muitos dos desafios na implementação de uma aplicação distribuída. Antes de falar sobre os desafios específicos dessas aplicações, é bom lembrar que o desenvolvimento de software em si já apresenta grandes desafios devido a sua complexidade, como dificuldades para:

- Prevenir as falhas: Metodologias de desenvolvimento para serem práticas não são perfeitas e permitem o surgimento de falhas.
- Remover as falhas: Não é possível encontrar todas as falhas, e falhas podem ser adicionadas enquanto estamos removendo outras falhas.
- Tolerar as falhas: Os mecanismos de tolerância a falhas em si podem apresentar falhas.
- Confiar nos componentes de prateleira (*off-the-shelf components*) que fazem parte dos nossos sistemas e que normalmente contém falhas.
- Confiar nas ferramentas de produtividade utilizadas no desenvolvimento que podem conter e adicionar falhas à aplicação.

A verdade é que é um grande desafio colocar em prática de maneira correta as técnicas de se obter segurança de funcionamento apresentadas no Capítulo 2. Os 8 principais desafios encontrados no desenvolvimento e execução de sistemas distribuídos são [16]:

- Escalabilidade
- Concorrência
- Tolerância a falhas
- Heterogeneidade
- Abertura (*Openness*)
- Segurança (*Security*)
- Transparência
- Qualidade de Serviço

3.2 Principais Desafios

A seguir, apresentamos uma discussão envolvendo os conceitos de escalabilidade, concorrência e tolerância a falhas em sistemas distribuídos.

3.2.1 Escalabilidade

A escalabilidade é a característica de um sistema que se mantém efetivo quando ocorre uma variação significante no número de recursos e usuários. Desenvolver uma aplicação escalável é um grande desafio pois muitos dos protocolos existentes apresentam uma perda de performance não linear com o aumento de recursos e usuários. Isso significa que a partir de um tamanho, o sistema não consegue fornecer o mesmo serviço e começa a apresentar uma performance inaceitável.

A verdade é que a escalabilidade é uma característica desejada mas que nem sempre é alcançada nos sistemas reais. Muitos requisitos para aplicações distribuídas conflitam com a escalabilidade do sistema. O nível de escalabilidade final depende de várias escolhas entre características conflitantes do sistema realizadas pelo projetista durante o desenvolvimento do sistema. Nos sistemas de quóruns, o grau de consistência implementado pode fazer toda a diferença entre um sistema escalável ou não. Um requisito rígido de manter todas as cópias de um dado fortemente atualizado é um dos maiores limitantes da escalabilidade do número de participantes da aplicação.

A escalabilidade de usuários normalmente se atinge com uma arquitetura hierárquica, onde não existe um gargalo único como um único servidor onde todos os usuários devem acessar. *Caching* é um exemplo de uma técnica para se obter escalabilidade de usuários, pois permite um descarregamento dos servidores principais com o uso de intermediários que podem responder pelos principais. O uso de vários servidores realizando a mesma tarefa é outra técnica bastante usada.

Churn

O *churn* de um sistema distribuído dinâmico é a variação no conjunto de participantes, i.e., os eventos de entradas e saídas arbitrárias de processos no sistema. A ocorrência desses eventos gera perturbações no sistema, as quais podem resultar em [27]: perda de mensagens, inconsistência de dados, aumento da latência observada pelo usuário, aumento no uso de largura de banda, interrupções parciais ou totais de serviço; indisponibilidade de recursos e variações bruscas no desempenho do sistema. Devido ao *churn*, a quantidade de processos em um sistema distribuído dinâmico pode aumentar significativamente, fazendo com que as aplicações necessitem apresentar escalabilidade em relação ao número de participantes nestes ambientes.

O *churn* é um desafio presente exclusivamente em sistemas distribuídos dinâmicos e que faz esses sistemas possuirem necessidade de arquiteturas auto-organizáveis, capazes de detectar e tratar as mudanças que podem ocorrer no conjunto de participantes. Essa propriedade de autoadaptação tem por objetivo assegurar que o sistema continuará atendendo, durante todo o ciclo de vida, aos requisitos de funcionamento da aplicação, evitando a deterioração da qualidade de serviço.

Tentativas de minimizar os efeitos do *churn* envolvem o uso de heurísticas na seleção dos processos usados para compartilhar determinado recursos ou prover algum serviço. Estas heurísticas consistem em escolher processos que possuem maior probabilidade de permanecerem por mais tempo no sistema, servindo como fontes confiáveis de recursos ou serviços.

Stutzbach et al. [48] considera o histórico dos processos para definir essas heurísticas e indica que existe uma forte correlação entre os tempos que determinado processo per-

manece no sistema. Assim, é possível estimar de maneira aproximada o tempo que um processo permanecerá no sistema com base em seus comportamentos prévios observados.

3.2.2 Concorrência e Consistência

Uma aplicação distribuída, de uma forma ou de outra, compartilha recursos entre os seus clientes. Devido a natureza independente dos participantes de um sistema distribuído, várias coisas podem acontecer ao mesmo tempo, como vários clientes tentando acessar o mesmo recurso simultâneamente. Essa concorrência merece bastante atenção, pois o descuido pode resultar em erros extremamente difíceis de serem diagnosticados.

Na implementação de aplicações distribuídas, é de grande importância a realização de controle de concorrência nos acessos às regiões críticas, ficando atento aos possíveis problemas de condições de corrida (*race conditions*) e *deadlocks*. Esses problemas são mais complicados nos sistemas distribuídos, pois é preciso de uma solução para exclusão mútua baseada apenas por troca de mensagens.

Um meio de se obter exclusão mútua que vem ganhando bastante popularidade é a utilização de uma aplicação distribuída própria para fornecer serviço de sincronização. Essa aplicação permite realizar a coordenação necessária do sistema mas se torna uma dependência crítica aos seus clientes e precisa apresentar segurança de funcionamento exemplar. Uma aplicação bem conceituada que fornece serviço de sincronização é o Apache ZooKeeper [4, 32].

Um outro requisito que surge da concorrência com o uso de replicação de dados, e que pode variar em intensidade entre as aplicações distribuídas é o de consistência. Consistência diz respeito se as operações realizadas sobre uma coleção de objetos replicados produzem resultados que satisfazem os critérios de correção dos objetos. Existem diferentes critérios de correção para objetos replicados, como o de linearizabilidade (*linearizability*), a consistência sequencial (*sequential consistency*), coerência (*coherence*) e variações da consistência fraca (*weak consistency*) [17].

Linearizabilidade

Esse é o critério mais rigoroso, e devido a natureza assíncrona dos sistemas distribuídos, nem sempre é possível satisfazer na realidade. Na literatura de memória compartilhada distribuída a linearizabilidade é normalmente chamada de consistência atômica (*atomic consistency*).

Um serviço de objetos compartilhados replicados é dito ser linearizável se para toda execução existe uma intercalação das operações enviadas por todos os clientes que satisfaz os seguintes critérios [17]:

L1 A intercalação das operações segue a especificação de uma única copia correta do objeto. Ou seja, uma leitura retorna o valor da última operação de escrita ou se não houver escrita anterior, retorna o valor inicial.

L2 A ordem das operações na intercalação é consistente com o tempo real em que as operações ocorreram na execução real.

Esse critério não é possível ser sempre satisfeito na realidade pois não existe um relógio global para decidir o tempo real em que cada operação aconteceu.

Consistência sequencial

Esse é o critério mais rigoroso usado na prática. Um serviço de objetos compartilhados replicados é dito ser consistente sequencialmente se para toda execução existe uma intercalação das operações enviadas por todos os clientes que satisfaz os seguintes critérios [17]:

SC1 A intercalação das operações segue a especificação de uma única copia correta do objeto. Ou seja, uma leitura retorna o valor da última operação de escrita ou se não houver escrita anterior, retorna o valor inicial.

SC2 A ordem das operações na intercalação é consistente com a ordem no programa de cada cliente que as executou.

O critério SC1 é o mesmo que o L1. O critério SC2 se refere a ordem no programa ao invés da ordem temporal, que é exatamente o que torna esse critério implementável. Note que todo o sistema precisa ser considerado para avaliar se o sistema satisfaz as condições da consistência sequencial.

Coerência

Esse critério alivia os custos de SC2 exigindo o ordenamento apenas para escritas ao mesmo registrador. Pode ser visto como uma consistência sequencial por registrador. Dessa forma, dois registradores diferentes são independentes e não atrasam um ao outro.

Consistência fraca

Esse critério é uma tentativa de evitar os custos da consistência sequencial enquanto mantém os efeitos da mesma.

As intercalações citadas nas definições não precisam realmente acontecer, as definições apenas estipulam que a execução deve acontecer como se fosse uma intercalação satisfazendo as condições.

3.2.3 Tolerância a Falhas

Falhas podem acontecer de maneiras diferentes em um sistema distribuído, e é importante ter meios para tolerá-las.

Partições na rede é um tipo de falha exclusivo desses sistemas, e acontece por exemplo, quando um roteador da rede para de funcionar e impossibilita comunicação entre os dois lados da rede dividida pelo roteador. Em uma rede ponto a ponto como a Internet, topologias complexas e escolhas de roteamento independentes significa que a conectividade pode ser assimétrica: é possível comunicar do processo p para o processo q, mas não ao contrário. Conectividade pode também ser intransitiva: é possível comunicar de p para q, e de q para r, mas p não pode comunicar diretamente com r [16]. Logo, os processos podem não conseguir se comunicar ao mesmo tempo. Temos a premissa de confiabilidade que implica que eventualmente qualquer falha de *link* ou roteador será consertada ou contornada.

Como visto no Capítulo 2, existem várias técnicas para buscar tolerância a falhas, sendo a mais comum a replicação dos componentes do sistema. Na replicação dos componentes, é comum considerar que o defeito de uma réplica é uma falha do sistema e que

uma réplica pode apresentar defeito de duas maneiras diferentes: por parada (a réplica para de enviar e receber mensagens) ou por apresentar comportamento malicioso (a réplica envia mensagens que não estão de acordo com a especificação do sistema). Nesse trabalho, consideraremos apenas as falhas por parada.

Um requisito comum para uma aplicação tolerante a falhas é a transparência de replicação e a transparência de falhas. A transparência de replicação exige que as várias cópias do mesmo dado sejam apresentadas como uma só. Isso é, o cliente realiza operações sobre os dados como se existisse apenas uma cópia de cada dado. A transparência de falhas é o mascaramento das falhas de forma a esconder qualquer falha tolerável pelo sistema.

Um serviço tolerante a falhas sempre garante comportamento estritamente correto apesar de um certo número de falhas e partições na rede. Comportamento correto diz respeito à consistência dos dados fornecidos aos clientes, como descrito na Seção 3.2.2. Comportamento correto as vezes também diz respeito ao tempo de demora das respostas do serviço, em casos que demoras possam ser fatais. O sistema tolerante a falhas precisa coordenar seus componentes para manter o comportamento correto mesmo com falhas, que podem acontecer a qualquer momento.

Um desafio de qualquer aplicação tolerante a falhas é estruturar o programa de forma a manter uma separação entre código funcional, que implementa o serviço do sistema e o código não funcional, como o código que implementa a tolerância a falhas. Existem *frameworks* que procuram ajudar nesse desafio, como por exemplo a biblioteca BFT-SMaRt [10], que procura facilitar e organizar o desenvolvimento de aplicações tolerante a falhas que utilizam o método de replicação Máquinas de Estados (Seção 2.3.2).

Uma abstração útil para facilitar na estruturação de uma aplicação tolerante a falhas é a de grupo. Essa abstração provê um *framework* para prover primitivas de comunicação especiais aos processos de um grupo, como as exigidas pelas técnicas de replicação passiva e replicação máquina de estado descritas na Seção 2.3. Essa estrutura de grupos foi primeiramente apresentada em [11] e hoje está presente em vários outros sistemas distribuídos tolerante a falhas [42].

3.3 Problemas Clássicos em Sistemas Distribuídos

Esta seção apresenta os principais problemas clássicos encontrados no desenvolvimento de aplicações distribuídas. Estes problemas são chamados de clássicos (ou fundamentais) porque são encontrados no desenvolvimento de qualquer aplicação distribuída com requisitos de segurança de funcionamento. Os problemas normalmente possuem soluções centralizadas e distribuídas.

As soluções centralizadas normalmente são implementadas por um sistema que fornece a solução como serviço. Esses sistemas não chegam a ficar muito complexos mas precisam apresentar uma exemplar segurança de funcionamento para não prejudicar os seus clientes.

As soluções distribuídas, por sua vez, são normalmente compostas por um conjunto de protocolos realizados pelos próprios processos que precisam da solução. Essas soluções normalmente apresentam uma maior complexidade que as soluções centralizadas.

3.3.1 Filiação

O conceito de grupo aparece em diversos protocolos dos sistemas distribuídos, por isso da importância de soluções para o problema de filiação (*membership*). Esse problema pode ser definido como um acordo realizado entre os processos de um grupo para saber, em tempo de execução, quem pertence e quem não pertence ao grupo, determinando assim a composição do mesmo [18, 19].

A lista de processos pertencentes a um grupo, chamada de **visão**, pode ser estática ou dinâmica, a mesma distinção entre sistemas distribuídos estáticos e dinâmicos.

O **serviço de pertinência** é o componente do sistema distribuído que fornece a solução para o problema da filiação, ou seja, esse serviço é responsável por manter a coordenação entre todos os processos a respeito de quem pertence ao grupo em um dado instante. Um serviço de pertinência normalmente apresenta as seguintes três primitivas:

- $join(G, p)$: inclui o processo p no grupo G;
- $leave(G, p)$: remove o processo p do grupo G;
- $view(G, V^i)$: invocada pelo serviço de pertinência para enviar uma nova visão V^i para os processos do grupo G. Dizemos que o processo $p \in G$ que recebe V^i **instala** a visão V^i.

