UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS GRADUAÇÃO EM ENGENHARIA ELÉTRICA

IMPLEMENTAÇÃO DE BLOCOS DE RECUPERAÇÃO DISTRIBUÍDOS SEGUNDO UMA ABORDAGEM POR LINGUAGEM

DISSERTAÇÃO SUBMETIDA À UNIVERSIDADE FEDERAL DE SANTA CATARINA

PARA A OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA ELÉTRICA

VOLNEI VELLEDA RODRIGUES

FLORIANÓPOLIS, DEZEMBRO DE 1992.

IMPLEMENTAÇÃO DE BLOCOS DE RECUPERAÇÃO DISTRIBUÍDOS SEGUNDO UMA ABORDAGEM POR LINGUAGEM

VOLNEI VELLEDA RODRIGUES

ESTA DISSERTAÇÃO FOI JULGADA ADEQUADA PARA A OBTENÇÃO DO TÍTULO DE

MESTRE EM ENGENHARIA

ESPECIALIDADE **ENGENHARIA ELÉTRICA**, ÁREA DE CONCENTRAÇÃO **SISTEMAS DE CONTROLE E AUTOMAÇÃO INDUSTRIAL**, E APROVADA EM SUA FORMA FINAL PELO PROGRAMA DE PÓS-GRADUAÇÃO

Prof. Joni da Silva Fraga, Dr.
Orientador

Prof. Roberto de Souza Salgado, Ph. D.
Coordenador do Curso de Pós-Graduação em Engenharia Elétrica

BANCA EXAMINADORA

Prof. Joni da Silva Fraga, Dr.
Orientador

Prof. Jean-Marie Farines, Dr. Ing.

Prof. Marcelo Ricardo Stemmer, Dr. Ing.

Prof. Raul Fernando Weber, Dr.

À minha esposa Lídia Márcia

AGRADECIMENTOS

Agradeço a todos que, de alguma forma contribuíram para que este trabalho fosse realizado. Dentre estes, sou especialmente grato :

-À minha esposa Lídia, cujo carinho, compreensão e apoio são sempre fundamentais;

-Aos meus pais, Euto e Enylda, por tudo o que fizeram por mim e pelo constante exemplo de vida;

-Ao Professor Joni da Silva Fraga, por seus esforços na orientação deste trabalho;

-Aos amigos Eraldo Silveira e Silva e Luiz Nacamura Jr., por sua contribuição e incentivo;

-Aos bolsistas Alex Sander e Marcelo Maia, que colaboraram nas atividades de implementação;

-Aos amigos da UFSC e ETFSC, pelo estímulo e amizade;

-Aos meus familiares e amigos, pelo constante apoio;

-Aos amigos Marcelo, Paulo Renato e Maria Cristina, que iniciaram comigo a "aventura" que resultou neste trabalho.

SUMÁRIO

RESUMO

Neste trabalho é proposto um modelo para programação e gerenciamento de redundâncias em sistemas distribuídos, voltados a aplicações de tempo real. Podem ser identificados dois modelos : o modelo de tolerância a faltas, relacionado à detecção e recuperação de erros e o modelo de tratamento de faltas, associado à restituição de redundâncias perdidas durante a operação do sistema. Ambos os modelos foram incorporados à Linguagem de Implementação de Sistemas -LIS. A natureza desta linguagem incentiva a adoção de mecanismos de tolerância a faltas, particularmente aqueles que provêm do conceito de blocos de recuperação.

O modelo de tolerância a faltas proposto fundamenta-se na criação de um tipo módulo especial, o módulo tolerante a faltas, do qual são originadas duas instâncias. Estas instâncias (primária e secundária) diferem inicialmente em suas configurações internas e devem ser executadas em estações de processamento diferentes. Este modelo constitui uma implementação do modelo DRB (Bloco de Recuperação Distribuído), visando ampliar a abordagem proposta na literatura. A ocorrência de erros pode causar a troca de funções entre as instâncias e suas respectivas reconfigurações internas. Dois aspectos devem ser ressaltados: as características de tolerância a faltas são transparentes aos demais módulos do sistema e o modelo proposto é independente do hardware. O modelo de tratamento de faltas tem por objetivo básico a restauração do DRB, sempre que uma falta ou um conjunto de faltas venha a proporcionar uma degradação no seu comportamento.

ABSTRACT

This work presents a model for programming and management of redundancies in real-time distributed systems. In this proposal, two basic models can be identified : the fault tolerance model and the fault treatment model. The first one is related with error detection and error recovery aspects. The second one aims to restore lost redundancies, during the system operation. Both models were incorporated to a system implementation language - LIS. The characteristics of this language, such as modular decomposition and dynamic configuration, support the adoption of fault tolerance mechanisms.

