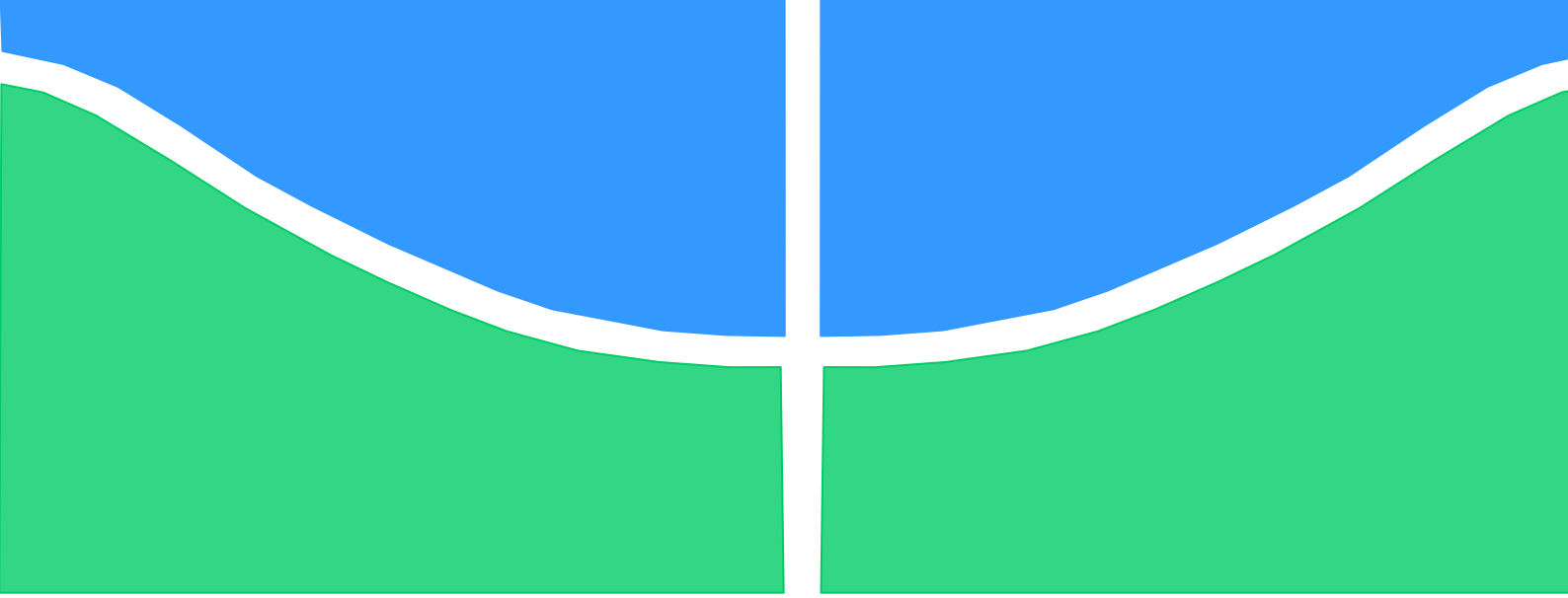

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Curso de Engenharia de Software

ESTRUTURA DISTRIBUÍDA PARA O PADRÃO DE PROGRAMAÇÃO EM N VERSÕES

Autor: Wagner Jerônimo Santos
Orientador: Dr. Luiz Augusto Fontes Laranjeira

Brasília, DF
2013

WAGNER JERÔNIMO SANTOS

ESTRUTURA DISTRIBUÍDA PARA O PADRÃO DE PROGRAMAÇÃO EM N VERSÕES

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Orientador: Dr. Luiz Augusto Fontes Laranjeira

Brasília, DF
2013

CIP – Catalogação Internacional da Publicação*

Santos, Wagner Jerônimo.
Estrutura Distribuída para o Padrão de Programação em N Versões / Wagner Jerônimo Santos. Brasília: UnB, 2013. 103 p. : il. ; 29,5 cm.

Monografia (Graduação) – Universidade de Brasília
Faculdade do Gama, Brasília, 2013. Orientação: Dr. Luiz Augusto Fontes Laranjeira.

1. Programação em N Versões. 2. Tolerância à faltas. 3. Confiabilidade 4. Sistema Distribuído 5. Padrão de projeto
I. Laranjeira, Luiz Augusto Fontes. II. Dr.

CDU Classificação

ESTRUTURA DISTRIBUÍDA PARA O PADRÃO DE PROGRAMAÇÃO EM N VERSÕES

Wagner Jerônimo Santos

Monografia submetida como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software da Faculdade UnB Gama - FGA, da Universidade de Brasília, em 10/12/2013 apresentada e aprovada pela banca examinadora abaixo assinada:

Prof. Dr.: Luiz Augusto Fontes Laranjeira, UnB/ FGA
Orientador

Prof. Dr.: André Barros de Sales, UnB/ FGA
Membro Convidado

Prof. Dra.: Carla Rocha Aguiar, UnB/ FGA
Membro Convidado

Brasília, DF
2013

Esse trabalho é dedicado à Deus, minha família e amigos.

AGRADECIMENTOS

Agradecer primeiramente a Deus, por me iluminar e abençoar minha trajetória.

A minha mãe, Maria Amélia, pelo apoio e por tudo que sempre fez por mim, pela simplicidade, exemplo, amizade, amor e carinho, fundamentais na construção do meu caráter.

Aos meus familiares que sempre me acolheram e estiveram presentes na minha vida me fazendo uma pessoa melhor.

Aos meus amigos que sempre estiveram ao meu lado e sempre pude contar em todos os momentos.

Ao Professor Doutor Luiz Laranjeira, que foi de suma importância para a realização desse trabalho, pela paciência e dedicação a mim prestadas. Muito obrigado!

A todos que de alguma forma ajudaram, agradeço por acreditarem no meu potencial, principalmente nos momentos difíceis, quando nem eu mais acreditava.

You may be disappointed if you fail, but you are doomed if you don't try. - Beverly Sills

RESUMO

Como a sociedade moderna tem-se tornado cada vez mais dependente de computadores e redes de comunicação, a vulnerabilidade e eventual paralisação destes sistemas poderá afetar significativamente a vida diária das pessoas, tanto do ponto de vista social quanto do ponto de vista econômico. Sendo assim, sistemas computacionais com características de alta confiabilidade são cada vez mais necessários. A implementação de sistemas confiáveis é dispendiosa, pois implica na utilização de técnicas baseadas na exploração de redundância. Tais técnicas usualmente requerem, para cada componente do sistema, a inclusão de várias instâncias do mesmo, com funcionalidades equivalentes, a fim de possibilitar sua recuperação ou substituição em caso de faltas. Para atender a esses critérios de confiabilidade, Padrões de Projeto (*Design Patterns*) podem ser utilizados no desenho (design) de sistemas de software e hardware. O Padrão de Programação em N Versões é utilizado para tolerar faltas de *software* e, se implementado em uma arquitetura distribuída, pode também tolerar faltas de *hardware*. Nesse trabalho é proposta uma estrutura distribuída para a implementação do Padrão de Programação de N Versões que elimina pontos únicos de falha tanto de software como de hardware. Esta estrutura é implementada de forma genérica, com a finalidade de diminuir o esforço requerido para sua utilização com programas de diferentes funcionalidades.

Palavras-chave: programação em N versões, tolerância à faltas, padrão de projeto, confiabilidade, sistemas distribuídos.

ABSTRACT

As modern society becomes increasingly dependent on computers and communication networks, the susceptibility to malfunction and eventual failure of these systems may significantly affect people's daily lives, both socially and economically. Therefore, highly reliable computer systems are becoming ever more necessary. Nevertheless, the implementation of these systems is expensive, as it requires the use of techniques based on the exploitation of redundancy. These techniques often demand the inclusion of several instances of each system component, with the same functionality, so as to enable its recovery or replacement in case of faults. In order to meet reliability criteria design patterns may be utilized in the design of hardware and software systems. The N-Version Programming pattern is used to tolerate software faults and, if implemented in a distributed architecture, may also tolerate hardware faults. This work proposes a distributed structure for the implementation of the N-Version Programming pattern which eliminates single points of failure in both software and hardware. This structure is implemented in a generalized fashion, with the goal of lowering the required effort for its use with programs with different functionalities.

Keywords: n-version programming, fault tolerance, design pattern, reliability, distributed systems.

LISTA DE FIGURAS

LISTA DE QUDROS

SUMÁRIO

1. INTRODUÇÃO

A programação em N Versões é um método de tolerância à faltas de *software* muito conhecido que realiza mascaramento de faltas baseado na diversidade de *software*. A ideia do presente modelo é baseada no conceito da geração independente de N versões de software funcionalmente equivalentes a partir de uma mesma especificação inicial. Os resultados da execução dessas versões são enviados para um votador que executa uma estratégia de votação para determinar a saída correta. O objetivo do padrão de programação N Versões é minimizar a probabilidade de faltas simultâneas nas várias versões (ARMOUSH, 2010).