Estas primitivas devem atender as seguintes propriedades [29, 44]:

- **Própria inclusão:** Se um processo p instala a visão V^i então $p \in V^i$;
- **Monotonicidade local:** Se um processo p instala V^j depois de instalar V^i então $j > i$;
- **Visão inicial:** Todos os processos corretos instalam a visão $V^0 = G$ (configuração inicial);
- **Acordo:** Se um processo correto instala V^i então todos os processos corretos acabam por instalar V^i;

Um serviço de pertinência pode utilizar um detector de falhas [14] para monitorar quais processos estão corretos ou defeituosos. Existem diversos tipos de detectores de falhas, que se diferem na precisão (exatidão) e completude das suas respostas. Devido a essa interação com detectores de falhas, alguns autores [29, 44] consideram também as seguintes propriedades:

- **Completude:** Se um processo p é defeituoso, então uma visão V^i instalada por algum processo correto terminará por não conter p, i.e. $p \notin V^i$;
- **Precisão:** Se um processo q instala uma visão V^i e sendo p um processo tal que $p \notin V^i$ então p é defeituoso.

Estas propriedades dizem respeito a serviços de **pertinência não particionável** onde todos os processos corretos concordam com uma mesma visão do sistema, ao contrário dos sistemas com serviço de **pertinência particionável** onde admite-se várias visões diferentes do grupo, tolerando-se inclusive partições na rede, onde define-se operações

de junção (*merge*) a serem usadas quando os casos de particionamento forem tratados, regenerando o grupo original [35]. A grande diferença, em termos de especificação do problema, diz respeito a inclusão de uma nova operação $merge(G, V^i, V^j)$ para a junção de partições distintas e a modificação da propriedade de Acordo (uma vez que agora nem todos os processos instalam a mesma visão) e Precisão (nestas condições de sistema nem todo processo não pertencente a uma visão pode ser considerado defeituoso).

Um serviço de pertinência pode ser implementado pelos próprios processos que fazem parte do grupo ou através de um sistema próprio que fornece esse serviço. Cada alternativa tem seus pontos positivos e negativos, a primeira opção requer um protocolo de filiação que é um desafio em si enquanto a segunda é um outro sistema que se torna uma dependência crítica aos seus clientes e precisa apresentar segurança de funcionamento exemplar. Uma aplicação bem conceituada que fornece serviço de pertinência é o Apache ZooKeeper [4, 32]. O principal protocolo estudado nesse trabalho, o *FreeStore*, utiliza um protocolo de filiação executado pelos próprios processos do sistema sendo reconfigurado.

3.3.2 Consenso

Em um sistema distribuído, formado por vários processos independentes, o problema do consenso consiste em fazer com que todos os processos corretos acabem por decidir o mesmo valor, o qual deve ter sido previamente proposto por algum dos processos do sistema. Formalmente, este problema é definido em termos de duas primitivas [31]:

- *propose(G,v)*: o valor v é proposto ao conjunto de processos G.
- *decide(v)*: executado pelo protocolo de consenso para notificar ao(s) interessado(s) (geralmente alguma aplicação) que v é o valor decidido.

Para estas primitivas satisfazerem a correção (*safety*) e a vivacidade (*liveness*), as mesmas devem verificar as seguintes propriedades [5, 7]:

- **Acordo**: Se um processo correto decide v, então todos os processos corretos terminam por decidir v.
- **Validade**: Um processo correto decide v somente se v foi previamente proposto por algum processo.
- **Terminação**: Todos os processos corretos terminam por decidir.

A propriedade de acordo garante que todos os processos corretos decidem o mesmo valor. A validade relaciona o valor decidido com os valores propostos e sua alteração dá origem a outros tipos de consensos. As propriedades de acordo e validade definem os requisitos de correção (*safety*) do consenso, já a propriedade de terminação define o requisito de vivacidade (*liveness*), que garante que o protocolo de consenso progredi.

Um dos trabalhos mais importantes envolvendo o problema do consenso é [22], onde é provada a impossibilidade de se resolver este problema em um sistema distribuído completamente assíncrono onde pelo menos um processo pode ser faltoso (qualquer tipo de falta). Em vista disso, a maioria das propostas encontradas na literatura consideram algum comportamento temporal do sistema, onde componentes (processos e/ou canais)

obedecem a requisitos temporais. Estes requisitos geralmente estão relacionados com *timeouts* encapsulados em *detectores de falhas* [14] ou são atendidos a partir da ocorrência de possíveis comportamentos síncronos do sistema subjacente.

3.3.3 Difusão Atômica

O problema da difusão atômica [31], conhecido também como difusão com ordem total, consiste em fazer com que todos os processos corretos, membros de um grupo, entreguem todas as mensagens difundidas neste grupo na mesma ordem.

A difusão atômica é definida sobre duas primitivas básicas [31]:

- *A-multicast*(G, m) ou *TO-multicast*(G, m): primitiva utilizada para difundir a mensagem m no grupo G.
- *A-deliver*(p, m) ou *TO-deliver*(p, m): chamada pelo protocolo de difusão atômica para entregar à aplicação a mensagem m, difundida pelo processo p.

Formalmente, um protocolo de difusão atômica deve satisfazer as seguintes propriedades [31]:

- **Validade:** Se um processo correto difundiu m no grupo G, então algum processo correto pertencente à G terminará por entregar m ou nenhum processo pertencente à G está correto;
- **Acordo:** Se um processo correto pertencente a um grupo G entregar a mensagem m, então todos os processos corretos pertencentes a G acabarão por entregar m;
- **Integridade:** Para qualquer mensagem m, cada processo correto pertencente ao grupo G entrega m no máximo uma vez e somente se m foi previamente difundida em G;
- **Ordem Total Local:** Se dois processos corretos p e q entregam as mensagens m e m' difundidas no grupo G, então p entrega m antes de m' se e somente se q entregar m antes de m'.

Um resultado teórico interessante é que a difusão atômica e o consenso são problemas equivalentes em sistemas onde os processos então sujeitos tanto a falhas de parada [14] quanto a falhas bizantinas [15]. *Joseph et al.* [9] propuseram um protocolo para difusão atômica em ambientes dinâmicos, considerando apenas a possibilidade de falhas por parada de processos.

Capítulo 4

Implementação do *FreeStore*

Um dos objetivos desse trabalho é a implementação dos protocolos de reconfiguração de sistemas de quóruns *FreeStore*. Esse capítulo apresenta o *FreeStore* e vários detalhes da implementação realizada. A Seção 4.2 discute a linguagem de programação utilizada. A especificação da implementação se encontra na Seção 4.3 e logo em seguida descrevemos a implementação. A Seção 4.5 aborda algumas das técnicas de engenharia de software utilizadas. Por fim, outras implementações realizadas nesse trabalho são apresentadas na Seção 4.6. Os resultados coletados dos experimentos realizados com estas implementações serão apresentados no Capítulo 5.

4.1 Visão Geral do *FreeStore*

O *FreeStore* [2] compreende um conjunto de protocolos para a realização de reconfigurações em sistemas de quóruns. Sua organização é modular e apresenta uma desejável separação de conceitos diferenciando claramente os papéis de geradores de sequências de visões [1], reconfiguração e leitura/escrita no registrador.

O *FreeStore* introduz a ideia de geradores de sequências de visões, cuja finalidade é englobar todos os requisitos de sincronia necessários para a tarefa de reconfiguração. O resultado da execução destes geradores são sequências de visões que permitem a realização da reconfiguração do sistema. Basicamente, uma visão nada mais é do que um conjunto com os identificadores dos servidores presentes no sistema.

Um gerador de sequência de visões pode ser construído de pelo menos duas formas diferentes, de acordo com o nível de sincronia que o ambiente de execução deve apresentar. A primeira utiliza um protocolo de consenso de forma que todos os servidores envolvidos concordam (entram em acordo) com a nova sequência de visões a ser instalada. Na verdade todos os servidores geram uma única sequência com uma única visão para ser instalada no sistema. O protocolo de consenso [37] é bem conhecido e muito utilizado mas possui a limitação de não ser implementável em um sistema distribuído completamente assíncrono. Outra possível construção dos geradores de sequência de visões é sem o uso de consenso, que não adiciona nenhuma limitação quanto ao modelo em que a sua implementação pode ser executada. Uma das motivações do trabalho é a análise na prática dos ganhos e perdas de cada um desses geradores de sequência de visões.

[1] O conceito e a forma como as visões foram implementadas são discutidos na Seção 4.4.1

Em termos gerais, o protocolo de reconfiguração do *FreeStore* funciona da seguinte forma:

Primeira etapa
Recebe pedidos de entrada (*join*) e saída (*leave*) do sistema. Esses pedidos são acumulados por um período de tempo até que o sistema passe para a segunda etapa.

Segunda etapa
Escolhe o gerador de visão a ser executado e inicializa a geração de uma nova sequência de visões para atualizar o sistema. As visões desta sequência sempre são mais atuais do que a visão atual instalada no sistema, contemplando os pedidos de atualizações coletados na primeira etapa. O protocolo vai para a terceira etapa quando essa sequência é gerada.

Terceira etapa
A sequência gerada na segunda etapa é instalada de forma que, no final, uma única visão é efetivamente instalada no sistema.

O *FreeStore* garante as seguintes propriedades [2]:

- **Armazenamento (Segurança)** Os protocolos de leitura e escrita satisfazem as propriedades de segurança de um registrador de leitura e escrita atômico [36].
- **Armazenamento (Terminação)** Todas operações de leitura ou escrita executadas por clientes correto terminam.
- **Reconfiguração - *Join* (Segurança)** Se um servidor j instala a visão v tal que $i \in v$, então o servidor i executou a operação *join* ou é membro da visão inicial.
- **Reconfiguração - *Leave* (Segurança)** Se um servidor j instala a visão v tal que $i \notin v \wedge (\exists v' : i \in v' \wedge v' \subset v)$, então o servidor i executou a operação *leave*.
- **Reconfiguração - *Join* (Terminação)** O evento *enable operations* terminará por ocorrer em todo servidor correto que executa a operação *join*.
- **Reconfiguração - *Leave* (Terminação)** O evento *disable operations* terminará por ocorrer em todo servidor correto que executa a operação *leave*.

O *FreeStore* possui um modelo de chegada de servidores que permite que os servidores do sistema possam ser inicializados a qualquer momento e não apresenta limites aos tempos de transmissão e/ou processamento, sendo assim implementável em um sistema assíncrono. Para uma leitura mais aprofundada sobre os protocolos do *FreeStore*, e o protocolo em si, ver o artigo original em [2].

4.2 Linguagem de Programação da Implementação: *Go*

A implementação foi realizada na linguagem de programação ***Go*** [28]. *Go* é uma linguagem aberta, compilada, concorrente e com coletor de lixo que surgiu em 2009 e segue os seguintes princípios:

- Eficiência de uma linguagem compilada com tipagem estática com a facilidade de programação de uma linguagem dinâmica.
- Segurança (*safety*): de tipo e de memória.
- Concorrência intuitiva através de *goroutines* e canais para comunicação entre elas.
- Coletor de lixo eficiente.
- Compilação rápida.
- Desenvolvida tendo ferramentas em mente.

Go apresenta várias características de programação concorrente, como por exemplo *goroutines*, que provê uma abstração eficiente de threads. Uma *goroutine* é facilmente criada para executar uma função com a instrução *go*, como em:

```
1 go func()
```

O custo de criação de uma *goroutine* é muito baixo comparado com uma thread do padrão *pthread*, o que as diferencia das *threads* tradicionais. Devido a isso, o programador é incentivado a pensar em como solucionar os problemas de forma concorrente, o que pode muitas vezes resultar em um *design* muito mais natural e de mais fácil compreensão do que na programação tradicional. Estruturar um programa de forma concorrente as vezes se assemelha a programar um sistema distribuído local, o que é familiar aos desenvolvedores da área.

A linguagem apresenta vários aspectos que aumentam a legibilidade de código em *Go*, facilitando a manutenção do código e gerando um impacto positivo à segurança de funcionamento do software. Uma outra característica interessante de um código em *Go* é que funções podem retornar múltiplos valores. Essa funcionalidade permite uma nova forma de tratamento de erros de maior legibilidade, fugindo do tratamento de exceções característico do *Java*. No *Java*, um erro é uma exceção que modifica o fluxo de execução do programa, enquanto que em *Go*, um erro é um tipo de valor como um outro qualquer. As funções em *Go* falham retornando um erro não nulo e uma simples instrução *if* separa o código de tratamento de erro do código normal. Por exemplo:

```
1 valor, erro = LerValor()
2 if erro != nil {
3     // Tratamento do erro
4     log.Println("Erro ao ler")
5     valor = valorPadrao
6 }
7
8 // Código que utiliza o valor
9 ...
```

Outra nova funcionalidade bastante usada em *Go* é a de canais. Canais é um tipo de dado com controle de concorrência implementado pela linguagem e utilizado na comunicação entre *goroutines*, onde uma *goroutine* envia algo pelo canal e outra *goroutine* recebe. Essa comunicação pode ser síncrona ou assíncrona, o que permite vários interessantes usos para realizar a coordenação entre as *goroutines*.

Um dos focos da linguagem foi tornar fácil a criação de ferramentas destinadas a facilitar o desenvolvimento em *Go*. Por consequência, várias ferramentas existem e são usadas por grande parte dos programadores, por exemplo *gofmt*, que formata código em *Go* de forma consistente e permite que os programas na linguagem tenham aparência similar, o que aumenta bastante a legibilidade do código. Outras ferramentas altamente disseminadas permitem apresentação de documentação, a instalação e a execução de testes com nada mais além do código fonte em *Go*.