The proposed fault tolerance model is based on a special module type, the fault-tolerant module. This module rises two instances : primary and secondary. These instances must be executed in different computers. The initial configurations of the instances are opposite. This approach establishes a DRB (Distributed Recovery Block) implementation and extends the proposal presented in literature. Error occurrences may cause functions to switch between instances. Besides, their internal configurations may be changed. Two aspects are relevant : fault tolerance characteristics are transparent for the other system modules, and the proposed model are hardware independent. The fault treatment model allows DRB to be restored when faults cause degeneration in its behavior.

CAPÍTULO 1

INTRODUÇÃO

A segurança de funcionamento de um sistema informático pode ser compreendida como a confiança que, de forma justificada puder ser depositada em seu funcionamento [Laprie, 85]. Há duas formas básicas de prover a segurança de funcionamento de um sistema : a prevenção de faltas e a tolerância a faltas. A prevenção a faltas consiste em evitar, através da utilização de metodologias adequadas, a ocorrência de faltas, durante a construção do sistema. A tolerância a faltas visa tornar o sistema imune às faltas, isto é, mesmo na presença destas, o sistema deve ser capaz de fornecer os serviços conforme sua especificação.

A obtenção de características de tolerância a faltas se dá pelo acréscimo no sistema de recursos adicionais, que podem ser de hardware, de software ou de tempo. Esses recursos adicionais, os quais não seriam necessários num sistema livre de faltas, são denominados redundâncias.

Podem ser apontadas duas técnicas clássicas utilizadas para a construção de sistemas tolerantes a faltas : Blocos de Recuperação [Randell, 75] e Programação a N-versões [Avizienis, 85]. A primeira trata particularmente com faltas de projeto (software), enquanto a segunda pode

também ser usada para faltas de hardware. Vários trabalhos têm sido desenvolvidos no sentido de aplicar essas técnicas a sistemas distribuídos de tempo real. De um modo geral, as restrições de tempo são incorporadas às técnicas a partir de mecanismos de "watch-dog" [Hecht, 76].

O conceito de Bloco de Recuperação Distribuído (DRB) foi apresentado em [Kim, 84] como uma abordagem uniforme para o tratamento de faltas de hardware e de software em aplicações de tempo real. O DRB utiliza características de redundâncias ativas e passivas para obter uma rápida recuperação no caso de falhas parciais do sistema. As implementações de DRB que têm sido propostas [Kim, 88] estão associadas a uma estrutura particular de hardware, fortemente acoplada, onde dois elementos processadores comunicam-se através de compartilhamento de memória em uma única estação.

O presente trabalho visa ampliar esta abordagem, a partir do desenvolvimento de ferramentas que simplifiquem a programação de redundâncias, tornando o modelo independente do hardware, o que possibilitaria sua extensão a sistemas distribuídos fracamente acoplados. A implementação de DRB proposta neste trabalho está fundamentada na criação de um tipo na Linguagem de Implementação de Sistemas - LIS [Fraga, 89], o Tipo Módulo Tolerante a Faltas. O tipo módulo FT originará duas instâncias (primária e secundária), as quais diferem em suas configurações internas e devem ser executadas em estações diferentes. A ocorrência de erros pode causar a troca de funções entre as instâncias no modelo, bem como as suas reconfigurações internas.

As extensões da linguagem LIS, que dão suporte ao DRB, reduzem os esforços do programador na programação de redundâncias. Ao programador caberão apenas as atribuições que lhe são inerentes: programação dos algoritmos de aplicação e do teste de aceitação, com os limites de tempo associados (mecanismos de "timeout").

Do ponto de vista dos outros módulos de aplicação, há uma completa transparência dos módulos tolerantes a faltas. Desta maneira, o processo de configuração permanece independente da fase de programação dos módulos, conforme recomenda o estilo de programação da linguagem LIS. Este estilo de programação permite uma configuração flexível, que possibilita a restauração do DRB, durante a operação do sistema, quando da ocorrência de faltas permanentes. O modelo proposto não requer nenhuma característica especial de hardware, para implementar o DRB.

A realização deste trabalho teve como ponto de partida a experiência adquirida pelo grupo de pesquisas em sistemas distribuídos do Laboratório de Controle e Microinformática (LCMI -UFSC) no desenvolvimento de ferramentas de software. Em particular, o ambiente ADES - Ambiente de Desenvolvimento e Execução de Software - [Fraga, 89] serviu de suporte à implementação dos modelos propostos. Esse ambiente foi desenvolvido visando a construção de software distribuído para aplicações de tempo real, segundo uma abordagem de programação distribuída centrada em uma linguagem de implementação de sistemas (Linguagem LIS).