Em adição às faltas de software, para se cobrir as faltas de *hardware* fazendo o uso da programação em N Versões é necessário que, não somente as versões sejam distribuídas em vários computadores, como também se fazer uso de uma estrutura onde não existam pontos únicos de falha.

De acordo com (ARMOUSH, 2010) o ponto único de falha da estrutura de programação em N versões é o votador. Assim sendo, este trabalho incluiu o desenvolvimento de um *framework* para a implementação da programação em N versões de forma distribuída, de modo a remover do votador a condição de ponto único de falha.

Outro objetivo deste trabalho foi implementar o padrão de N versões através de uma estrutura genérica, que incorporasse componentes de gerenciamento de entrada e saída abrangentes, permitindo a sua reutilização com diversas aplicações no formato N versões.

Este trabalho foi organizado da forma seguinte. A seção 2 apresenta o referencial teórico do trabalho. A seção 3 propõe uma solução para o problema apresentado incluindo os requisitos do sistema, sua arquitetura e seus modelos fundamentais. A seção 4 compreende a discussão de aspectos específicos do trabalho desenvolvido. A seção 5 considera trabalhos relacionados quanto a Programação em N Versões e Tolerância à Faltas. A seção 6 completa o trabalho com observações de conclusão.

2. REFERENCIAL TEÓRICO

Este capítulo tem como objetivo abordar os conceitos de Falta, Erro e Falha, Confiabilidade e Tolerância à Faltas, além do próprio Padrão de Programação em N Versões.

2.1. FALTA, ERRO E FALHA

Na linguagem cotidiana, falta, erro e falha são termos usados indistintamente. No contexto da computação confiável, porém, eles têm significados diferentes. Uma falta pode ser um defeito físico de hardware ou um defeito de projeto de software, ou de programação (bug). Em contraste, um erro é o resultado da manifestação de uma falta quando esta é ativada. Quando o erro afeta perceptivelmente o serviço externo provido pelo sistema, desviando-o de suas especificações, caracteriza-se uma falha (KOREN e KRISHNA, 2007).

De acordo com o *IEEE Standard Glossary of Software Engineering Terminology* (1991), os seguintes conceitos são definidos como:

- Falta de *hardware*: Um defeito em um dispositivo de hardware ou componente, por exemplo, um curto-circuito ou um fio partido.
- Falta de *software*: Um passo incorreto, processo ou dados incorretos em um programa de computador.
- Erro: A diferença entre, uma condição ou valor computado, observado, ou medido e o valor (condição) verdadeiro, especificado ou teoricamente correto.
- Falha: A incapacidade de um sistema ou componente de executar suas funções requeridas dentro de requisitos de desempenho especificados

2.2. CONFIABILIDADE E TOLERÂNCIA À FALTAS

Segundo Pradhan (WEBER, 2002) confiabilidade é capacidade de um sistema atender a suas especificações, dentro de condições definidas, provendo serviço continuamente durante certo período de tempo considerando-se que estava operacional no início deste período.

Confiabilidade é uma medida de probabilidade, pois a ocorrência de falhas pode ser matematicamente tratada como um fenômeno aleatório. Não deve ser confundida com disponibilidade, que é a probabilidade de um sistema estar provendo serviço em um determinado instante de tempo. Um sistema pode ser de alta disponibilidade e ainda assim ter uma confiabilidade relativamente baixa (WEBER, 2002).

No projeto de sistemas de alta confiabilidade faz-se necessária a utilização de técnicas de tolerância à faltas, que superem (tolerem) as faltas em tempo de execução, impedindo a ocorrência de falhas. Essas técnicas garantem o funcionamento correto do sistema mesmo na

ocorrência de faltas e são geralmente baseadas na exploração inteligente de redundância, exigindo componentes adicionais ou algoritmos especiais (WEBER, 2002).

(KOREN e KRISHNA, 2007) apresentam à tolerância à faltas como um exercício de exploração e gestão de redundância. A redundância é a propriedade de se ter recursos adicionais, além dos necessários para o funcionamento correto do sistema. A redundância é explorada para mascarar ou contornar de outra forma estas falhas, mantendo assim o nível desejado de funcionalidade.

As técnicas de tolerância à faltas podem ser classificadas em duas classes básicas: 1) mascaramento de faltas; e 2) detecção e localização de faltas, seguidas de reconfiguração do sistema. Na primeira classe, mascaramento, faltas não se manifestam como falhas, pois são mascaradas na origem. Na segunda classe faltas são detectadas, os componentes onde elas ocorreram são isolados e o o sistema é reconfigurado, excluindo tais componentes. A primeira classe geralmente emprega mais redundância que a segunda e, por não envolver os tempos gastos para as tarefas de detecção, localização e reconfiguração, é a preferida para sistemas de tempo real críticos (WEBER, 2002).

2.3. PADRÃO DE PROGRAMAÇÃO EM N VERSÕES

Redundância tem sido muito utilizada em engenharia, especialmente em *hardwares*, para aumentar confiabilidade de sistemas que operam em ambientes incertos. A chave da compreensão de sua utilização é que, se uma instância de um componente falhar, uma instância alternativa, redundante, estará disponível e será acionada para substituí-lo (KHOURY, 2012).

A adição de instâncias replicadas do mesmo software não se faz suficiente para a superação de faltas de *software*, porque a ocorrência de uma falta será comum a todas as instâncias quando estas forem submetidas à mesma sequência de entradas. Para que a exploração de redundância seja funcional, é necessário também incluir, em algum nível, a diversidade de *software* (KOREN e KRISHNA, 2007).

As áreas tradicionais onde são empregados sistemas tolerantes à faltas são (WEBER, 2002):

- Aplicações críticas de sistemas de tempo real como medicina;
- Controle de processos e transportes aéreos;
- Aplicações seguras de tempo real como transportes urbanos;

- Aplicações em sistemas de tempo real de longo período de duração sem manutenção, como em viagens espaciais, satélites e sondas;
- Aplicações técnicas como telefonia e telecomunicações;
- Aplicações comerciais de alta disponibilidade como sistemas de transação e servidores de redes.

O padrão de programação em N versões foi utilizado em sistemas como:

- *Space shuttle*: ônibus espacial que apresenta 2 versões do seu sistema de aviação;
- *Boeing* 777: Sistema de controlo de voo, um único programa foi desenvolvido, mas 3 processadores diferentes, e 3 compiladores diferentes foram utilizados para obter a diversidade;
- *Airbus* A320/330/340: O sistema *fly-by-wire* é composto por 5 computadores em execução simultânea.

O Padrão de Programação em N Versões, apresentado a seguir, foi descrito em (ARMOUSH, 2010)

2.3.1. Nome do Padrão

N-Version Programming Pattern (NVP) – Padrão de Programação em N Versões

2.3.2. Outros nomes

Master-Slave Pattern (Buschmann, 1995 *apud* Armoush, 2010) - Padrão Mestre-Escravo.

2.3.3. Tipo

Padrão de Software

2.3.4. Resumo

A programação em N Versões é um método de tolerância à falta de software muito conhecida baseado na diversidade de software e mascaramento de faltas. É definida como a geração independente de N >= 2 módulos de software funcionalmente equivalentes chamados de versões a partir da mesma especificação inicial. Esse padrão inclui N versões distintas, mas funcionalmente idênticas, de um programa que estão rodando em paralelo para realizar a mesma tarefa com a mesma entrada, produzindo N saídas. Um votador é usado para produzir o resultado correto; ele recebe N resultados como entrada e usa esses resultados para determinar a saída correta de acordo com um esquema de votação específico.

2.3.5. Contexto

Desenvolver um software tolerante à falta para um sistema de segurança altamente crítica em situações onde:

- Um software altamente confiável é necessário
- O alto custo de desenvolvimento de múltiplas versões pode ser coberto
- Existem N equipes de desenvolvimento disponíveis para desenvolver as diferentes versões
- Existe a possibilidade de usar N unidades redundantes de hardware para executar essas versões em paralelo

2.3.6. Problema

Como superar (tolerar) faltas de software, que possam ter permanecido após o desenvolvimento do sistema, a fim de melhorar sua confiabilidade e segurança em tempo de execução.

2.3.7. Estrutura do Padrão

O padrão de programação N Versões é uma abordagem de tolerância à faltas bem conhecida que é normalmente utilizada apenas em aplicações de alta criticidade, devido ao seu alto custo de desenvolvimento (ARMOUSH, 2010).

A ideia do presente modelo é baseada no conceito de geração independente de N versões funcionalmente equivalentes a partir de uma mesma especificação inicial. Os resultados da execução destas versões são enviados para o votador que executa uma estratégia de votação para determinar a saída correta. O objetivo do padrão de programação N Versões é minimizar a probabilidade de ocorrência de faltas simultâneas na execução das várias versões.

A estrutura deste modelo é mostrada na Figura 1, em que a função de cada unidade é apresentada.