Go foi escolhida a linguagem da implementação desse trabalho devido a essas características citadas e por estar sendo muito utilizada no desenvolvimento de sistemas distribuídos. Um exemplo de aplicação desenvolvida em *Go* é *NSQ* [41], uma plataforma distribuída para troca de mensagens em tempo real com tolerância a falhas e alta disponibilidade utilizada em produção em várias empresas. *Go* também foi reportado ser usado dentro de vários projetos internos no *Google*, onde possui vários desenvolvedores trabalhando exclusivamente no desenvolvimento da linguagem.

4.3 Especificação da Implementação

As implementações realizadas nesse trabalho seguem as especificações descritas nessa seção. Aqui são apresentados os requisitos funcionais e não-funcionais, a arquitetura, as *APIs*, as premissas assumidas para o correto funcionamento do sistema e o modelo de tolerância a falhas.

4.3.1 Requisitos

O sistema possui os seguintes requisitos:

Registrador R/W
: Essa é a funcionalidade primária do sistema. O serviço do sistema é prover armazenamento de dados através de um registrador de leitura e escrita. O dado do registrador é acessado com a operação de leitura e modificado com a operação de escrita.

  O registrador precisa se comportar como se não fosse replicado, ou seja, a leitura do registrador deve retornar o valor da última operação de escrita conforme a consistência sequencial vista na Seção 3.2.2.

Reconfiguração Dinâmica
: O sistema é uma aplicação distribuída e precisa apresentar a possibilidade de inclusão e remoção de servidores que participam do sistema durante sua execução, i.e., sem que seja necessário parar o sistema.

  A reconfiguração deve ser realizada de forma transparente para os clientes do sistema, mantendo o correto fornecimento de seu serviço e afetando de forma mínima possível o desempenho do sistema.

Tolerância a falhas
: O sistema precisa apresentar tolerância a falhas enquanto houver uma maioria de servidores sem erros. Ou seja, o sistema precisa fornecer seu serviço corretamente

enquanto houver um número de falhas menor ou igual a $\lfloor \frac{N-1}{2} \rfloor$, onde N é o número de servidores presentes no sistema em um dado momento. Como o número de servidores integrantes do sistema varia, esse limite também varia. Durante as reconfigurações existem períodos onde duas ou mais visões estão ativas no sistema (por exemplo, quando os servidores de uma visão antiga estão enviando seus estados para um visão mais atual). Nesses períodos, o limite de falhas deve ser satisfeito para cada uma das visões. Por outro lado, sempre teremos apenas uma visão instalada no sistema onde as operações dos clientes são executadas e esse limite de falhas deve ser respeitado.

4.3.2 Arquitetura

A implementação representa um sistema de quóruns com capacidade de reconfiguração e é dividida em clientes e sistema:

Cliente
O cliente é a parte da implementação que contém a lógica de leitura e escrita na memória compartilhada. O cliente tem papel essencial para garantir o correto funcionamento do sistema e a tolerância a falhas.

Sistema
O sistema fornece a memória compartilhada. O sistema é replicado e pode ser constituído de um número qualquer de servidores executando ao mesmo tempo em plataformas diferentes. O sistema é reconfigurável e falhas de servidores são toleradas.

A Figura 4.1 ilustra a arquitetura da implementação, onde o conjunto dos servidores replicados constitui o sistema e cada cliente diferente pode utilizar o sistema simultaneamente.

4.3.3 *APIs*

O cliente e o sistema são implementados em forma de bibliotecas para facilitar a integração do sistema dentro de aplicações maiores que precisem de memória compartilhada. As bibliotecas apresentam as seguintes *APIs*:

Cliente

Funções

Read() (int, error)
Retorna o valor armazenado no sistema e um erro, o valor só é válido se o erro for nulo.

Write(v int) error
Escreve o valor v no sistema, retorna um erro caso alguma coisa errada ocorrer.

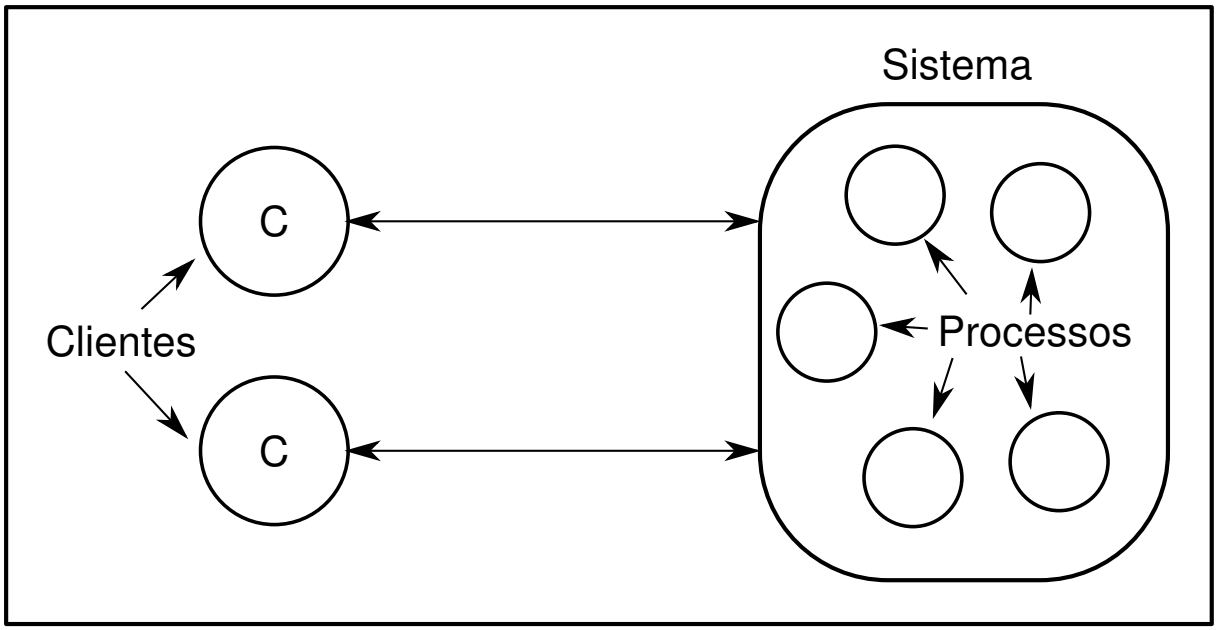


Figura 4.1: Arquitetura da implementação.

Sistema

Funções

Run(bindAddr string, initialView *view.View, useConsensus bool)
Inicia a execução de um servidor do sistema.

Parâmetros

bindAddr: o endereço em que o servidor irá fornecer seus serviços.

initialView: visão inicial do sistema. Se o servidor for membro dessa visão inicial, todos os seus serviços serão ativados no final da inicialização, caso contrário, o servidor irá enviar o pedido de *Join* para todos os servidores dessa visão.

useConsensus: se *true*, o servidor irá utilizar o gerador de sequência de visões com consenso, caso contrário não utilizará o protocolo do consenso. Esse parâmetro deve ser igual em todos os servidores em execução do sistema.

4.3.4 Premissas

Nessa seção são apresentadas as premissas assumidas em relação ao ambiente em que o sistema será executado. Essas premissas estão dividas em premissas do *FreeStore*, as quais são necessárias para que seus protocolos funcionem corretamente, e premissas da implementação, as quais são limitações de nossa implementação devido a simplificações adotadas na fase de projeto do sistema.

Premissas do *FreeStore*

O *FreeStore* possui as seguintes premissas quanto ao ambiente em que o sistema será executado [2]:

- **Cada servidor possui um identificador único.**

  Essa premissa foi satisfeita identificando os servidores pelo seu endereço da rede junto com a sua porta e restringindo a execução dos servidores do sistema à uma mesma rede local. Outra forma seria um administrador atribuir os identificadores aos servidores durante suas inicializações.

- **Utiliza um protocolo de comunicação confiável (*Reliable communication protocol*):** que retransmite mensagens perdidas ou corrompidas.

  Essa premissa foi satisfeita realizando toda comunicação entre os servidores do sistema através do protocolo *TCP/IP*, que retransmite mensagens perdidas ou corrompidas.

- **Limite no número de falhas:** Para cada visão ativa v, tolera não mais do que $\lfloor \frac{N-1}{2} \rfloor$ falhas, onde N é o número de servidores presentes na visão.

- **Tamanho do quórum:** Para cada visão v, consideramos um quórum de tamanho de $\lceil \frac{N+1}{2} \rceil$, onde N é o número de servidores presentes na visão.

- **Saídas suaves:** Um servidor correto $i \in V(t)$ ($V(t)$ representa conceitualmente a visão atual do sistema) que pede para sair do sistema no tempo t permanece no sistema até que ganhe o conhecimento de que uma visão mais atualizada $V(t'), t' > t, i \notin V(t')$ foi instalada.

- **Reconfigurações finitas:** O número de pedidos de reconfiguração (*joins/leaves*) em uma execução é finito. Isso é necessário para garantir que as operações dos clientes eventualmente completam.

Premissas da Implementação

A implementação possui as seguintes premissas quanto ao ambiente em que o sistema será executado. Note que estas premissas foram utilizadas apenas para não desviar o foco desse trabalho, que é a análise sobre as formas de reconfiguração de memória compartilhada através de sistemas de quóruns em sistemas distribuídos dinâmicos.

- **O servidor líder não falha:** Essa premissa permite predeterminar um servidor para sempre ser o líder do protocolo do consenso e sempre enviar a visão atual do sistema para os novos servidores que forem sendo inicializados. Essa limitação é resolvida através do emprego de algum protocolo de eleição de líder [24, 47], que escolhe outro servidor correto como líder quando o líder atual falha.

- **Sem falhas maliciosas no canal de comunicação:** Essa premissa permite que as comunicações sejam realizadas sem se preocupar com criptografia.

  Essa premissa é satisfeita apenas na situação em que as plataformas computacionais e os canais de computação do sistema são confiáveis, ou seja, não apresentam

conflitos de interesse [43]. Em sistemas reais não se deve considerar que os canais de comunicação sejam confiáveis e o uso de criptografia para integridade e confidencialidade é obrigatório.

- **Todos os servidores do sistema são executados dentro de uma mesma rede local:** A implementação utiliza o *IP* e a porta que um servidor utiliza como seu identificador. Esse identificador deixa de ser único quando os servidores estão em redes diferentes e algum roteador entre elas utiliza *NAT* [33].
- **Todos os servidores do sistema utilizam o mesmo tipo de gerador de sequência de visões:** Essa condição é essencial para o correto funcionamento de reconfigurações do sistema. Servidores com diferentes geradores de sequência de visões não conseguem interagir entre si na hora da reconfiguração.

4.3.5 Modelo de Tolerância a falhas

O modelo de tolerância a falhas é a especificação de tolerância a falhas do sistema. A intenção é ser explícito em relação às falhas que não tratamos, às que detectamos e às que toleramos.

O sistema tem como única falha detectada mas não tratada a falha da premissa de que em todo instante, existe uma maioria de servidores sem falhas. Nesse caso, a operação pode ter efeito não determinístico, e o usuário (o programa que utiliza a biblioteca) deve parar de utilizar o sistema a partir desse ponto.

O sistema tolera várias possíveis falhas relacionadas as suas operações desde que estejam de acordo com a premissa de que sempre existe uma maioria de servidores sem falhas, exemplos de falhas toleradas são:

- **Quedas de servidores:** Seja por falta de energia, falhas de software ou de hardware.
- **Falhas de comunicação do TCP detectadas:** Um exemplo é a impossibilidade de abrir uma conexão TCP com um dos servidores.

O sistema não tolera as seguintes falhas:

- **Falhas bizantinas:** Um servidor que sofre este tipo de falha pode apresentar um comportamento arbitrário.
- **Falhas não toleradas pelo Sistema Operacional e pelo Hardware:** O caso de uma falha silenciosamente levar a plataforma a corromper dados do sistema não é tolerada pela implementação. Esse erro é similar a uma falha bizantina, ao qual não toleramos.
- **Falhas de comunicação do TCP não toleradas:** Um exemplo é os dados a serem comunicados serem corrompidos de forma que os mecanismo de detecção de erro do protocolo TCP não os detecte e ainda represente um pedido ou resposta válida da implementação de *RPC* utilizada.

O tratamento das falhas toleradas pelo sistema é realizado no cliente do sistema e esse é o principal motivo pelo qual os usuários do sistema devem utilizar a biblioteca de cliente desenvolvida.

As falhas toleradas que caem na categoria de queda de servidores podem ser facilmente reparadas com a reinicialização do servidor (com o mesmo identificador) ou adicionando um novo servidor à visão no lugar do antigo (removendo o servidor que falhou). O protocolo de leitura e escrita irá consertar (atualizar) o registrador do servidor na próxima operação realizada.

4.4 Implementações

O sistema pode ser dividido em módulos que se interagem conforme na Figura 4.2. Esses módulos são: visão atual, registrador R/W, reconfiguração e gerador de visões. O registrador R/W precisa da visão atual para saber se um pedido é atual ou não. O módulo de reconfiguração altera a visão atual e o valor no registrador ao final de cada reconfiguração e invoca o gerador de visões que, por sua vez, retorna uma nova sequência de visões quando gerada.

A próxima seção apresenta as estruturas básicas usadas nas implementações. As seções seguintes apresentam as implementações dos protocolos de R/W e de reconfiguração.

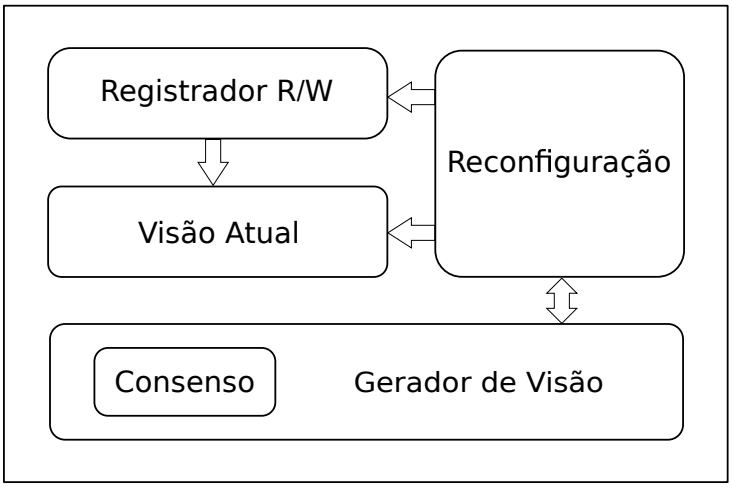


Figura 4.2: Módulos do sistema.

4.4.1 Estruturas Básicas: Visões, Identificadores de Servidores e *Updates*

A implementação faz uso dos seguintes tipos de dados em todo o sistema: visão, identificadores de servidores e *updates*. Esses tipos são altamente correlacionados e juntos servem como tipos básicos para a realização dos protocolos implementados.

Identificadores de Servidores Representa um servidor. Identificamos um servidor pelo seu endereço da rede junto com a sua porta.

```go
type Process struct {
   Addr string
}
```

Update Representa atualizações no grupo dos participantes do sistemas. É composto por um tipo (entrada/saída) e um servidor.

```go
type Update struct {
   Type   updateType
   Process Process
}

type updateType string

const (
   Join  updateType = "+"
   Leave updateType = "-"
)
```

Visão Representa a composição do sistema. Pode ser vista como um conjunto de *updates*.

```go
type View struct {
   Entries map[Update]bool
   Members map[Process]bool // Cache, pode ser reconstruído com Entries
}
```

Duas visões são iguais quando suas *Entries* são iguais. Dizemos que uma visão A é mais atualizada que uma visão B se A contém todas as atualizações que B possui mais algumas outras atualizações. Essas operações foram implementadas com os métodos: *Equal* e *LessUpdatedThan*.

```go
func (v *View) Equal(v2 *View) bool {
   if len(v.Entries) != len(v2.Entries) {
      return false
   }

   for k2, _ := range v2.Entries {
      if _, ok := v.Entries[k2]; !ok {
         return false
      }
   }
   return true
}

func (v *View) LessUpdatedThan(v2 *View) bool {
   if len(v2.Entries) > len(v.Entries) {
      return true
   }
```

```
18
19      for k2, _ := range v2.Entries {
20          if _, ok := v.Entries[k2]; !ok {
21              return true
22          }
23      }
24      return false
25  }
```

Como visões são usadas tanto no registrador quanto nos protocolos de reconfiguração, seu uso é feito sempre através dos seus métodos, que foram intencionalmente projetados de forma a não modificar a visão. Dessa forma o código que usa visões fica mais limpo e não há preocupações com controle de concorrência.

Cada servidor do sistema contém uma visão atual do sistema. Essa visão é muito utilizada para verificar o quão atual são os eventos que o servidor recebe e é modificada apenas pelo módulo de reconfiguração. Essa visão, por sua vez, necessita controle de concorrência e o mesmo é implementado pelo tipo *CurrentView*.

```
1  type CurrentView struct {
2      view *View
3      mu   *sync.RWMutex
4  }
```

A distinção entre os tipos *View* e *CurrentView* é uma separação entre as visões que não precisam de controle de concorrência e as que precisam. Como a maioria dos usos de visões trabalham em cima de uma visão imutável, a implementação foi projetada de forma a trabalhar com o tipo imutável e utilizar um tipo específico para quando é uma visão específica (a visão atual do sistema). Esse tipo específico é composto por uma visão imutável e um *mutex* utilizado no controle ao acesso e à troca da visão.

A última estrutura básica relacionada às visões é o tipo *ViewSeq*. Esse tipo representa sequências de visões (como as geradas pelos geradores de sequência de visões) e apresentam métodos para retornar a visão menos atualizada ou a mais atualizada (utilizadas nos protocolos de reconfiguração).

```
1  type ViewSeq []*View
```

4.4.2 Implementação do Protocolo de Leitura e Escrita

O protocolo de leitura e escrita em um registrador implementado é bem parecido com o descrito na Seção 2.3.3. A única mudança é que as requisições dos clientes agora precisam conter a visão a qual a requisição é destinada, i.e., a visão mais atual conhecida pelo cliente. A razão disso é que o mesmo utiliza uma visão que pode estar desatualizada para enviar cada requisição aos servidores do sistema. O sistema ao receber um pedido com uma visão antiga informa o cliente da nova visão instalada, com a qual o mesmo deve reenviar seu pedido.

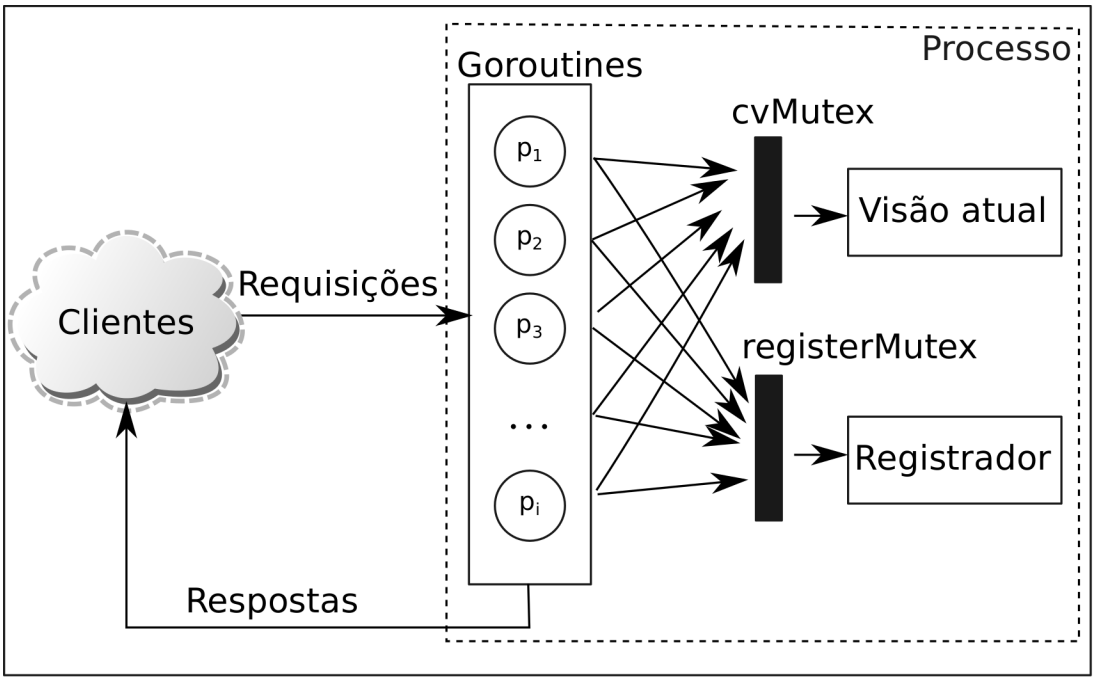


Figura 4.3: Processamento das requisições dos clientes.

Essa parte do sistema implementa a interface utilizada pelo cliente: o processamento dos pedidos de *read* e *write*. O registrador é simplesmente uma variável de tipo genérico. Associado ao registrador temos um *mutex* para realizar o controle de concorrência entre *goroutines* concorrentes. Esse mesmo *mutex* é utilizado pelo protocolo de reconfiguração para suspender as operações de leitura e escrita quando o servidor estiver trocando a sua visão do sistema.

A Figura 4.3 ilustra como é realizado o processamento dos pedidos em um servidor do sistema: quando um servidor do sistema recebe uma dessas requisições, ele inicializa uma nova *goroutine* para atender a requisição. Essa *goroutine* "tranca" o *cvMutex* para acessar a visão atual e caso o pedido seja para esta visão, "tranca" o *registerMutex* para acessar o valor do registrador e retorna uma resposta para o cliente. Caso o pedido não seja para esta visão, retorna a visão para o cliente.

As mensagens transmitidas entre o cliente e o sistema contém a seguinte estrutura de dados:

```
1 type Value struct {
2   Value    interface{} // Valor do registrador
3   Timestamp int // O timestamp do valor do registrador
4   View view.View // A visão do cliente
5   Err  error    // Erro caso houver
6 }
```

Um cliente realiza as operações de leitura e escrita da seguinte forma:

```
1 // Write v to the system's register.  Can be run concurrently.
2 func (cl *Client) Write(v interface{}) error {
```

```
3     readValue, err := cl.readQuorum()
4     if err != nil {
5        // Special case:  diffResultsErr
6        if err == diffResultsErr {
7           // Do nothing - we will write a new value anyway
8        } else {
9           return err
10       }
11    }
12
13    writeMsg := RegisterMsg{}
14    writeMsg.Value = v
15    writeMsg.Timestamp = readValue.Timestamp + 1
16    writeMsg.ViewRef = cl.ViewRef()
17
18    err = cl.writeQuorum(writeMsg)
19    if err != nil {
20       return err
21    }
22
23    return nil
24 }
25
26 // Read executes the quorum read protocol.
27 func (cl *Client) Read() (interface{}, error) {
28    readMsg, err := cl.readQuorum()
29    if err != nil {
30       // Special case:  diffResultsErr
31       if err == diffResultsErr {
32          log.Println("Found divergence: Going to 2nd phase of read protocol")
33          return cl.read2ndPhase(readMsg)
34       } else {
35          return nil, err
36       }
37    }
38
39    return readMsg.Value, nil
40 }
41
42 func (cl *Client) read2ndPhase(readMsg RegisterMsg) (interface{}, error) {
43    err := cl.writeQuorum(readMsg)
44    if err != nil {
45       return nil, err
46    }
47
48    return readMsg.Value, nil
49 }
```

Onde a função *readQuorum* pede o valor atual do registrador de todos os membros

da visão atual e retorna o valor com o maior *timestamp* depois que o mesmo receber a resposta de uma maioria. A função *writeQuorum* por sua vez pede a todos os membros da visão atual para escrever no registrador o valor em *writeMsg*, e retorna com sucesso quando recebe a confirmação de uma maioria. Ambas essas funções tratam o caso em que a visão atual do sistema esteja desatualizada:

```
1  func (thisClient *Client) readQuorum() (RegisterMsg, error) {
2     destinationView, viewRef := thisClient.ViewAndViewRef()
3
4     resultChan := make(chan RegisterMsg, destinationView.NumberOfMembers())
5     go broadcastRead(destinationView, viewRef, resultChan)
6
7     var failedTotal int
8     var resultArray []RegisterMsg
9     var finalValue RegisterMsg
10    for {
11       receivedValue := <-resultChan
12
13       if receivedValue.Err != nil {
14          if oldViewError, ok := receivedValue.Err.(*view.OldViewError); ok {
15             if destinationView.LessUpdatedThan(oldViewError.NewView) {
16                thisClient.updateCurrentView(oldViewError.NewView)
17                return thisClient.readQuorum()
18             }
19          }
20
21          failedTotal++
22       } else {
23          resultArray = append(resultArray, receivedValue)
24
25          if receivedValue.Timestamp > finalValue.Timestamp {
26             finalValue = receivedValue
27          }
28       }
29
30       everyProcessReturned := len(resultArray)+failedTotal ==
             destinationView.NumberOfMembers()
31       systemFailed := everyProcessReturned && failedTotal >
             destinationView.NumberOfToleratedFaults()
32       if systemFailed {
33          if thisClient.couldGetNewView() {
34             return thisClient.readQuorum()
35          } else {
36             return RegisterMsg{}, errors.New("Failed to get read quorun")
37          }
38       }
39
40       if len(resultArray) == destinationView.QuorumSize() {
41          for _, val := range resultArray {
```

```
42            if finalValue.Timestamp != val.Timestamp {
43               return finalValue, diffResultsErr
44            }
45         }
46         return finalValue, nil
47      }
48   }
49}
```

As linhas 4-5 enviam a requisição à todos os servidores da visão. O loop da linha 10 é uma iteração sobre os resultados das requisições enviadas. O caso da visão do cliente estar desatualizada é tratada nas linhas 14-19. Por fim é testado se uma maioria foi obtida. Caso todos os processos tenham retornado ou falhado (linhas 32-38), a função *couldGetNewView()* tenta obter uma nova visão em algum outro lugar (configurável na inicialização do cliente), tratando assim o caso em que todos os processos da visão anterior tenham saído do sistema antes do cliente ficar sabendo da nova visão.

```
1  func (thisClient *Client) writeQuorum(writeMsg RegisterMsg) error {
2     destinationView, viewRef := thisClient.ViewAndViewRef()
3
4     writeMsg.ViewRef = viewRef
5
6     resultChan := make(chan RegisterMsg, destinationView.NumberOfMembers())
7     go broadcastWrite(destinationView, writeMsg, resultChan)
8
9     var successTotal int
10    var failedTotal int
11    for {
12       receivedValue := <-resultChan
13
14       if receivedValue.Err != nil {
15          if oldViewError, ok := receivedValue.Err.(*view.OldViewError); ok {
16             if destinationView.LessUpdatedThan(oldViewError.NewView) {
17                log.Println("View updated during write quorum")
18                thisClient.updateCurrentView(oldViewError.NewView)
19                return thisClient.writeQuorum(writeMsg)
20             }
21             continue
22          }
23
24          failedTotal++
25       } else {
26          successTotal++
27       }
28
29       everyProcessReturned := successTotal+failedTotal ==
             destinationView.NumberOfMembers()
30       systemFailed := everyProcessReturned && failedTotal >
             destinationView.NumberOfToleratedFaults()
```

```
31      if systemFailed {
32         if thisClient.couldGetNewView() {
33            return thisClient.writeQuorum(writeMsg)
34         } else {
35            return errors.New("Failed to get write quorun")
36         }
37      }
38
39      if successTotal == destinationView.QuorumSize() {
40         return nil
41      }
42   }
43 }
```

A estrutura do *writeQuorum* é bem similar a do *readQuorum*. As linhas 6-7 enviam a requisição à todos os servidores da visão. O loop da linha 11 é uma iteração sobre os resultados das requisições enviadas. O caso da visão do cliente estar desatualizada é tratada nas linhas 15-22. Por fim é testado se uma maioria foi obtida. Novamente, caso todos os processos tenham retornado ou falhado (linhas 31-37), a função *couldGetNewView()* trata o caso em que todos os processos da visão anterior tenham saído do sistema.