Fonte de Entrada de Dados: Ela representa a fonte de informação que é usado como entrada para o sistema desenvolvido. Normalmente, estes dados são providos pelo utilizador do sistema, ou por sensores externos utilizados para monitorar o valor corrente de algumas variáveis ambientais tais como: temperatura, pressão, velocidade, luz, etc.

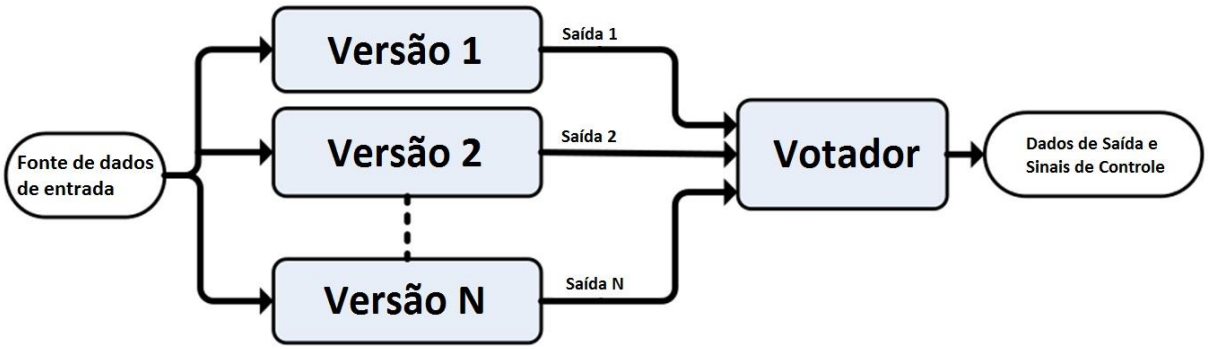

Figura 1 - Padrão de Programação N Versões

Saída de dados e sinais de controle: Os dados de saída podem conter alguns sinais de controle para ativar atuadores ou mensagens para outros componentes fora do sistema.

Versão 1, 2 ... N: Estas versões são o resultado de N implementações do software por equipes de programadores independentes, com base na mesma especificação inicial. Assim, estas versões são funcionalmente equivalentes e executam a mesma ação sobre os dados de entrada para produzir o resultado final. Normalmente, estas diversas versões são executados em paralelo em diferentes dispositivos de hardware para gerar N saídas que são posteriormente processadas pelo votador para determinar o resultado melhor ou mais correto, com base em uma estratégia de votação. Também é possível executar estas versões sequencialmente no mesmo hardware, mas neste caso, aumentaria o tempo de execução em, pelo menos N vezes, o que torna esta escolha menos atraente.

Votador: O votador recebe os N resultados das diversas versões como entrada, e as usa para determinar o resultado final e sinais de controle baseando-se em uma técnica de votação específica, como a votação por maioria (*majority voting*), que é a técnica mais comumente utilizada. A técnica da maioria é uma variante da técnica M-*out-of*-N, onde

$$\boldsymbol{M} = \left\lceil \frac{(\boldsymbol{N}+\mathbf{1})}{\mathbf{2}} \right\rceil \qquad (1)$$

Ele produz uma saída que representa uma concordância entre, pelo menos, M versões.

2.3.8. Implicações

Esta seção mostra as implicações da utilização deste padrão com respeito a alguns requisitos não funcionais tais como confiabilidade, segurança operacional, custo e modificabilidade, usando uma votação por maioria em relação ao sistema básico.

2.3.8.1. Confiabilidade

Sob a hipótese de que o votador é livre de falhas, o padrão NVP continuará a prover um resultado correto, desde que, pelo menos, M versões não contenham faltas comuns.

A melhoria relativa percentual da confiabilidade (*Reliability Relative Improvement* - RRI) é calculada com a confiabilidade obtida (R_{new}) em relação a confiabilidade anterior (R_{old}) de software é:

$$\boldsymbol{RRI = \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\%} \quad \textbf{(2)}$$

$$\boldsymbol{RRI = \frac{\left(\sum_{i=M}^{N} \binom{N}{i} (R)^{i} (1-R)^{N-i}\right) - R}{1 - R} \times 100\%} \quad \textbf{(3)}$$

2.3.8.2. Segurança Operacional

A segurança operacional é a capacidade existente em um sistema de falhar sem causar danos humanos, materiais ou ambientais.

O padrão NVP inclui duas técnicas de projeto: programação diversificada e mascaramento de faltas por meio de votação. De acordo com os requisitos de software no padrão IEC 61508-3 (1998), as recomendações para estas técnicas são mostradas na Quadro 1.

Quadro 1 - Recomendações para cada nível de segurança de integridade

Técnicas	***SIL1***	***SIL2***	***SIL3***	***SIL4***
Programação diversificada	R	R	R	AR
Detecção de falta com votação	-	R	AR	AR
PR – Pouco recomendado R - Recomendado AR – Altamente Recomendado				

SIL - *Safety Integrity Level* - é uma maneira de indicar a taxa de falha tolerável de uma função de segurança operacional em particular (MAGNETROL INTERNATIONAL, 2012).

De acordo com o Quadro 1, as recomendações médias deste padrão para a integridade de diferentes níveis de segurança operacional estão apresentadas na Quadro 2.

Quadro 2 - Recomendações do Padrão NVP para cada nível de segurança de integridade

Padrão (*Pattern*)	***SIL1*** [1]	***SIL2***	***SIL3***	***SIL4***
Padrão de Programação de N Versões	PR	R	MR	AR
PR – Pouco recomendado R - Recomendado MR – Muito Recomendado AR – Altamente Recomendado				

A segurança operacional nesse padrão depende de que as versões que falharam não tenham produzido a mesma saída incorreta constituindo uma maioria no processo de votação. No caso de um amplo espectro de saída, é improvável que as versões que porventura falharem produzam resultados errados semelhantes, o que minimiza a probabilidade de se encontrar uma maioria entre as saídas incorretas. Esta condição permite se modificar o padrão para levar o sistema a um estado de falha segura. Por outro lado, o pior caso ocorre quando M versões produzirem resultados incorretos semelhantes na saída lógica. Neste caso, os resultados incorretos podem constituir uma maioria no processo de votação, levando a uma condição de falha do sistema em que não serão tomadas precauções de segurança.

Para calcular o RSI (*Relative Safety Improvement* – Melhoria Relativa da Segurança Operacional) considera-se a probabilidade de falha do sistema original $P_{UF(old)}$:

$$P_{UF(old)} = 1 - R = f \quad (4)$$

O padrão NVP conterá falhas em nível de sistema somente quando pelo menos M módulos apresentarem o mesmo resultado errôneo. Portanto, $P_{UF(new)}$, a probabilidade de falha do novo sistema utilizando este padrão é dada por:

$$P_{UF(new)} = \sum_{i=M}^{N} \binom{N}{i} (1-R)^i (R)^{N-i} \quad (5)$$

A melhoria da segurança operacional em relação percentual é:

$$\begin{aligned} RSI &= \frac{P_{UF(new)} - P_{UF(old)}}{0 - P_{UF(old)}} \times 100\% \\ RSI &= \left(1 - \frac{P_{UF(new)}}{P_{UF(old)}}\right) \times 100\% \\ RSI &= \left(1 - \frac{\sum_{i=M}^{N} \binom{N}{i} (1-R)^i (R)^{N-i}}{1-R}\right) \times 100\% \\ RSI &= \left(1 - \frac{\sum_{i=M}^{N} \binom{N}{i} (f)^i (1-f)^{N-i}}{f}\right) \times 100\% \end{aligned} \quad (6)$$

2.3.8.3. Custo

Este padrão é um dos padrões mais caros. O custo pode ser dividido em duas partes.

Custo recorrente: inclui o custo das N unidades diferentes de hardware a ser utilizadas para a execução em paralelo das N versões do software. Assim, o custo de retorno será (N × 100%) em comparação com o sistema original que utilizava uma única versão.

Custo de desenvolvimento: O desenvolvimento independente e funcionalmente equivalente das N Versões de software pode ser estimado como se segue:

As N versões compartilham as mesmas especificações. Portanto apenas um conjunto de especificações precisa ser desenvolvido (KOREN e KRISHNA, 2007).

O custo para o desenvolvimento de N versões antes da fase de verificação e de validação é N vezes o custo de desenvolvimento de uma versão única (KOREN e KRISHNA, 2007).

As diferentes versões podem ser usadas para validar umas às outras (no período de testes), o que vai diminuir o custo de ferramentas de verificação e validação (KOREN e KRISHNA, 2007).

A gestão de um projeto de N versões impõe uma sobrecarga não encontrada no desenvolvimento de software tradicional (KOREN e KRISHNA, 2007).

Informações exatas sobre o custo de se criar N versões da mesma especificação são limitadas. Dados disponíveis sobre o custo de desenvolvimento de softwares multi-versão mostram que este custo aumenta sub linearmente com o número de componentes de software. Além disso, cada versão adicional custa cerca de 75% a 80% do custo de uma única versão (ARMOUSH, 2010).