4.4.3 Implementação do Protocolo de Reconfiguração

O protocolo de reconfiguração descreve a troca de mensagens necessária para realizar uma reconfiguração. Esse protocolo é relacionado com todos os outros protocolos do sistema pois depende de um gerador de sequência de visões, pode alterar o conteúdo do registrador e também modificar a visão atual. O protocolo implementado foi levemente discutido na Seção 4.1 e pode ser encontrado em [2].

Essa parte do sistema implementa as requisições de *join*, *leave* e a troca de mensagens referentes ao progresso do protocolo de reconfiguração em cada servidor. Essas mensagens são:

INSTALL-SEQ É enviada por um servidor que quer instalar uma sequência de visões. Essa mensagem é repassada para todos os servidores para garantir que todos a receberão. O quórum de *INSTALL-SEQ* com a mesma visão associada e a mesma visão a ser instalada significa que a visão menos atualizada da sequência será instalada.

```
1 type InstallSeqMsg struct {
2    InstallView      *view.View // visão que deve ser instalada
3    ViewSeq          ViewSeq // sequência de visões original
4    AssociatedView   *view.View // visão que gerou a sequência
5    Sender           *view.Process // Servidor que enviou a mensagem
6 }
```

STATE-UPDATE É enviada por um servidor para informar qual é o seu estado atual (valor do registrador e *timestamp*) além de *recv*. O quórum de um *STATE-UPDATE* com a mesma visão associada significa que este será o estado inicial da visão sendo instalada.

```
type StateUpdateMsg struct {
   Value          interface{} // Valor do registrador
   Timestamp      int // timestamp do registrador
   Recv           map[view.Update]bool // recv do servidor
   AssociatedView *view.View // visão associada
}
```

VIEW-INSTALLED É enviada por um servidor para informar que instalou uma visão. O quórum de um *VIEW-INSTALLED* com a mesma nova visão libera um servidor que está saindo do sistema para ser encerrado. Isso garante o progresso do sistema.

```
type ViewInstalledMsg struct {
   InstalledView view.View // visão instalada
}
```

Os pedidos de *join* e *leave* são armazenados na variável *recv* do tipo *map*, qualquer acesso à *recv* utiliza um *mutex* como controle de concorrência. A reconfiguração só começa a ser de fato executada quando um *timer* dispara depois de um tempo pré-determinado desde a última reconfiguração, e então uma nova sequência de visões é requisitada. A Figura 4.4 ilustra esse servidor. Cada *goroutine* primeiro "tranca" o *cvMutex* para verificar se a visão do pedido é a atual, se sim, "tranca" o *recvMutex* para adicionar a atualização ao *recv*.

O gerador de sequências de visões é invocado com uma função que recebe a visão associada e a sequência inicial (que normalmente é a visão atual com as atualizações em *recv*). Mais de uma sequência de visões pode ser gerada por uma única invocação do gerador. As sequências geradas são enviadas novamente para o código de reconfiguração por um canal do tipo *newViewSeq*:

```
type newViewSeq struct {
   ViewSeq       ViewSeq // sequência de visões gerada
   AssociatedView view.View // visão associada
}
```

Um diferencial do *FreeStore* é a separação de conceitos entre o registrador R/W e a reconfiguração do sistema. Conforme ilustrado na Figura 4.5, esses protocolos agem de forma independente com exceção do *mutex* do registrador, que é a única parte comum entre os protocolos. Esse *mutex* é acessado em cada leitura e escrita no sistema, enquanto o algoritmo de reconfiguração apenas acessa esse *mutex* no momento da instalação de uma nova visão.

4.4.4 Implementação dos Geradores de Sequência de Visões

Um gerador de sequência de visões captura os requisitos de sincronia necessários entre os servidores para o protocolo de reconfiguração. Esse trabalho implementa dois geradores de sequência de visões. Um faz uso do protocolo de consenso e assim garante que todos os servidores recebam a mesma sequência de visões de seus geradores, enquanto o outro

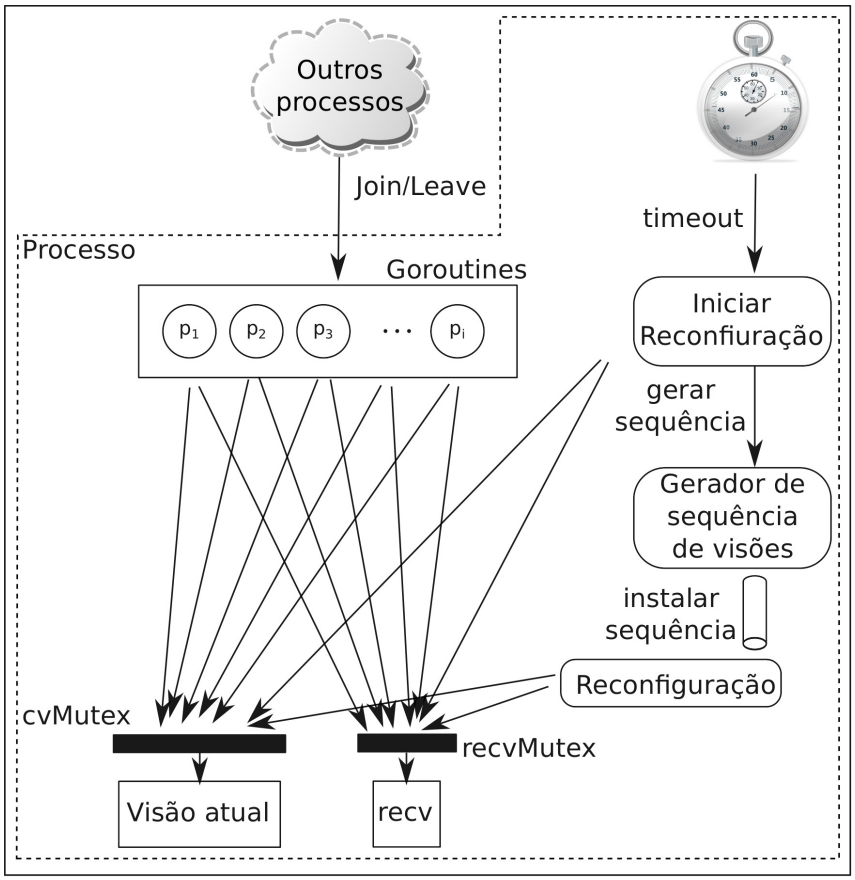


Figura 4.4: Reconfiguração do sistema

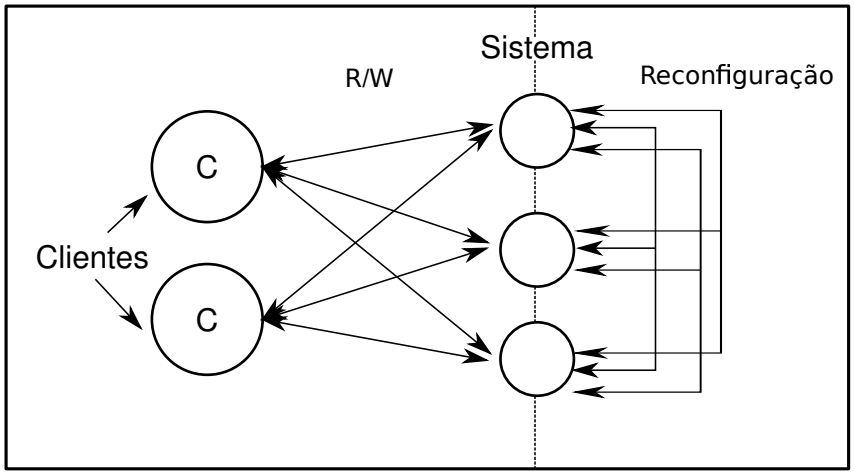


Figura 4.5: Protocolo de reconfiguração é separado do protocolo de leitura e escrita.

não, mas garante a propriedade de que toda sequência gerada pelo gerador contém ou está contida em uma sequência gerada anteriormente, o que é suficiente para garantir as propriedades de segurança e vivacidade do *FreeStore*. A definição formal e os protocolos dos geradores de visões podem ser encontrados em [2].

Gerador de Sequência de Visões com Consenso

Esse gerador utiliza um algoritmo de consenso para alcançar os requisitos de acordo dos geradores de sequência de visões. A sequência de visões gerada nesse gerador tem a característica especial de ser igual em todos os servidores do sistema, pois todos os servidores acabam por decidir a mesma sequência antes de gerá-la (propriedade do consenso).

Nesse trabalho, a solução do consenso implementada foi a do **Paxos**, como descrita em [37]. A implementação considera todos os servidores da visão atual do sistema como *acceptors* e *learners*. A solução é utilizada da seguinte forma: quando o protocolo de reconfiguração requisita uma sequência, um líder é selecionado entre os servidores do sistema para propor uma sequência de visões enquanto todos os outros esperam pela conclusão do protocolo. A sequência de visões proposta pelo líder contém apenas uma visão cujo os membros são definidos pela visão atual mais os pedidos de *join* e *leave* que o líder recebeu desde a última reconfiguração.

Para utilizar a solução do consenso é preciso criar uma instância do consenso. Uma instância possui um identificador único que a diferencia das outras instâncias. O identificador utilizado para gerar uma sequência é igual ao número de atualizações da visão atual do sistema. Como toda visão instalada pelo sistema é mais atualizada do que a anterior, esse número não se repete e pode ser usado como identificador.

Gerador de Sequência de Visões sem Consenso

A captura dos requisitos de sincronia necessários entre os servidores para o protocolo de reconfiguração é realizada nesse gerador através da troca das seguintes mensagens:

SEQ-VIEW É enviada por um servidor para informar a atual sequência de visões sendo proposta ao gerador. O quórum de *SEQ-VIEW* com a mesma visão associada significa que os servidores convergiram para uma sequência de visões, o servidor então envia uma *SEQ-CONV* informando a sequência convergida.

```
type ViewSeqMsg struct {
   AssociatedView view.View // visão associada
   ProposedSeq    ViewSeq // sequência de visões proposta
   LastConvergedSeq ViewSeq // ultima sequência convergida
}
```

SEQ-CONV É enviada por um servidor quando o mesmo ganha o conhecimento de que os servidores convergiram para uma sequência. Um quórum de *SEQ-CONV* com a mesma visão associada significa que aquela sequência foi gerada pelos geradores.

```
type SeqConvMsg struct {
   AssociatedView view.View // visão associada
   Seq ViewSeq // sequência de visões convergida
}
```

Cada servidor, quando requisita uma sequência de visões, cria uma instância do gerador. A instância do gerador possui sua própria *goroutine* e é identificada pela visão associada, essa *goroutine* processa todas as mensagens relativas a sua visão associada e não precisa de nenhum outro controle de concorrência. A primeira *SEQ-VIEW* contém a visão atual do sistema mais as atualizações no *recv*, e é enviada assim que a instância é criada. A troca de mensagens que se sucede constrói em cada servidor uma sequência com visões mais atualizadas que a visão associada.

A Figura 4.6 ilustra o processamento das mensagens utilizadas pelo gerador de sequência de visões sem consenso. Toda requisição é enviada para o canal da visão associada, onde a *goroutine* correspondente realiza o devido processamento. Cada instância do gerador realiza seu próprio processamento e pode gerar mais do que uma sequência de visões, desde que mantenha a propriedade de uma estar contida na outra. De fato, sequências diferentes podem ser geradas em servidores diferentes, mas esta propriedade é mantida.

4.4.5 Inicialização do Sistema e do Cliente

Sistema

Um servidor do sistema, quando inicializado, realiza as seguintes preparações:

- **Inicializa *listener*:** Variável que contém a conexão do servidor com a rede.
- **Inicializa *thisProcess*:** Variável do tipo *Process* que contém o identificador do próprio servidor.
- **Inicializa *useConsensus*:** Variável do tipo *bool* que determina qual gerador de sequências usar.

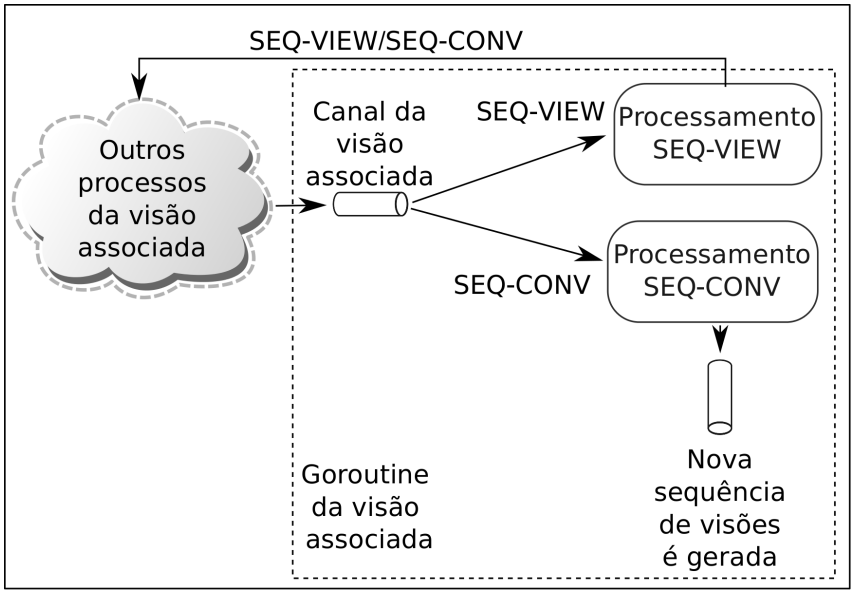


Figura 4.6: Gerador de sequências de visões sem consenso.