2.3.8.4. Modificabilidade

Existem três tipos de modificações possíveis:

- A modificação de uma única versão: É possível modificar uma única versão, ou para remover uma falta recém-descoberta, ou para melhorar a função mal programada (ARMOUSH, 2010). Neste caso, a especificação inicial permanece sem qualquer modificação, e a modificação desta versão é similar à modificação do software de versão única seguindo um procedimento de remoção de falta (bug) padrão.
- Modificação de todas as versões membro: a razão para esta modificação é, ou para adicionar uma nova funcionalidade, ou para melhorar o desempenho global das N versões do software (ARMOUSH, 2010). Neste caso, a especificação inicial deve ser modificada e, todas as N versões devem ser alteradas e testadas de forma independente por equipes distintas para implementar os ajustes necessários nos módulos afetados.

Em geral, a modificação das N versões do software é notavelmente mais difícil do que a modificação do software de versão única.

- Modificação do votador: A separação do votador das N versões permite modificações fáceis ou alterações da técnica de votação.

2.3.8.5. Impacto da utilização do padrão sobre o Tempo de Execução

Existem dois métodos para se executar um software com N Versões: pode-se utilizar a execução paralela, que é o mais comum, ou a execução sequencial.

No caso de execução paralela em N unidades independentes de hardware (idênticas ou não), as diferentes versões podem implicar em tempos de execução diferentes, uma vez que são implementadas por diferentes equipes de programadores. Consequentemente, o votador precisa esperar todas as saídas para iniciar o algoritmo de votação. Assim, o tempo total de execução é determinado pelo tempo requerido pela versão mais lenta, mais o tempo, relativamente pequeno, necessário para a execução do algoritmo de votação. Em geral, se negligenciarmos o tempo de execução do votador, então o tempo de execução do software com N versões é ligeiramente igual a um software de versão única.

No caso da execução sequencial, o tempo de execução aumentará N vezes. Esta desvantagem faz com que a execução sequencial seja menos atraente, especialmente para aplicações de tempo crítico (*time-critical*).

2.3.9. Implementação

O sucesso do padrão NVP depende do desenvolvimento independente das N versões requeridas e do nível de diversidade real existente nas versões resultantes, o que possibilita que faltas comuns sejam evitadas. A fim de aumentar o nível de diversidade e a independência das versões projetadas, os problemas de aplicação apresentados abaixo devem ser levadas em consideração tanto quanto possível durante o processo de desenvolvimento:

- A utilização de especificações completas, corretas, e cuidadosamente documentadas para impedir a propagação de erros de especificação para as versões;
- O uso de equipes independentes e isoladas de programadores com diversidade na sua formação e experiência;
- A utilização de diferentes algoritmos;
- O uso de diferentes linguagens de programação;
- O uso de diferentes compiladores, ferramentas de desenvolvimento e métodos de teste;

- O uso de diversas técnicas de implementação.

Existem várias técnicas de votação que podem ser usadas neste padrão para implementar o componente votador, como:

- *Votação por maioria*: É o método mais simples e mais comum que pode ser utilizado para se encontrar a saída. A quantidade de resultados diferentes que precisam concordar entre si é dada pela Equação 7, onde N é a quantidade de versões.

$$\left\lfloor \frac{(N+1)}{2} \right\rfloor \quad (7)$$

- *Pluralidade de votação* (PV): É um eleitor simples, que implementa uma votação M-*out-of*-N, onde M é menor do que a rigorosa maioria.
- *Votação de consenso* (CV): Este método de votação é usado para software multi-versão com espaço de saída pequeno. Neste método, o resultado de maior concordância é escolhido como a saída correta (ARMOUSH, 2010).
- *Votação de máxima probabilidade* (MLV): O eleitor usa a confiabilidade de cada versão para fazer uma estimativa mais precisa do resultado correto mais provável (ARMOUSH, 2010).
- *Votação adaptativa*: Apresenta um fator de ponderação individual para cada versão que é posteriormente incluído no processo de votação. Estes fatores de ponderação são dinamicamente mutáveis para modelar e gerenciar diferentes níveis de qualidade das versões (ARMOUSH, 2010).

A escolha da técnica de votação mais apropriada depende do tipo de dados utilizados, do desvio nas saídas das versões, do tipo de concordância de resultados desejado (ARMOUSH, 2010), do tamanho e cardinalidade do espaço de saída, da funcionalidade do votador (ARMOUSH, 2010), da confiabilidade das diferentes versões, e de vários outros fatores.

2.3.10. Consequências e Efeitos

Os principais inconvenientes do Padrão NVP são: a complexidade do desenvolvimento de N versões independentes, além de seu custo de desenvolvimento elevado, e a forte dependência com respeito à especificação inicial, que pode propagar falhas correlacionadas a todas as versões. O problema de falhas correlacionadas é crítico na utilização deste padrão com respeito à segurança operacional e à confiabilidade do sistema.

2.3.11. Padrões Relacionados

O padrão mais relacionado é o Padrão Bloco de Recuperação que utiliza diversas versões de software com testes em vez de usar um algoritmo de votação de aceitação. Por outro lado, a programação N versões é usada para melhorar a confiabilidade do software, e utiliza múltiplas unidades. Assim, é possível combinar este padrão com o Padrão de Design Heterogêneo para o projeto dessas unidades de hardware diferentes para lidar com as falhas sistemáticas de hardware. Esta combinação vai melhorar a confiabilidade e segurança do hardware, bem como a do software.

2.4. TRABALHOS RELACIONADOS

Com o objetivo de entender e levantar os diversos problemas que a Programação em N Versões pode implicar, foram analisados alguns trabalhos correlatos recentes, que têm como objetivo discutir os aspectos de implementação deste padrão (apresentados na seção 2.3.9).

2.4.1. Diversidade através da programação N-Version: Estado Atual, Desafios e Recomendações

O trabalho de Khoury apresenta resultados de várias experiências que foram realizados para avaliar os benefícios da programação em N versões e traz algumas recomendações levantadas com base neste estudo. Khoury ainda conclui que a diversidade para ser eficaz, deve ser introduzida de forma orientada e informada e abranger o maior número possível de fases de desenvolvimento do software (KHOURY, 2012).

O trabalho referido apresenta o estado da arte em relação à diversidade de software. Ele também nota que as abordagens propostas podem ser classificadas com base na fase de desenvolvimento de software na qual a diversidade é introduzida. Podem ser distinguir quatro tipos principais de diversidade introduzidas em um software: a diversidade de implementação, a diversidade das linguagens de programação, a diversidade de design e, finalmente, a diversidade de dados (KHOURY, 2012).

2.4.2. Padrões de Projeto para Implementação de Segurança e Tolerância a Falhas

O trabalho de Gawand, Mundada e Swaminathan apresenta uma abordagem orientada a objetos baseada em padrões de projetos e conceitos de reflexão computacional para sistemas de controle complexos (GAWAND, MUNDADA e SWAMINATHAN, 2011).

Para atingir esses objetivos, Gawand, Mundada e Swaminathan, analisam diversos padrões de projetos como TMR (*Triple Module Redundance*) e o Padrão de Projeto Reflexivo,

um refinamento do Padrão de Projeto *State* (GAWAND, MUNDADA e SWAMINATHAN, 2011).

O aspecto de tolerância à faltas é abordado com uma variante do Padrão de Projeto Reflexivo. Esta variante é baseada em uma máquina de estados com um componente responsável por prover serviços de tolerância à faltas utilizando estados redundantes de sistema.

2.4.3. Técnicas de Tolerância a Faltas de *Software*

O trabalho de Joshi tem como objetivo introduzir algumas técnicas de tolerância à faltas. Joshi traz uma revisão em relação a: (a) Programação em N Versões e os seus principais problemas: o isolamento de faltas entre as versões, o custo de implementação e a perda de desempenho ocasionada por sua utilização; e (b) Blocos de Recuperação, que é uma técnica baseada em testes de aceitação e recuperação modular de faltas (JOSHI, 2012).

Também são apresentadas comparações entre as técnicas de tolerância à faltas, e suas limitações e perspectivas são discutidas.

2.4.4. Sobre Faltas Representativas de Injeções de Faltas de Software On

Natella apresentam em seu trabalho um estudo experimental para avaliar a representatividade das injeções de faltas sugeridas pela técnica de injeção de faltas chamada G-SWFIT[1] (*Generic Software Fault Injection Technique*), com base em estudos recentes de dados de campo que caracterizaram falhas de software residuais em sistemas complexos (Cotroneo *et al*, 2012).

Os resultados obtidos por Natella, através de mais de 3.8 milhões de experimentos individuais em três sistemas reais, apontam que 72% das injeções de faltas realizadas não são representativas de faltas residuais de software, isto é, tais faltas poderiam ser facilmente detectadas por baterias de teste normalmente utilizadas na fase de verificação e validação de software. Os autores propõem a remoção destas faltas não representativas e uma nova abordagem para refinar o modelo de injeção de faltas (NATELLA *et al*, 2011).

O objetivo da aplicação desse filtro é obter resultados mais significativos e reduzir o custo de campanhas de injeção de faltas em softwares complexos (NATELLA *et al*, 2011).

[1] G-SWFIT é uma técnica de injeção de faltas, considerada como o estado da arte nesta área, que: a) define quais tipos de faltas de software devem ser introduzidas a fim de emular realisticamente um software defeituoso e injeta instâncias destas faltas usando mutação do código binário executável de um programa.

2.4.5. Separação de Tolerância a Faltas e Aspectos Não Funcionais: Padrões Orientados a Aspecto e Avaliação

Hameed, Williamns e Smith propõe uma separação de preocupações (*concerns*) entre os requisitos funcionais e não funcionais de sistema, considerando que os aspectos não funcionais são transversais às funcionalidades (HAMEED, WILLIANMS e SMITH, 2010).

Para realizar essa separação, levando em contas aspectos da dependabilidade, Hameed, Williamns e Smith, propõe um *framework* orientado a aspectos que modulariza e separa aspectos de *design* e implementação da dependabilidade do núcleo das funcionalidades (HAMEED, WILLIANMS e SMITH, 2010).

3. DESCRIÇÃO DA SOLUÇÃO

Para se cobrir as faltas de *hardware* fazendo o uso da programação em N Versões é necessário que, não somente, o sistema seja distribuído em vários computadores, como também se fazer uso de uma estrutura onde não existam pontos únicos de falha, que de acordo com a estrutura proposta por (ARMOUSH, 2010), é constituída pelo o votador.

Um ponto único de falha é um componente do sistema que não está protegido por redundância/replicação e, portanto, em caso de falta, provoca a falha do sistema (DOOLEY, 2001).

Dessa forma, neste trabalho foi desenvolvido um *framework* para a implementação do padrão de programação em N versões de forma distribuída, com a remoção do ponto único de falha – o votador – e com a adição de componentes de gerenciamento de entradas e saídas que tornassem a estrutura mais genérica, possibilitando a sua utilização com aplicações diversas no formato de N versões.

A arquitetura do *framework* conta com componentes redundantes – para a superação de faltas tanto de software quanto de hardware – e com componentes que monitoram a execução da aplicação, tornando possível a detecção de faltas e a tomada de ações para a superação das mesmas.

3.1. REQUISITOS

3.1.1. Visão Geral

O *framework* para o Padrão de Programação em N Versões (ARMOUSH, 2010) fornece uma maneira genérica e distribuída para suportar o desenvolvimento de N versões de um sistema, com a finalidade de superar não somente as faltas de *software* – que o padrão se propõe tratar – como também as faltas de *hardware*.

Fazendo o uso deste *framework* as equipes de desenvolvimento poderão focar seu esforço somente na aplicação a ser desenvolvida, sem a preocupação com os outros componentes do sistema – como o votador e os gerenciadores de entrada e saída – diminuindo consideravelmente o esforço de desenvolvimento.

O objetivo do framework é melhorar a confiabilidade e disponibilidade em sistemas que requerem estes requisitos não funcionais, de forma que a ocorrência de faltas nas versões individuais sejam mascaradas pela votação dos resultados das demais versões. No padrão original uma falta no votador caracterizaria uma falha do sistema, pois o votador é um ponto único de falha (*single point of failure*) nesta arquitetura. Esse problema foi solucionado na

variante do *framework* implementado com o uso de redundância do votador, que opera com duas instâncias no modo ativo/*standby*.

3.1.2. Requisitos Funcionais

RF1 – O sistema deve distribuir as requisições recebidas para as N versões que compõe o sistema – Distribuidor de Entrada.

RF2 – O sistema deve coletar os resultados produzidos por cada uma das N versões e enviá-los para o processo de votação – Coletor de Resultado.

RF3 – O sistema deve realizar a votação dos resultados produzidos pelas N versões - Votador.

RF4 – O sistema deve possuir dois conjuntos independentes de componentes auxiliares, cada um contendo um Distribuidor de Entrada, um Coletor de Resultado e um Votador.

RF5 – O sistema deve possuir um conjunto de componentes auxiliares funcionando como primário e outro funcionando como secundário.

RF6 – O sistema deve possuir duas instâncias de cada um dos componentes auxiliares, uma ativa e outra em espera.

RF7 – O sistema deve monitorar periodicamente seus componentes auxiliares redundantes.

RF8 – O sistema deve colocar em estado ativo uma instância de um componente auxiliar secundário que se encontrava em estado de espera, quando esta for acometida por uma falha. .

RF9 – O sistema deve colocar preferencialmente em estado ativo as instâncias do conjunto de componentes auxiliares primários.

3.1.3. Requisitos Não Funcionais

RNF1 – O sistema deve possuir uma lista de controle de acesso.

RNF2 – O sistema deve conferir se a requisição recebida se encontra na lista de controle de acesso antes de aceitar tal requisição.

3.2. ARQUITETURA

Para eliminar os pontos únicos de falha do sistema é necessário inserir redundância adicional, de forma que ele tenha pelo menos mais uma unidade de cada componente vulnerável (*no single point of failure*). Um possível desenho desse projeto utilizando uma

quantidade mínima de *hosts* adicionais sem que o sistema fique comprometido é apresentado na Figura 2. Nesta arquitetura as N versões do aplicativo rodam cada uma em um dentre N *hosts* (na implementação em particular N = 3) distintos. Dois hosts adicionais são utilizados para executarem cada uma das instâncias (ativa e em espera) dos componentes auxiliares do sistema (distribuidor de entrada, coletor de resultados e votador).

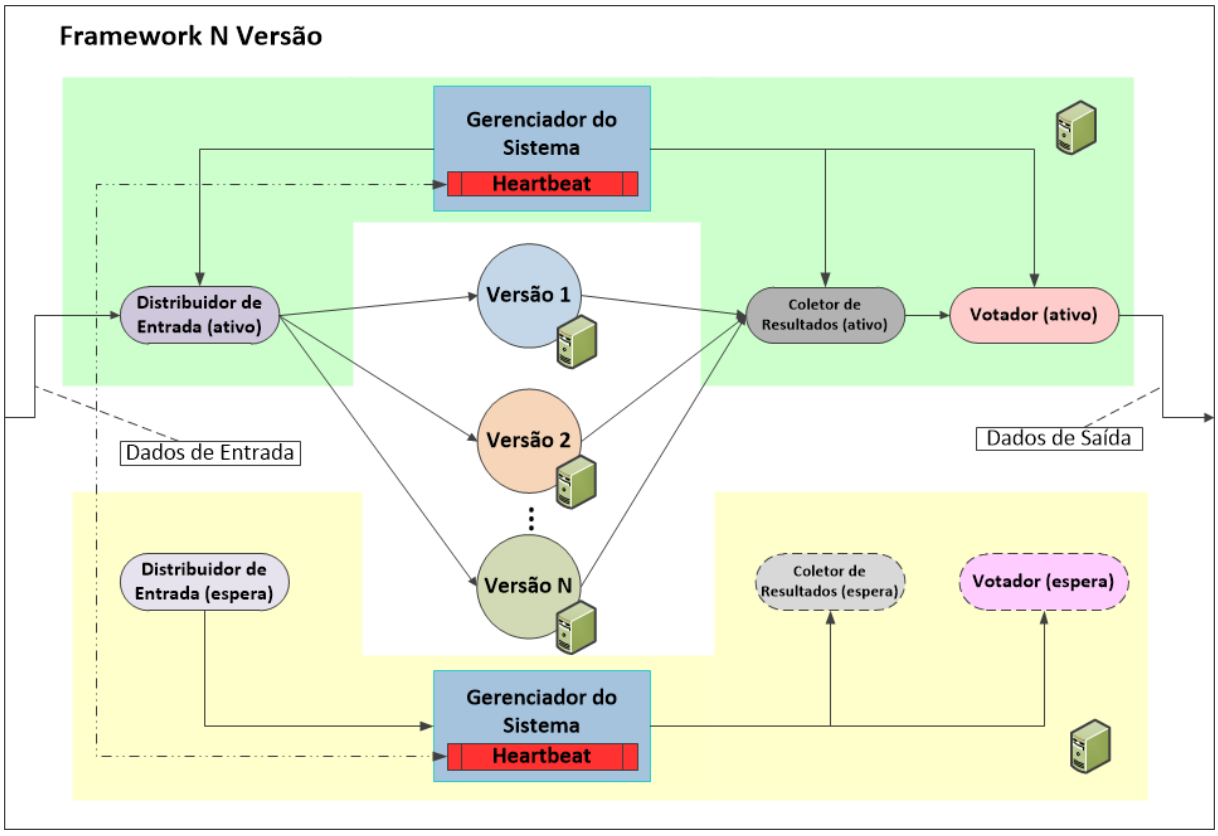

Figura 2 - Projeto Distribuído do Padrão de Programação N Versões com Redundância na infraestrutura

Uma instância do componente denominado *System Manager* – Gerenciador do Sistema – é executada em cada um dos hosts adicionais e tem a finalidade de monitorar o funcionamento dos dois conjuntos de instâncias, um ativo e o outro em espera (*standby*), dos componentes auxiliares, e realizar a troca para o estado ativo de uma determinada instância em estado de espera, em caso de falha na instância correspondente previamente em estado ativo (*failover*), ou realizar a troca do conjunto de instâncias previamente ativo pelo conjunto de instâncias em espera (o qual se tornará ativo) em caso de falha de um dos *hosts* onde estes conjuntos são executados.