- **Inicializa *register*:** Variável que representa o registrador. Essa variável é inicializada para um valor inicial com *timestamp* 0 e seu *mutex* inicializa "trancado" e só é "destrancado" quando o servidor instala uma visão a que pertence, i.e., habilita a execução de operações de *R/W*.
- **Inicializa *currentView*:** Variável do tipo *CurrentView* que contém a atual visão instalada no sistema. Essa visão é utilizada em vários lugares diferentes da implementação para fornecer os atuais membros do sistema.

  A inicialização de *currentView* depende da visão inicial do sistema, que é definida estaticamente na implementação, e é feita da seguinte forma:

  - **Servidor pertence à visão inicial do sistema:** O servidor simplesmente instala a visão inicial do sistema.
  - **Servidor não pertence à visão inicial do sistema:** O servidor requisita a visão atual do sistema a outro servidor do sistema. O endereço desse servidor é passado na sua inicialização.
- **Outros:**
  - Os módulos responsáveis pela escuta de novas conexões são inicializados.
  - As estruturas principais são inicializadas: os canais e os *maps*, por exemplo.
  - As *goroutines* que sempre estão ativas são inicializadas. Elas são do protocolo de reconfiguração:
    - *Goroutine* de processamento de *VIEW-SEQ*.
    - *Goroutine* de processamento de *INSTALL-SEQ*.
    - *Goroutine* de processamento de *STATE-UPDATE*.
    - *Goroutine* que mantém o timer que dispara uma nova reconfiguração.

Cliente

Um cliente, quando inicializado, realiza a inicialização da *currentView* requisitando a um servidor do sistema a visão atual. Após isso, tal servidor está apto a realizar operações de R/W no sistema.

4.5 Práticas de Engenharia de Software Utilizadas

4.5.1 Controle de Versão

Durante toda a etapa de desenvolvimento foi realizado o controle de versão do código com a ferramenta *Git* [26]. Essa prática proporciona a possibilidade de recuar em uma versão antiga de algum componente do sistema. Por exemplo, foram implementadas duas abordagens diferentes no desenvolvimento do protocolo do consenso, e essa prática possibilita uma troca fácil entre estas versões. Outro benefício dessa prática é que o *Git* permite adicionar regras sobre o código a ser versionado, utilizando-se disso, foi imposto que todo código tenha sido formatado corretamente antes de ser versionado.

4.5.2 Detector de Condições de Corrida (*Race Detector*)

A linguagem de programação *Go* disponibiliza com a sua instalação um detector de condições de corrida bastante útil. Essa ferramenta ajuda a detectar escritas e leituras concorrentes, que estão sendo executadas sem controle de concorrência, realizadas por *goroutines* diferentes, e que representam condições de corrida no código. A utilização da ferramenta é bem simples por ser específica para esta linguagem: basta executar o programa com a opção *-race*, não sendo necessário adicionar nenhum código específico dessa ferramenta no código.

Essa ferramenta ajudou na detecção de várias condições de corrida perigosas na implementação desse trabalho: uma nos métodos de uma visão e três no protocolo de reconfiguração. Durante todo o período de desenvolvimento, a implementação foi executada com essa ferramenta.

4.5.3 *Unit Testing*

Foi implementado *unit tests* para os métodos do tipo *View*. Esses testes são executados sempre após uma alteração em algum dos métodos de *View* e novos testes são adicionados quando novas funcionalidades são adicionadas. A abrangência dos testes é limitada ao tipo *View* devido a natureza distribuída dos outros códigos, o que dificulta bastante a criação dos testes. Para conferir a cobertura dos testes criados, foi utilizada a ferramenta *cover*, que indica as partes do código executadas pelos testes criados.

Essa prática ajudou na detecção de um *bug* que foi adicionado inadvertidamente no código. Outro benefício dessa prática é que os testes também fornecem exemplos do funcionamento dos métodos de *View*.

4.6 Outras Implementações

Outras implementações também foram realizadas nesse trabalho, como a de um sistema de quóruns estático e a do sistema de quóruns dinâmico *DynaStore*. Essas implementações foram realizadas como parte do estudo da área e também para serem comparadas nos experimentos desse trabalho.

4.6.1 Sistema de Quóruns Estático

A implementação do sistema de quóruns estático foi baseada no clássico ABD [6] implementado no *FreeStore* e descrito na Seção 4.4.2. Efetivamente, essa implementação é uma simplificação da implementação realizada do *FreeStore*, retirando o código de reconfiguração, geração de sequência de visões e as referências a visões nas mensagens de leitura e escrita.

4.6.2 *DynaStore*

O *DynaStore* [1] foi escolhido por ser o primeiro protocolo de reconfiguração a não precisar do protocolo de consenso, apresentando um funcionamento bem diferente do *FreeStore*. No *DynaStore*, o protocolo de reconfiguração é acoplado na lógica de leitura e escrita do registrador, o que o torna um pouco mais complicado. Devido a isso será apresentado apenas um breve resumo do protocolo, para mais detalhes ver o artigo original [1].

No *DynaStore* não existe a separação de cliente e servidor como no *FreeStore*. Cada servidor contém a lógica de leitura e escrita do sistema. Em cada fase de leitura ou escrita, o servidor sempre realiza a verificação se existe alguma atualização nova para a visão atual do sistema. Para garantir a consistência sequencial em meio a reconfigurações, o protocolo define a seguinte ordem em relação as fases de leitura e escrita e a verificação por novas atualizações:

- Na fase da leitura, a verificação por atualizações da visão deve ser realizada **antes** da leitura de um quórum.
- Na fase da escrita, a verificação por atualizações da visão deve ser realizada **após** a escrita em um quórum.

Essa ordem estabelecida acima mantém a consistência sequencial (toda operação de leitura retorna o valor da última operação de escrita realizada) porque as operações são reiniciadas quando uma visão muda durante a operação. Como as operações verificam por atualizações antes da fase de leitura e depois da fase de escrita, que é o início e o fim de cada operação, nenhuma reconfiguração altera a semântica do registrador.

O grande problema que torna o *DynaStore* imprático, é que a verificação por atualizações por si só é uma operação em uma memória compartilhada. Como essa verificação é realizada no mínimo duas vezes por operação de leitura e escrita, o desempenho é bem menor do que o do *FreeStore*, como será demonstrado no capítulo seguinte. Uma comparação teórica mais aprofundada entre o *DynaStore* e o *FreeStore* pode ser encontrada em [2]

Capítulo 5

Resultados

Após a conclusão da implementação do *FreeStore*, foram realizados experimentos visando analisar o desempenho do sistema. Esse capítulo apresenta o resultado desses experimentos. Na Seção 5.1 descrevemos quais, como e onde os experimentos foram realizados. Os resultados da implementação do *FreeStore* se encontram na Seção 5.2, e em seguida, comparamos esses resultados com os de outros sistemas de quóruns. Por último, apresentamos o comportamento do sistema na presença de falhas e reconfigurações na Seção 5.4.

5.1 Experimentos Realizados

Os experimentos focaram na medição de duas métricas: a latência e o *throughput*. A latência é o tempo do início ao término de uma operação requisitada pelo cliente. No caso de uma memória compartilhada distribuída tolerante a falhas, a latência é o tempo que demora para o cliente receber a confirmação de conclusão de uma operação de leitura ou de escrita. O *throughput* é o número de operações que um sistema realiza por segundo. Nos experimentos realizados, foi realizado a medição tanto da latência quanto do *throughput* de operações de leitura separado das operações de escrita.

Os experimentos realizados foram: (1) alguns *micro-benchmarks* para avaliar a latência e o *throughput* da implementação do *FreeStore* para diferentes operações, tamanhos de dados e quantidades de servidores; (2) comparação entre o modelo estático e os modelos dinâmicos (representado pelo *FreeStore* e o *DynaStore*); e (3) um experimento que mostra o comportamento do *throughput* do sistema *FreeStore* na presença de falhas e reconfigurações.

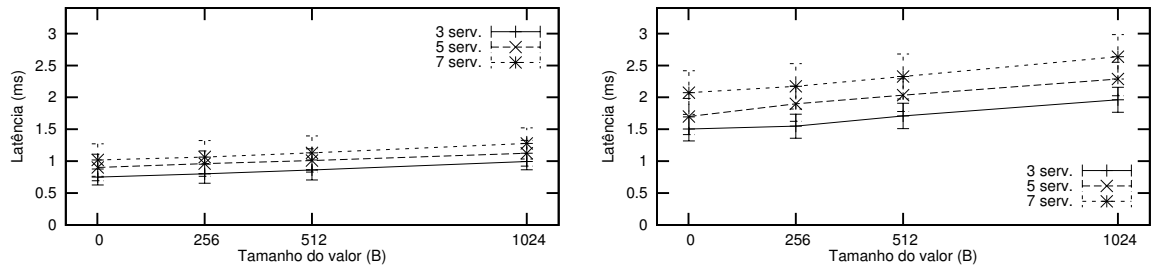
No primeiro e segundo experimento, tanto a latência quanto o *throughput* foram medidos nos clientes, sendo que o *throughput* foi calculado somando-se quantas operações o conjunto de clientes conseguiu executar dividido pelo intervalo de tempo. As latências apresentadas a seguir são dadas pelo tempo médio obtido a partir de 100000 execuções da operação e excluindo-se 5% dos valores com maior desvio. No terceiro experimento, o *throughput* do sistema como um todo foi medido a partir do *throughput* de cada servidor (escolhendo-se o valor maior entre os servidores) e o conhecimento da visão do sistema em cada momento.

Os experimentos foram executados no Emulab[1] [49], em um ambiente consistindo por até 20 máquinas *pc3000* (3.0 GHz 64-*bit* Pentium Xeon com 2GB de RAM e interface de rede gigabit) conectadas a um *switch* de 100Mb. Cada servidor executou em uma máquina separada, enquanto que até 26 clientes foram distribuídos uniformemente nas máquinas restantes (2 por máquina). O ambiente de *software* utilizado foi o sistema operacional Fedora 15 64-*bit* com *kernel* 2.6.20 e compilador *Go* 1.2.

5.2 Resultados do *FreeStore*

O primeiro experimento foi a análise de uma série de *micro-benchmarks* utilizados para avaliar a implementação do *FreeStore*, os quais consistem na leitura e escrita de valores com diferentes tamanhos (i.e. 0, 256, 512 e 1024 *bytes*) e para diferentes números de servidores.

As figuras 5.1a e 5.1b mostram a latência da leitura e escrita, respectivamente, para o sistema configurado com 3, 5 e 7 servidores (i.e., tolerando 1, 2 ou 3 falhas), enquanto que as figuras 5.2a e 5.2b apresentam os valores de *throughput* para as mesmas configurações. Já as figuras 5.3a e 5.3b apresentam a relação entre o *throughput* e a latência de leitura e escrita, respectivamente, para o sistema configurado com 3 servidores. Na latência por throughput, cada ponto foi calculado utilizando uma quantidade diferente de clientes $(1, 2, 4, ..., 16)$ e somando-se quantas operações os clientes conseguiram executar dividido pelo intervalo de tempo.



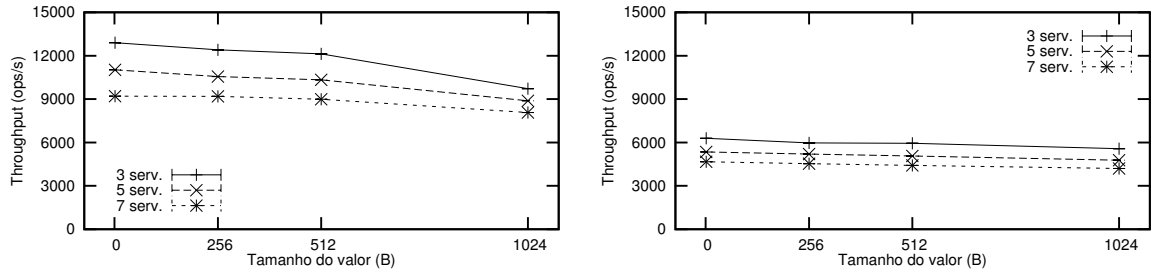
(a) Operações de leitura.

(b) Operações de escrita.

Figura 5.1: Latência do *FreeStore*.

Neste primeiro conjunto de resultados, podemos perceber que tanto o aumento no número de servidores quanto no tamanho do valor a ser lido/escrito faz com que o desempenho do sistema diminua. Este comportamento era esperado e deve-se ao fato de que com mais servidores um maior número de mensagens são enviadas e esperadas para atingir um quórum, enquanto que o aumento no tamanho do valor faz com que estas mensagens também tenham um tamanho maior. Finalmente, podemos perceber que os custos de uma leitura são praticamente a metade dos custos de uma escrita. Isso deve-se ao protocolo de leitura e escrita empregado [6], que requer dois passos de comunicação para leituras e quatro passos para escritas, considerando execuções sem concorrência entre leituras e escritas.

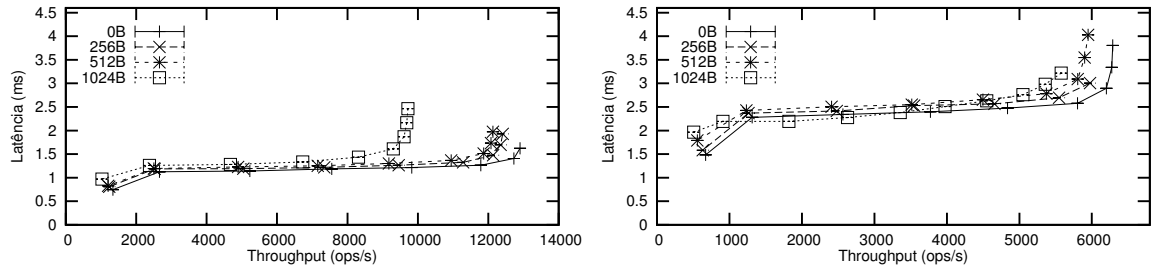
[1]Emulab é uma instalação de computadores disponível, sem custos, a pesquisadores da área de redes de computadores e sistemas distribuídos.



(a) Apenas operações de leitura.

(b) Apenas operações de escrita.

Figura 5.2: *Throughput* do *FreeStore*.



(a) Leitura com ($n = 3$ servidores).

(b) Escrita com ($n = 3$ servidores).

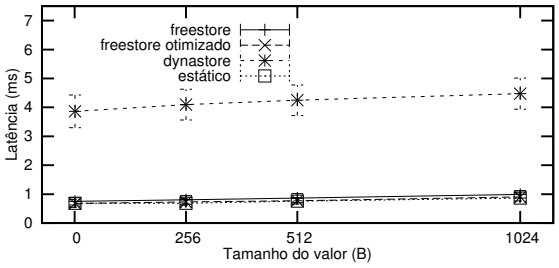
Figura 5.3: Latência por *throughput* do *FreeStore*.

5.3 Comparação entre o Modelo Estático e os Modelos Dinâmicos

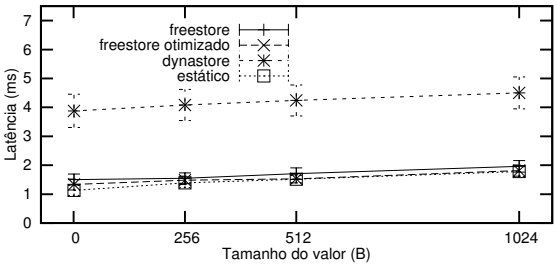
Este segundo experimento visa comparar o desempenho do *FreeStore* com a sua versão para ambientes estáticos (onde o conjunto de servidores é fixo) e com o *DynaStore*. Basicamente, os custos adicionais no *FreeStore* em relação à versão estática é o envio da visão atual conhecida pelo cliente em cada mensagem e a verificação se a mesma é a visão mais atualizada do servidor. Reportamos também os resultados para uma implementação otimizada do *FreeStore*, onde apenas um *hash* criptográfico (*SHA-1*) das visões são anexados nas mensagens, diminuindo o tamanho das mesmas. Uma visão de 3, 5 e 7 processos, quando codificada para ser anexada em uma mensagem, tem tamanho de 390, 469 e 548 bytes, respectivamente, o que é bem maior que um *hash* de 20 *bytes* como os do *SHA-1*.

As figuras 5.4a e 5.4b apresentam os valores para a latência de leitura e escrita, respectivamente, para o sistema configurado com 3 servidores, enquanto que as figuras 5.5a e 5.5b apresentam os valores de *throughput*. Já as figuras 5.6a e 5.6b apresentam a relação entre o *throughput* e a latência de leitura e escrita, respectivamente, considerando um valor de 512 *bytes* e calculando as medidas da mesma forma que no primeiro experimento.

Nestes experimentos podemos perceber que a versão otimizada do *FreeStore* apresenta um desempenho praticamente igual a versão estática. Conforme já comentado, a modularidade do *FreeStore* faz com que os protocolos de reconfiguração praticamente não

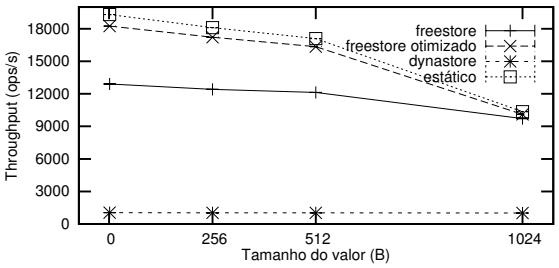


(a) Operações de leitura.

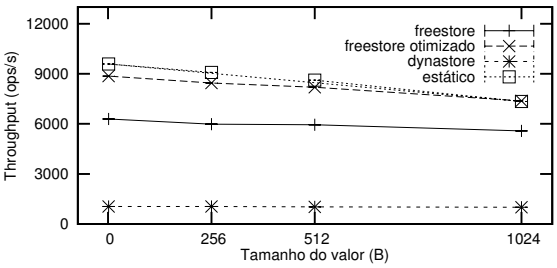


(b) Operações de escrita.

Figura 5.4: Dinâmico vs. Estático: Latência ($n = 3$ servidores).

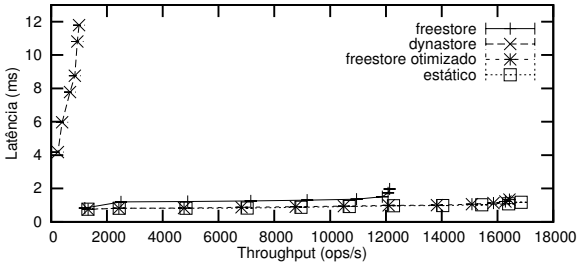


(a) Apenas operações de leitura.

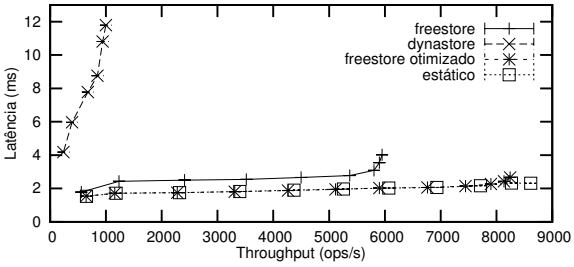


(b) Apenas operações de escrita.

Figura 5.5: Dinâmico vs. Estático: *Throughput* ($n = 3$ servidores).



(a) Leitura (valor de 512 *bytes*).



(b) Escrita (valor de 512 *bytes*).

Figura 5.6: Dinâmico vs. Estático: Latência por *Throughput* ($n = 3$ servidores).

interfiram nos protocolos de leitura e escrita, quando executados em períodos onde não estão ocorrendo reconfigurações. Pelos mesmos motivos anteriormente descritos, as escritas apresentaram praticamente a metade do desempenho das leituras.

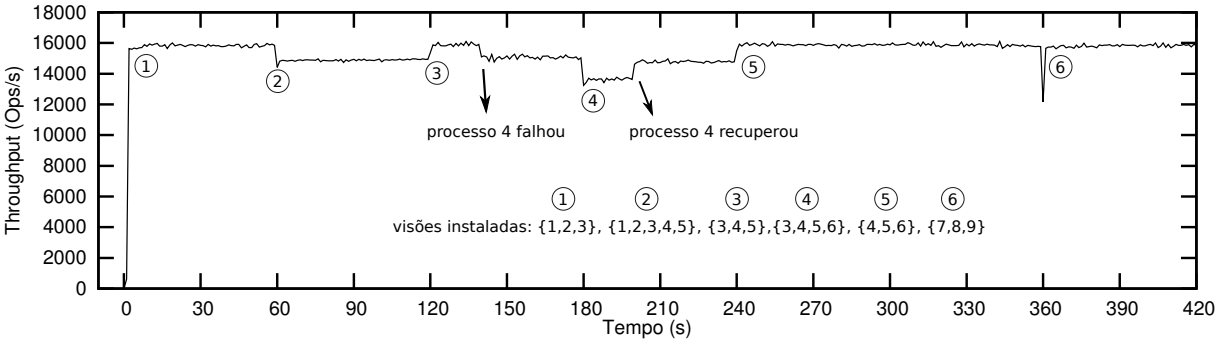
Já o *DynaStore*, por apresentar um grande acoplamento entre as operações de leitura e escrita e reconfigurações, apresentou um desempenho bem inferior.

5.4 Comportamento com Falhas e Reconfigurações

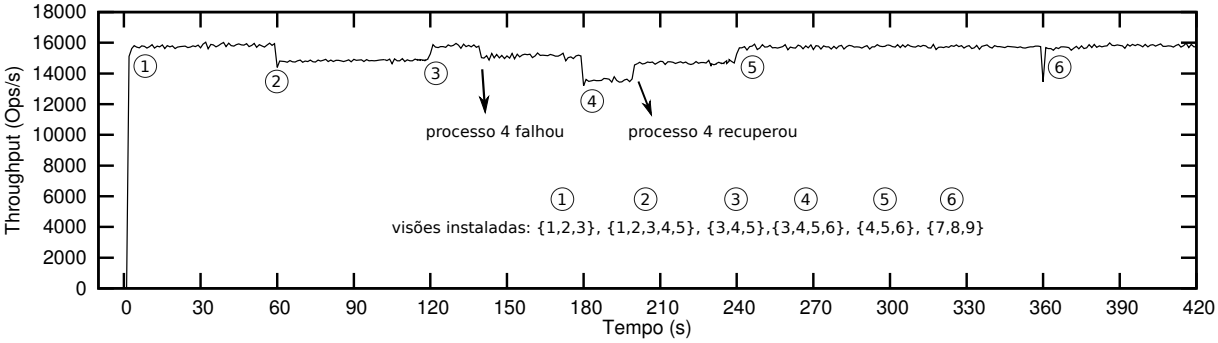
Este último experimento visa analisar o comportamento do *FreeStore* em funcionamento mediante reconfigurações e falhas de processos. Para isso, o sistema foi inicialmente configurado com 3 servidores (visão inicial = {1, 2, 3}) e os clientes (nesse experimento utilizamos 18 clientes) executaram operações de leitura de um valor com tamanho de 512 *bytes*. O período entre reconfigurações foi configurado em 60 segundos, i.e., a cada 60 segundos uma reconfiguração é executada.

As figuras 5.7a e 5.7b mostram o *throughput* apresentado pelo sistema durante a sua execução utilizando ou não um protocolo de consenso para reconfigurações (i.e., utilizando cada um dos dois geradores discutidos na Seção 4.1), respectivamente. Os eventos que ocorreram durante a execução foram os seguintes:

- 0s - início do experimento.
- 30s - o servidor 4 envia um pedido de *join* e aguarda sua entrada no sistema.
- 40s - o servidor 5 envia um pedido de *join* e aguarda sua entrada no sistema.
- 60s - uma reconfiguração adiciona os servidores 4 e 5 no sistema, instalando a visão {1,2,3,4,5}.
- 80s - o servidor 1 envia um pedido de *leave* e aguarda sua saída do sistema.
- 100s - o servidor 2 envia um pedido de *leave* e aguarda sua saída do sistema.
- 120s - uma reconfiguração remove os servidores 1 e 2 do sistema, instalando a visão {3,4,5}.
- 140s - o servidor 4 sofre um *crash* (falha).
- 160s - o servidor 6 envia um pedido de *join* e aguarda sua entrada no sistema.
- 180s - uma reconfiguração adiciona o servidor 6 no sistema, instalando a visão {3,4,5,6}.
- 200s - o servidor 4 se recupera e volta ao sistema.
- 220s - o servidor 3 envia um pedido de *leave* e aguarda sua saída do sistema.
- 240s - uma reconfiguração remove o servidor 3 do sistema, instalando a visão {4,5,6}.
- 300s - o sistema tenta executar uma reconfiguração, mas como não havia nenhum pedido, nada é processado.
- 320s - os servidores 4,5 e 6 enviam um pedido de *leave* e aguardam pelas suas saídas do sistema.



(a) Com consenso.



(b) Sem consenso.

Figura 5.7: Reconfigurações e falhas.

- 340s - os servidores 7,8 e 9 enviam um pedido de *join* e aguardam pelas suas entradas no sistema.
- 360s - uma reconfiguração removes os servidores 4,5 e 6 e adiciona os servidores 7,8 e 9 no sistema, instalando a visão {7,8,9}.
- 420s - fim do experimento.

Neste experimento podemos perceber que o desempenho do sistema diminui quando mais servidores estiverem presentes no mesmo. Além disso, devido às características de desacoplamento e modularidade do *FreeStore*, o desempenho é praticamente o mesmo para ambas as abordagens de reconfiguração (com ou sem consenso). De fato, o tempo médio de uma reconfiguração foi de 19ms na abordagem sem consenso (como os pedidos de reconfiguração foram recebidos por todos os processos antes do início da reconfiguração, todos tinham a mesma proposta e o protocolo do gerador converge com apenas um passo – *SEQ-VIEW*, além disso este tempo também engloba os passos do algoritmo de reconfiguração), bloqueando as operações de R/W por apenas 4ms. Para a solução com consenso, o tempo médio de uma reconfiguração foi de 40ms (tempo para executar o protocolo de consenso e os passos de algoritmo de reconfiguração), bloqueando as operações de R/W por 4ms. Note que o tempo em que as operações de R/W ficaram bloqueadas foi o mesmo em ambas as abordagens, pois este bloqueio somente ocorre depois de uma sequência ser gerada (i.e., após a execução do gerador).