O Gerenciador de Sistema também poderia ser usado para monitorar as N versões independentes da aplicação com a finalidade de realizar um *restart* de uma versão em caso de falha desta. Para isto, é necessário executar uma instância do Gerenciador de Sistema em

cada um dos três hosts onde rodam as 3 versões do aplicativo. O Gerenciador de Sistema também poderia ser programado para fazer este *restart* até um número máximo de vezes (configurável) dentro de um determinado intervalo de tempo (também configurável) de modo a deixar esta versão em estado de falha permanente (sem *restart*) quando a falha nesta instância for recorrente. Neste trabalho, isto não foi implementado porque a tolerância à faltas do aplicativo já foi implementada através do padrão de N versões. Caso o fizéssemos isto aumentaria ainda mais a confiabilidade e disponibilidade da aplicação. Esta opção fica, porém, na lista de trabalhos futuros.

No sistema distribuído com redundância foi utilizada a técnica de *failover*, onde caso a instância ativa (*active*) de um componente falhe, a instância correspondente em estado de espera (*standby*) toma o seu lugar e o sistema continua a funcionar normalmente, eliminando assim a possibilidade um ponto único de falha. E caso o *host* onde as instâncias ativas são executadas sofra uma falha, afetando todos os componentes que rodam deste *host*, o *host* onde roda a versão em espera (*standby*) assume suas funções, evitando assim a ocorrência de uma falha de sistema.

3.3. MODELOS FUNDAMENTAIS

Os modelos fundamentais do *framework* implementado que apresentamos a seguir têm como objetivo abordar aspectos tais como desempenho e confiabilidade dos processos e das redes de comunicação, assim como a segurança dos recursos presentes no sistema. Para tal, é necessário explicitar as suposições sobre o sistema em questão e realizar generalizações acerca de sua modelagem (COULOURIS, DOLLIMORE e KINDBERG, 2007).

3.3.1. Modelo de Interação

Em sistemas distribuídos a computação é realizada por processos rodando em hardwares distintos, que por sua vez interagem entre si através de mensagens enviadas/recebidas através de uma rede, resultando na comunicação e na coordenação entre eles (COULOURIS, DOLLIMORE e KINDBERG, 2007).

O modelo de interação é responsável por apresentar os componentes do sistema e os aspectos que podem interferir na comunicação dos mesmos.

A arquitetura do sistema foi apresentada na Seção 3.2.

Os aspectos que podem influenciar na comunicação dos componentes do sistema são apresentados na Quadro 3 (COULOURIS, DOLLIMORE e KINDBERG, 2007).

Quadro 3 - Aspectos que interferem na comunicação em sistemas distribuídos

Desempenho *Jitter*	Variação na latência de um pacote (*packet*) para o próximo. Mudanças nos tamanhos dos pacotes e congestionamentos temporários.
Desempenho *Timeout*	Tempo de espera de uma resposta esperada a ser enviada por um componente antes de se concluir sobre o estado de falha deste componente.
Desempenho *Latência*	Diferença de tempo entre o início de um evento e o momento em que seus efeitos tornam-se perceptíveis.
Aspecto temporal	Diz respeito ao aspecto temporal no sistema distribuído, uma vez que este pode ser síncrono, quando se conhecem os tempos máximos de transmissão das mensagens, ou assíncrono, quando tais tempos são indeterminados.

O *framework* desenvolvido é assíncrono uma vez que as partes envolvidas (*hosts*) não estão sendo executadas seguindo um mesmo *clock*.

3.3.2. Modelo de Segurança de Acesso

A natureza modular dos sistemas distribuídos, aliada ao fato de ser desejável que sigam uma filosofia de sistemas abertos, os expõe a ataques de agentes externos e internos. O modelo de segurança de acesso define e classifica as formas que tais ataques podem assumir, dando uma base para a análise das possíveis ameaças a um sistema e assim guiar o desenvolvimento de sistemas capazes de resistir a tais ameaças (COULOURIS, DOLLIMORE e KINDBERG, 2007). O modelo de segurança pode ser observado na Quadro 4.

No *framework* desenvolvido, foi implementado um controle de acesso de acordo com o endereço de IP do *host* originador de uma requisição recebida por uma das N versões do aplicativo. Este endereço de IP que deve ser previamente definido a fim de evitar roubo de recursos e vandalismo.

Com relação aos dados que trafegam na rede, estes são passados em formato de *bytes*, não sendo legíveis de forma direta. Para que seja feita a interpretação dos dados seria necessário que o atacante soubesse os tipos de dados que trafegam pela rede.

A indisponibilidade de tal informação dificulta o roubo dos dados. A utilização de criptografia na proteção destes dados, embora não implementada por não ser o foco do projeto, dificultaria ainda mais a sua apropriação indevida.

Quadro 4 - Ameaças, métodos de ataque e soluções para sistemas distribuídos

Ameaças	Método de Ataque	Soluções existentes
Vazamento	Escuta secreta	Criptografia
Interferência	Mascaramento	Autenticação
Interferência	*Replaying*	Autenticação
Interferência	Interferência de mensagem	Criptografia
Roubo de recurso	*Replaying*	Autenticação
Vandalismo	Negação de serviço	-
Vandalismo	*Replaying*	Autenticação
Código móvel	Inserção de código malicioso	*Sandbox*

3.3.3. Modelo de Faltas

A operação correta de um sistema distribuído é ameaçada quando ocorre uma falta em um dos computadores em que tal sistema é executado (incluindo falhas de *software*), ou em uma versão de software executada neste computador, ou ainda na rede de comunicação que os interliga. O modelo de faltas define e classifica as faltas possíveis fornecendo uma base para análise de seus efeitos em potencial e dando suporte ao projeto de sistemas capazes de tolerá-las, evitando a falha e mantendo correto o funcionamento do sistema (COULOURIS, DOLLIMORE e KINDBERG, 2007).

A Quadro 5 apresenta os tipos de falhas classificado por (COULOURIS, DOLLIMORE e KINDBERG, 2007) e suas respectivas soluções.

Quadro 5 - Tipos de falhas e soluções de projeto

	Solução
Falta por omissão do processo	As faltas de omissão de processo são tratadas com monitores dos componentes do sistema utilizando da técnica de *heartbeats*. Esta técnica consiste no envio de sinais periódicos para conferir se um processo está funcionando e, caso não esteja, comunica à instância do System Manager sob o qual ele está sendo executada a ocorrência da falha. Este, por sua vez, comunicará a falha ao outro Gerenciador do Sistema, que por sua vez ativará a instância do componente respectivo que estava em estado de espera.
Falta por omissão do canal	As faltas de omissão de um determinado canal de comunicação serão interpretadas como faltas de omissão dos processos que o utilizam pelo fato dos objetos remotos ficarem inacessíveis.
Falta de *crash* do *host*	O crash no *host* leva a uma falha de omissão de processo pelo fato do mesmo não estar acessível, logo esse problema é tratado pela solução das faltas por omissão do processo.
Falta de *crash* da rede de comunicação	O crash na rede de comunicação levaria a falta de comunicação dos hosts, o que caracteriza uma falha do sistema. Uma possível solução (não implementada) é adicionar uma comunicação redundante entre os *host*, para que exista mais de uma forma de para se realizar a checagem do funcionamento do *host* adjacente.
Faltas arbitrárias	As faltas arbitrárias de *software* que podem ocorrer no sistema são mitigadas com o uso dos resultados das N Versões e a realização de votação para se chegar ao consenso a respeito do resultado correto.

4. APRESENTAÇÃO E DISCUSSÃO DOS RESULTADOS

Nas subseções seguintes, serão apresentados os Problemas de Projeto e Soluções do trabalho desenvolvido, assim como uma discussão dos resultados obtidos.

4.1. PROBLEMAS DE PROJETO E SOLUÇÕES

Os principais problemas de projeto encontrado serão apresentados nas próximas subseções, assim como a solução desenvolvida.

4.1.1. Gerenciamento de Componentes

No *framework* desenvolvido, todos os componentes auxiliares – distribuidor de entrada, coletor de resultados e votador – estendem a classe `Componente`, que é responsável pelas operações de estado dos componentes – ativo e em espera.

A classe `Componente`, por sua vez, estende a classe `UnicastRemoteObject`, que é uma classe do pacote *server* do Java RMI. Dessa forma todos os componentes do *framework* são objetos remotos – objetos que podem receber invocações remotas (COULOURIS, DOLLIMORE e KINDBERG, 2007), podendo assim ser remotamente monitorados pelo Gerenciador do Sistema, que, por sua vez, tem acesso às operações de estado dos componentes utilizando Invocação de Método Remoto (RMI), já implementado na biblioteca padrão do Java.

O sistema Java RMI é um mecanismo que permite que um objeto em uma máquina virtual Java possa chamar métodos em um objeto em outra máquina virtual Java (ORACLE, -).

Objetos remotos possuem dois conceitos fundamentais apresentados por (COULOURIS, DOLLIMORE e KINDBERG, 2007):

- Referência de objeto remoto: objetos podem invocar os métodos de um objeto remoto, se tiverem a sua referência.
- Interface remota: todo objeto remoto tem uma interface remota especificando quais de seus métodos pode ser invocado de forma remota.

O RMI foi utilizado para facilitar o desenvolvimento do *framework*, pois dessa forma, as chamadas das operações dos componentes podem ser realizadas da mesma forma que um objeto local, o que facilita a comunicação entre o Gerenciador do Sistema e os componentes por ele monitorados e a comunicação entre as duas instâncias do Gerenciador do Sistema (que executam em *hosts* distintos).

A maior dificuldade encontrada no uso do Java RMI foi a falta de documentação quanto à configuração de uma das propriedades da JVM Java que precisava ser configurada para realizar a conexão com objetos remotos. Para se configurar essa propriedade o código abaixo pode ser utilizado da Figura 3.

```
System.setProperty("java.rmi.server.hostname", host);
```

Figura 3 - Código em Java para definir hostname nas propriedades do sistema

onde *host* é o endereço de IP do computador no qual o se encontra o objeto remoto.

4.1.2. Detecção de Falhas de *Hardware* nos Conjuntos de Componentes

A fim de detectar falhas de *hardware*, os dois Gerenciadores do Sistema que compõe o *framework* utilizam um mecanismo de *heartbeat* entre si.

O mecanismo de *heartbeat* funciona da seguinte forma: os nós emitem periodicamente mensagens "eu estou vivo". Quando uma sequência de *k* (onde *k* é configurável) mensagens de checagem é perdida (não é recebida por uma das instâncias do Gerenciador do Sistema), o nó omisso é declarado em estado de falha (KOREN e KRISHNA, 2007).

No *framework* desenvolvido, ao invés do envio de mensagens, foi usada uma chamada a um método remoto do Gerenciador do Sistema, com a finalidade de checar se ele está disponível ou não.

Dessa forma, na ocorrência de uma falha, o gerenciador do conjunto de componentes em espera pode tomar as ações necessárias para colocar estes seus componentes em estado ativo.

Um problema encontrado durante o desenvolvimento dessa funcionalidade do *framework* foi que, na ocorrência de uma falha no *host* em que roda o conjunto de componentes auxiliares ativos, o conjunto de componentes em estado de espera não conseguia receber a mensagem de *heartbeat*, porém o sistema esperava indefinidamente a realização da chamada do método remoto do Gerenciador do Sistema, não detectando assim a falha de *hardware*.

A solução para esse problema foi redefinir a classe que implementa a fábrica de objetos de conexão de sockets do Java RMI. Dessa forma, foram configuradas as características personalizadas dos sockets usados na comunicação do Java RMI, conforme a Figura 4, onde *timeout* é o tempo limite para esperar a resposta.

```
RMISocketFactory.setSocketFactory(
   new RMISocketFactory() {
      public Socket createSocket(String host, int port)
                                                  throws IOException {
         Socket socket = new Socket();
         socket.setSoTimeout(timeout);
         socket.setSoLinger(false, 0);
         socket.connect(new InetSocketAddress(host, port), timeout);

         return socket;
      }

      public ServerSocket createServerSocket(int port)
                                                  throws IOException {
         return new ServerSocket( port );
      }
   }
);
```

Figura 4 - Código Java para redefinir a fábrica de conexões do Java RMI

4.1.3. Realização de *Failover* dos Componentes

Failover é o processo de troca automática entre o sistema ativo e o sistema em espera na ocorrência de uma falha no sistema em espera (JAYASWAL, 2006).

No framework implementado a realização do *failover* entre os componentes que desempenham a mesma função é feita com a configuração do endereço IP que corresponde ao componente em questão. Essa configuração é possível com o uso de IP's virtuais.

Um IP virtual não está vinculado a nenhum *host* específico. Quando você se conecta a um endereço IP virtual, você não sabe a qual servidor você está conecta se conectando. O endereço IP permite a você se conectar a um serviço e receber respostas para todas as solicitações (JAYASWAL, 2006).

O componente em estado ativo é responsável por configurar o endereço IP virtual que está associado a seu papel no sistema. No caso de falha do componente quem realiza a ação de desassociar do mesmo o IP virtual correspondente é o Gerenciador do Sistema.

4.2. DISCUSSÃO DOS RESULTADOS

4.2.1. *Overhead* Devido à Utilização do *Framework*

Para analisar o *overhead* gerado pelo uso do *framework* desenvolvido, foi realizada uma análise de 100 requisições, enviadas 3 segundos após o recebimento da resposta, em uma aplicação que conta com 3 versões rodando em 3 hosts conectados na mesma local.

A captura do tempo foi realizada antes do envio da requisição e depois do recebimento da resposta para se determinar o tempo de processamento.

A aplicação realiza o processo de inversão de matrizes quadradas 10000 (a fim de proporcionar um tempo de processamento significativo) vezes, recebendo a matriz (de dimensão 10 neste experimento) a ser invertida e enviando o resultado para a votação.

O gráfico com os resultados é apresentado no Anexo 1.

A média de tempo do *overhead* das 100 requisições foi de 5,87%, gerando um tempo de execução médio de 4,545 segundos sendo o tempo de resposta para uma requisição feita diretamente a uma única versão é, em média, de 4,293 segundos.

Dessa forma observou-se que o uso do *framework* causou um impacto relativamente pequeno no tempo de execução da aplicação. Os demais fatores que poderiam influenciar no tempo de execução da aplicação – condição da rede, tempo de resposta da versão – exercem o mesmo impacto sobre uma aplicação de versão única que exerceriam sobre uma aplicação implementada em N versões.

4.2.2. Validação do Sistema com Injeção de Faltas

Para realizar a validação do uso do *framework*, foram elaborados experimentos com os tipos de falta que podem atingir um sistema distribuído. Para cada experimento, foi realizada a injeção da respectiva falta 10 vezes a fim de verificar se os componentes do *framework* se comportam da maneira esperada.

Para os experimentos, foram usadas mesmas 3 versões utilizadas para verificação do *overhead* com requisições enviadas 5 segundos após o recebimento da resposta da requisição. Os Gerenciadores do Sistema foram configurados para realizar o *heartbeat* a cada 1 segundo com tolerância de 3 mensagens de confirmação perdidas, totalizando 3 segundos para realização do *failover*.

Os experimentos são descritos nos próximos subitens.

4.2.2.1. Inserção de faltas de *software* em uma versão do sistema

Para validar a superação da falta de software promovida pela programação em N versões, foram inseridas faltas em uma das versões do software.

A primeira falta de *software* inserida foi na resposta provida pela versão, que passava a matriz sem realizar a inversão, caracterizando uma resposta incorreta. Outra falta inserida foi a finalização (simulando um *crash*) da versão após receber a requisição, causando omissão no processo da versão e o não recebimento por parte do Coletor de Resultados, que deve caracterizar um *timeout*. E por fim, foi inserida uma falta que provocava a alternância entre enviar e não enviar a requisição, causando o não recebimento de algumas respostas.

O resultado esperado é que, com o funcionamento correto das outras duas versões, a falta seja mascarada através do processo de votação realizado pelo sistema.

4.2.2.2. Finalização de processo dos componentes auxiliares

O objetivo desse experimento é validar a superação de faltas de omissão de processo dos componentes auxiliares do Gerenciador do Sistema. Para tal, durante a execução do sistema, os processos foram finalizados utilizando o PID (*Process Identifier*) do componente auxiliar através do *shell* (terminal) do sistema com o comando `kill -9 <PID>`.

Esperasse com esse experimento que o *host* secundário coloque em estado ativo respectivo componente auxiliar que foi finalizado, não causando falha no sistema.

4.2.2.3. Desligamento do host da eletricidade

O objetivo desse experimento é validar a superação de faltas de *hardware* proposto pelo *framework*. Dessa forma, durante a execução do sistema, o *host* primário foi desligado da fonte de alimentação, simulando assim uma falta de *hardware*.

Dessa forma, espera-se que o *host* secundário perceba essa falha e coloque em estado ativo todos os seus componentes para evitar a falha do sistema.

4.2.2.4. Travar *host*

Esse experimento visa validar a superação de faltas de *software* proposto pelo *framework*. Para simular a falta de software, no host primário foi executado um processo que levou o sistema a esgotar os seus recursos, levando-o ao travamento e caracterizando uma falha de *software*.

Esperasse que o *host* secundário identifique essa indisponibilidade do host primário e coloque seus componentes em estado ativo para evitar a falha do sistema.

4.2.2.5. Desligar *host* da rede

A fim de validar as falhas por problemas no canal de comunicação entre o host e a rede em que ele se conecta, durante a execução do sistema o host primário foi desconectado da rede, simulando assim uma falta na conexão do mesmo.