Reconfiguração nos piores casos

O caso acima descrito representa o melhor caso, quando todos recebem os mesmos pedidos de reconfiguração antes de iniciar a reconfiguração. Porém, nos casos em que os pedidos de reconfiguração foram recebidos de forma diferente pelos processos antes do início da reconfiguração, o tempo de reconfiguração varia de caso para caso. Para medir o tempo de reconfiguração para quando isso acontece, executamos 5 vezes o caso em que 3 processos {1,2,3} vão realizar a reconfiguração para adicionar mais 3 processos ao sistema. Para garantir que cada processo faça uma proposta diferente, descartamos alguns pedidos de *join* nos processos. Este experimento só faz sentido para a solução que não utiliza consenso, visto que o consenso resolve qualquer conflito de propostas com os mesmos passos de comunicação.

Na primeira execução, as visões propostas no inicio do gerador sem consenso pelos 3 processos foram {4,5,6}, {4,6} e {6}, onde 4,5,6 indicam os processos que querem entrar no sistema. O tempo total de reconfiguração, que engloba os passos do gerador e os de reconfiguração, foi de 8.01s, bloqueando as operações de R/W por 6s. Durante essa reconfiguração, foram geradas 4 sequências de visões que resultou na instalação de 3 visões: {1,2,3,6} -> {1,2,3,4,6} -> {1,2,3,4,5,6} (2 visões intermediárias). O tempo em que as operações de R/W ficaram bloqueadas foi alto devido ao fato de que elas somente são desbloqueadas quando a visão final for instalada, isto é necessário para garantir a segurança da reconfiguração [2].

Na segunda execução, as visões propostas foram {4,5,6}, {4,5} e {4}. O tempo total de reconfiguração foi de 6.01s, bloqueando as operações de R/W por 4s. Durante essa reconfiguração, foram geradas 3 sequências de visões que resultou na instalação de 2 visões: {1,2,3,4} -> {1,2,3,4,5,6} (apenas 1 visão intermediária). O tempo em que as operações de R/W ficaram bloqueadas foi menor nesse caso em relação ao anterior pois a reconfiguração utilizou apenas uma visão intermediária, ao invés de duas.

Na terceira execução, as visões propostas foram {4,5}, {4,6} e {6}. Essa reconfiguração foi divida entre duas reconfigurações. Onde a primeira durou 6.01s, bloqueando as operações de R/W por 4s, e levou o sistema para a visão {1,2,3,4,6}. A segunda reconfiguração, por sua vez, durou 19ms, bloqueando as operações de R/W por 4.34ms, e levou o sistema para a visão final {1,2,3,4,5,6}. A reconfiguração foi dividida em duas pois os geradores dos processos geraram a sequência {1,2,3,6} -> {1,2,3,4,6}, que quando instalada deixou o sistema com a visão final {1,2,3,4,6}. Em seguida a segunda reconfiguração adicionou o processo 5 ao sistema tendo um comportamento como no melhor caso (todos os processos propuseram a mesma visão).

Na quarta execução, as visões propostas foram {4,5,6}, {4} e {} (nenhuma proposta). O tempo total de reconfiguração foi de 2.01s, bloqueando as operações de R/W por 5ms. Durante essa reconfiguração, foram geradas 2 sequências de visões que resultou na instalação direta da visão final {1,2,3,4,5,6}.

Na quinta execução, as visões propostas foram {4,5,6}, {4,5,6} e {5}. O tempo total de reconfiguração foi de 2.01s, bloqueando as operações de R/W por 4.67ms. Durante essa reconfiguração, foram geradas 2 sequências de visões que resultou na instalação direta da visão final {1,2,3,4,5,6}.

As reconfigurações nos piores casos descritas acima, apesar de possíveis, são bem improváveis de acontecer. Acredita-se que é possível aprimorar a lógica de como um processo pede para entrar no sistema de forma a reduzir ainda mais a chance dos processos

possuírem diferentes propostas no momento de uma reconfiguração, mas em geral, esse raro risco de uma demora maior na reconfiguração devido a certas combinações de falhas é o preço que se paga pelo modelo assíncrono.

Capítulo 6

Conclusões

O presente capítulo conclui este trabalho. Primeiramente, uma visão geral sobre o trabalho é apresentada, em seguida algumas perspectivas de trabalhos futuros são expostas.

6.1 Visão Geral do Trabalho

Este trabalho discutiu uma implementação para os protocolos do *FreeStore* e através de um conjunto de experimentos possibilitou uma maior compreensão sobre o comportamento destes protocolos e do modelo dinâmico de memória compartilhada como um todo. Os experimentos também mostraram uma diferença mínima no desempenho entre o *FreeStore* e uma memória compartilhada estática, permitindo os benefícios de flexibilidade no sistema com um custo mínimo. A comparação entre *FreeStore* e *DynaStore* evidenciou a grande diferença de desempenho entre os dois protocolos e a importância do desacoplamento entre os protocolos de leitura e escrita e os de reconfiguração.

Os objetivos gerais desse trabalho, apresentados na Seção 1.2 foram alcançados com sucesso:

- O projeto da implementação do *FreeStore* foi apresentada na Seção 4.3.
- A implementação em si foi descrita na Seção 4.4.
- Sistemas comparados na Seção 5.3 foram apresentados na Seção 4.6.
- A análise dos resultados foram apresentados no Capítulo 5.

Esse trabalho também gerou o artigo "Proposta de Implementação de Memória Compartilhada Dinâmica Tolerante a Falhas", que recebeu o prêmio de melhor artigo no 15º Workshop de Testes e Tolerância a Falhas. O artigo apresentou a implementação e parte dos resultados obtidos durante esse trabalho [13].

Todas as implementações realizadas nesse trabalho estão disponível em:

- `http://www.github.com/mateusbraga/freestore`
- `http://www.github.com/mateusbraga/static-quorum-system`
- `http://www.github.com/mateusbraga/dynastore`

O código possui a licença de código aberto MIT [40], que permite qualquer um fazer o que quiser com o código desde que não tirem os créditos dos autores e não os responsabilizem por qualquer consequência.

6.2 Trabalhos Futuros

Como o objetivo desse trabalho foi estudar, implementar e analisar os protocolos do *FreeStore*, alguns aspectos da memória compartilhada em si teve menor atenção. Por exemplo, a estrutura da memória sendo compartilhada na implementação foi apenas um registrador. Uma possibilidade para trabalho futuro seria implementar a abstração de um sistema de arquivos e diretórios, o que acrescentaria maior flexibilidade e utilidade aos usuários do sistema. Outro aspecto que poderia receber mais atenção é a performance do sistema. Otimizações como evitar ao máximo a alocação de estruturas durante a comunicação do sistema pode reduzir consideravelmente o trabalho do coletor de lixo e assim melhorar o *throughput* do sistema.

Atualmente, a recuperação de um processo que foi reiniciado é realizada na próxima operação requisitada por algum cliente (na segunda fase da leitura ou em uma escrita). Uma possível melhoria seria fazer o processo reiniciado executar uma operação de leitura como se fosse um cliente, reduzindo assim o tempo médio de recuperação.

A implementação desse trabalho assumiu algumas premissas não essenciais e que podem ser eliminadas no futuro. Por exemplo, o uso de criptografia na comunicação entre clientes e servidores pode eliminar a premissa de que não há falhas maliciosas no canal de comunicação. Outra premissa relacionada ao uso do protocolo de consenso no gerador de sequência de visões correspondente, é que o servidor líder não falha. Essa premissa também pode ser eliminada com a utilização de um protocolo de eleição de líder. Por fim, é possível eliminar a restrição de que um processo só pode sair e entrar do sistema uma única vez modificando as atualizações de uma visão para conter um *nonce* que diferencia os pedidos de entrada e saída do mesmo processo [2].

Referências

[1] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *JACM*, 58:7:1–7:32, 2011. 2, 46

[2] Eduardo Alchieri, Alysson Bessani, Fabíola Greve, and Joni Fraga. Reconfiguração modular de sistemas de quóruns. In *Anais do 32º Simpósio Brasileiro de Redes de Computadores*, 2014. 2, 24, 25, 30, 39, 42, 46, 53, 56

[3] Eduardo A. P. Alchieri. *Protocolos Tolerantes a Falta Bizantinas para Sistemas Distribuídos Dinâmicos*. Tese de doutorado em Engenharia de Automação e Sistemas, Universidade Federal de Santa Catarina, Florianópolis, 2011. 1

[4] Apache Zookeeper. http://zookeeper.apache.org, acessado em 29/09/2013. 18, 22

[5] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003. 22

[6] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995. 46, 48

[7] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley Series on Parallel and Distributed Computing. Wiley-Interscience, 2nd edition, 2004. 22

[8] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, March 2004. vii, 4, 5, 6, 9, 10

[9] Ziv Bar-Joseph, Idit Keidar, and Nancy Lynch. Early-delivery dynamic atomic broadcast. In *Distributed Computing*, pages 1–16. Springer, 2002. 23

[10] Alysson N. Bessani, Eduardo A. P. Alchieri, and Paulo Souza. Bft-smart: High-performance byzantine-fault-tolerant state machine replication. http://code.google.com/p/bft-smart/, 2011. 20

[11] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993. 20

[12] Barry W. Boehm. Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1(1):75–88, January 1984. 10

[13] Mateus Braga and Eduardo Alchieri. Proposta de implementação de memória compartilhada dinâmica tolerante a falhas. In *Anais do 15º Workshop de Testes e Tolerância a Falhas*, 2014. 55

[14] Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996. 21, 23

[15] Miguel Correia, Nuno F. Neves, and Paulo Veríssimo. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96, 2006. 23

[16] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design (5th Edition)*. Addison Wesley, 5 edition, May 2011. 15, 16, 19

[17] George F Coulouris. *Distributed Systems: Concepts and Design, 4/e*. Pearson Education India, 2009. 18, 19

[18] Flaviu Cristian. Agreeing on who is present and who is absent in a synchronous distributed system. In *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on*, pages 206–211. IEEE, 1988. 21

[19] Flaviu Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–187, April 1991. 21

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, volume 7, pages 205–220, 2007. 5

[21] D. Djenouri, L. Khelladi, and A.N. Badache. A Survey of Security Issues in Mobile Ad Hoc and Sensor Networks. *Communications Surveys & Tutorials, IEEE*, 7(4):2–28, 2005. 1

[22] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985. 22

[23] Ian Foster. What is the grid? a three point checklist. *Grid Today*, 1(6):22–25, 2002. 1

[24] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Comput.*, 31(1):48–59, January 1982. 30

[25] David Gifford. Weighted voting for replicated data. In *Proc. of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, December 1979. 2

[26] Git. http://git-scm.com/, acessado em 14/11/2013. 45

[27] P. Brighten Godfrey, Scott Shenker, and Ion Stoica. Minimizing churn in distributed systems. *SIGCOMM Comput. Commun. Rev.*, 36(4):147–158, 2006. 17

[28] The Go Programming Language. http://golang.org, acessado em 05/10/2013. 25

[29] R. Guerraoui and L. Rodrigues. *Lecture Notes in Distributed Computing (Preliminary Draft)*. Springer-Verlag, Berlin, 2003. 21

[30] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997. 11, 12

[31] Vassos Hadzilacos and Sam Toueg. A modular approach to the specification and implementation of fault-tolerant broadcasts. *Dep. of Computer Science, Cornell Univ., New York, USA, Tech. Rep*, pages 94–1425, 1994. 22, 23

[32] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010. 18, 22

[33] Geoff Huston. Anatomy: A look inside network address translators. *The Internet Protocol Journal*, 7(3):2–32, 2004. 31

[34] Mohammad Ilyas. *The handbook of ad hoc wireless networks*, volume 29. CRC press, 2010. 1

[35] Idit Keidar, Jeremy Sussman, Keith Marzullo, and Danny Dolev. Moshe: A group membership service for wans. *ACM Trans. Comput. Syst.*, 20:191–238, 2002. 22

[36] Leslie Lamport. On interprocess communication (part II). *Distributed Computing*, 1(1):203–213, January 1986. 25

[37] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001. 24, 42

[38] Nancy Lynch and Alex A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *16th International Symposium on Distributed Computing - DISC*, pages 173–190, 2002. 2

[39] Jean-Philippe Martin and Lorenzo Alvisi. A framework for dynamic byzantine storage. In *Dependable Systems and Networks, 2004 International Conference on*, pages 325–334. IEEE, 2004. 2

[40] The MIT License (MIT). http://opensource.org/licenses/MIT, acessado em 05/06/2014. 56

[41] NSQ. http://bitly.github.io/nsq/, acessado em 19/11/2013. 27

[42] David Powell. Group communication. *Communications of the ACM*, 39(4):50–53, 1996. 20

[43] Pedro A. D. Rezende. Modelos de Confiança para Segurança em Informática. http://www.cic.unb.br/docentes/pedro/trabs/modelos_de_confianca.pdf, acessado em 29/09/2013. 31

[44] Aleta M. Ricciardi and Kenneth P. Birman. Using process groups to implement failure detection in asynchronous environments. In *ACM Symposium on Principles of Distributed Computing*, pages 341–353, Montreal - Quebec - Canada, 1991. 21

[45] Andre Schiper and Alain Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *Distributed Computing Systems, 1993., Proceedings the 13th International Conference on*, pages 561–568. IEEE, 1993. 12

[46] Ralf Steinmetz and Klaus Wehrle. *Peer-to-Peer Systems and Applications*. Springer-Verlag, 2005. 1

[47] Scott D. Stoller. Leader election in asynchronous distributed systems. *IEEE Trans. Comput.*, 49(3):283–284, March 2000. 30

[48] Daniel Stutzbach and Reza Rejaie. Understanding Churn in Peer-to-Peer Networks. In *IMC '06: Proceedings of the 6th ACM SIGCOMM on Internet measurement*, pages 189–202, New York, NY, USA, 2006. ACM Press. 17

[49] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of 5th Symposium on Operating Systems Design and Implementations*, 2002. 48

[50] Feng Zhao and Leonidas Guibas. *Wireless Sensor Networks: An Information Processing Approach*. Morgan Kaufmann, 2004. 1