O esperado é que o host secundário identifica a falta de acessibilidade ao host primário e coloque seus componentes no estado ativo.

4.2.2.6. Desligar rede local

Quando se cria uma falta na rede local, se espera que ambos os Gerenciadores do Sistema identifiquem a falta do host adjacente e coloque os seus componentes auxiliares em estado ativo. Essa falha no sistema ocorre por só existir um meio de conexão entre os *hosts*.

4.2.3. Resultados da Validação

A seguir, no Quadro 6, será apresentado o resultado dos experimentos realizados.

Quadro 6 – Quadro de resultados dos experimentos realizados

Injeção de Falta Realizada	Comportamento Esperado	Comportamento Obtido
Inserção de falta de software em uma versão do sistema	Que o processo de votação possibilite o mascaramento da falta, impedindo o sistema de atingir um estado de falha.	A resposta obtida foi a esperada e não foi observado nenhum comportamento externo da falta por parte do cliente.
Finalização de processo dos componentes auxiliares	Que o Gerenciador do Sistema do *host* secundário coloque em estado ativo o respectivo componente auxiliar que falhou.	O Gerenciador do Sistema do *host* secundário identificou a falta do componente finalizado e colocou em estado ativo o respectivo componente auxiliar em todas as simulações e o sistema não chegou ao estado de falha.
Desligamento do host da eletricidade	Que o Gerenciador do Sistema do *host* secundário coloque em estado ativo todos os componentes auxiliares.	O Gerenciador do Sistema do *host* secundário identificou a falta do host desligado e colocou em estado ativo todos os seus componentes auxiliares em todas as simulações e o sistema não chegou ao estado de falha.

Travar host	Que o Gerenciador do Sistema do *host* secundário coloque em estado ativo todos os componentes auxiliares por ele monitorados.	O Gerenciador do Sistema do *host* secundário identificou a falta do *host* travado e colocou em estado ativo todos os seus componentes auxiliares em todas as simulações e o sistema não chegou ao estado de falha.
Desligar host da rede	Que o Gerenciador do Sistema do *host* secundário coloque em estado ativo todos os componentes auxiliares por ele monitorados.	O Gerenciador do Sistema do *host* secundário identificou a falha do *host* sem conexão à rede e colocou em estado ativo todos os seus componentes auxiliares em todas as simulações e o sistema não chegou ao estado de falha.
Desligar rede local	Que ambos os Gerenciadores de Sistema, ativo e secundário, interpretem a condição detectada como uma falha no *host* adjacente e coloquem em estado ativo os componentes auxiliares por eles monitorados.	A falta da rede local causou, não somente a identificação da falha do host adjacente, como tornou os componentes auxiliares inacessíveis ao seu respectivo Gerenciador do Sistema, caracterizando uma falha nos componentes.

4.3. Limitações

O *framework* conta com suporte para 2 conjuntos de componentes auxiliares e *failback* (volta do estado de falha para o estado ativo da instância primária de um componente) não automático.

Essas características poderiam ser estendidas para abranger um número maior de componentes auxiliares, se o sistema assim necessitasse.

Caso fosse um requisito do sistema, também a operação de *failback* poderia ser automatizada. Isto requereria porém considerável tempo adicional de desenvolvimento.

Outra limitação percebida no *framework* foi a incapacidade de resolver o problema conhecido como *split-brain* (quando as duas instâncias de um determinado componente se tornam ativas por causa de uma falta na rede de comunicação). Para solucionar este problema seria necessário se implementar uma rede de comunicação redundante, por exemplo, utilizando-se cabos de comunicação serial, ou se utilizar uma instância adicional do

Gerenciador do Sistema (executado em hardware diferente) que se comunicasse com as outras duas instâncias deste através de uma rede de comunicação alternativa e servisse como árbitro entre as outras duas instâncias para decidir qual delas deveria induzir um componente monitorado ao estado ativo. Essa implementação fica como trabalho futuro para o *framework*.

5. CONCLUSÃO

Esse trabalho apresentou uma análise de uma técnica de tolerância à faltas - a Programação em N Versões - e identificou os seguintes problemas em sua utilização:

- O votador como ponto único de falha;
- A necessidade de se tolerar faltas de hardware além das faltas de software em sistemas reais;
- Como estender a utilização desta técnica (Programação a N Versões) a outras aplicações (software) sem incorrer em gastos adicionais de desenvolvimento com a infraestrutura de apoio (votador, distribuidor de entradas e coletor de saídas);

Em seguida foi apresentada uma solução para os problemas identificados com a utilização de uma Estrutura Distribuída para o Padrão de Programação em N Versões. O *framework* desenvolvido nesta solução atingiu o seu objetivo de melhorar a confiabilidade de um software implementado pelo Padrão de Programação em N Versões (em relação à sua implementação em versão única) conforme os seguintes pontos:

- Adicionando ao sistema a capacidade de tolerar faltas tanto de *software* como de *hardware*;
- Eliminando o ponto único de falha do padrão de N versões original; e
- Provendo generalidade, através de componentes (de software) auxiliares reutilizáveis, para que aplicações distintas possam se beneficiar do *framework* sem um custo adicional de adaptação (além das N versões).

Os resultados obtidos na utilização do *framework* com uma aplicação de inversão de matrizes mostraram a capacidade deste de promover a tolerância à vários tipos de faltas testadas através de experimentos de injeção de faltas, atendendo assim à sua principal finalidade, melhorar a confiabilidade do sistema.

Bibliografia

ARMOUSH, A. **Design Patterns for Safety-Critical**. 197f. Tese de Doutorado - Faculdade de Matemática, Informárica e Ciências Naturais: [s.n.], 2010. 197 p.
COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Sistemas Distribuídos. Conceitos e Projeto**. Editora Bookman. Porto Alegre, RS, Brasil. 2007.
DOOLEY, K. **Designing Large Scale LANs**. O'Reilly Media. [S.l.], p. 404. 2001.
FERREIRA, L. L.; RUBIRA, C. M. F. **Reflective Design Patterns to Implement Fault Tolerance**. In: Proceedings of the OOPSLA'98 Workshop on Reflective programming in C++ and Java. Vancouver, CA. 1998.
GAMMA, E. et al. **Design Patterns - Elements of Reusable Object-Oriented Software**. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc, 1995.
GAWAND, H.; MUNDADA, R. S.; SWAMINATHAN, P. **Design Patterns to Implement Safety na Fault Tolerance**. International Journal of Computer Applications. [S.l.]. 2011.
GLINZ, M. **On Non-Functional Requirements**. University of Zurich. Switzerland, p. 6. 2007.
GROTTKE, M. et al. **Ten Fallacies of Availability and Reliability Analysis**. [S.l.], p. 20. 2008.
HAMEED, K.; WILLIANMS, R.; SMITH, K. **Separation of Fault Tolerance and Non-Functional Concerns: Aspect Oriented Patterns and Evaluatio**. J. Software Engineering & Applications. [S.l.]. 2010.
JAYASWAL, K. **Administering Data Centers**. Indianapolis, IN. 2006.
JOSHI, K. C. **Techniques of Software Fault Tolerance**. International Journal of Computer Science & Engineering Technology (IJCSET). [S.l.]. 2012.
KHOURY, R. E. A. **Diversity Through N-Version Programming: Current State, Challenges and Recommendations**. In: I.J. Information Technology and Computer Science. [S.l.]. 2012.
KIENZLE, J. **Software Fault Tolerance Implementing N-Version Programming**. McGill University. [S.l.]. 2012.
KOREN, I.; KRISHNA, C. M. **Fault-Tolerant Systems**. San Francisco, CA, USA: Morgan Kaufmann Publishers In, 2007.
LARMAN, C. **Utilizando UML e Padrões**. [S.l.]: Bookman, 2007.
MAGNETROL INTERNATIONAL, I. **Understanding Safety Integrity Level**. [S.l.]: [s.n.], 2012.
MYLOPOULOS, J.; CHUNG, L.; NIXON, B. Representing and Using Nonfunctional Requirements: A Process-Oriented Approach. **IEEE Transactions on Software Engineering - Special issue on knowledge representation and reasoning in software development**, Piscataway, NJ, USA, v. 18, p. 483-497, Junho 1992.
NATELLA, R.; ET AL. **On Fault Representativeness of Software Fault Injection**. IEEE Transactions on Software Engineering. [S.l.]. 2011.
ORACLE. Remote Objects. **Java SE Documentation**, -. Disponivel em: <http://docs.oracle.com/javase/jndi/tutorial/objects/storing/remote.html>. Acesso em: 29 out. 2013.
WEBER, T. S. **Tolerância a falhas: conceitos e exemplos**. Instituto de Informática - UFRGS. Rio Grande do Sul. 2001.
WEBER, T. S. **Um roteiro para exploração dos conceitos básicos de tolerância a falhas**. Instituto de Informática – UFRGS. Rio Grande do Sul, p. 62. 2002.

ANEXOS

Anexo I Overhead causado pelo uso do framework