O restante desta dissertação está organizada da forma descrita a seguir. No capítulo 2 são apresentados a terminologia e os conceitos básicos

em segurança de funcionamento e tolerância a faltas. Nesse capítulo, são também apresentados e discutidos alguns dos modelos mais representativos destacados na literatura, que visam implementar a tolerância a faltas de projeto, em especial a sistemas distribuídos de tempo real. O capítulo 3 descreve os modelos de tolerância a faltas e de tratamento de faltas propostos, os quais implementam o conceito de DRB, segundo uma abordagem por linguagem. O capítulo 4 apresenta os detalhes da implementação desses modelos na linguagem LIS e discute os principais resultados obtidos. Finalmente, no capítulo 5 são apresentadas as conclusões finais deste trabalho.

CAPÍTULO 2

TOLERÂNCIA A FALTAS

2.1-Introdução

Este capitulo tem por objetivo apresentar a terminologia básica utilizada em segurança de funcionamento e, mais particularmente, em tolerância a faltas. A terminologia apresentada baseia-se, essencialmente, em [Laprie, 89] e [Powell, 90].

Neste capítulo, são também discutidos os aspectos gerais relacionados à tolerância a faltas, em especial no que se refere a sistemas distribuídos para aplicações em tempo real. Além disto, são apresentados os principais modelos propostos na literatura visando implementar a tolerância a faltas a sistemas distribuídos.

2.2-Segurança de Funcionamento : conceitos básicos

Um *sistema* consiste de um grupo de *componentes* configurados de maneira a interagirem entre si. Os conceitos de sistema e componente são

recursivos, na medida em que os componentes também constituem sistemas. O ponto de parada da recursividade é aquele onde o sistema pode ser considerado atômico, o que depende do grau de abstração desejado. O *serviço* fornecido por um sistema expressa o seu comportamento, tal como é percebido por sistemas externos (usuários do primeiro).

A *Segurança de Funcionamento* de um sistema computacional é uma propriedade que está relacionada com a qualidade do serviço oferecido, permitindo que, justificadamente, possa ser depositada confiança nesse serviço [Laprie, 85].

Obstruções à Segurança de Funcionamento

A segurança de funcionamento tem como obstruções as circunstâncias indesejáveis *falta*, *erro* e *falha* (em correspondência aos termos em inglês *fault*, *error* e *failure*, respectivamente), entre as quais há uma relação de causa e efeito :

- Uma falha ocorre quando o serviço entregue pelo sistema difere do estabelecido na sua especificação. Como a falha manifesta-se a nível de serviço, pode ser percebida e avaliada pelo usuário. A existência de uma falha é decorrência de um erro.
- Um erro é parte do estado interno do sistema, podendo, portanto, ser detectado. A existência de um erro deve-se a uma falta.
- Uma falta pode ser definida como a causa física ou algorítmica de um erro. Uma falta é dita dormente ou inativa, até o momento em que dá origem a um estado errôneo.

Sintetizando, um erro é a manifestação, a nível de sistema de uma falta e uma falha é a manifestação, a nível de serviço, de um erro. Deve ser observada a recursividade dos conceitos de falta, erro e falha, à medida em que são analisados diferentes níveis de abstração do sistema. Desta forma, uma falta a nível de sistema corresponde a uma falha a nível de componente.

Classes de Faltas e Semântica de Falhas

Podem ser identificados três critérios básicos para a classificação das faltas : em relação à persistência, causas e efeitos.

Em termos de sua persistência, uma falta pode ser temporária (transiente) ou permanente. Em termos de suas prováveis causas, uma falta pode ser, basicamente, classificada em : física (decorrente de fenômenos físicos adversos, sejam eles internos ou externos) ou humana (faltas de projeto ou de interação). Uma classificação mais abrangente, segundo este critério, pode ser encontrada em [Laprie, 90].

Em termos de seus efeitos, isto é, dos erros que pode vir a ocasionar, uma falta pode ser classificada como : falta por valor, falta de temporização, ou ainda, falta arbitrária [Powell, 90]. Uma falta por valor está associada à entrega, por parte de um ou mais componentes do sistema, de um serviço fora da faixa de valores especificados para o mesmo. Uma falta de temporização está associada à entrega de valores corretos, fora dos limites de tempo especificados. Um caso extremo destas faltas são as faltas de omissão, em decorrência das quais o serviço efetivamente não é entregue. A persistência da omissão de um serviço caracteriza uma falta de "crash".

Uma falta arbitrária engloba erros tanto de valor como de temporização. Duas classes de faltas arbitrárias podem ser identificadas : faltas maliciosas e faltas de improvisações. Faltas maliciosas estão associadas à entrega de um serviço fora do conjunto de valores especificados, em instantes aleatórios. Faltas de improvisações estão associadas à entrega espontânea de um item de serviço que não é esperado.

A classificação das faltas segundo seus efeitos permite determinar a semântica de falhas de um sistema. A semântica de falhas define as maneiras pelas quais o sistema pode falhar. O projeto de sistemas tolerantes a faltas deve estar fundamentado num conjunto de hipóteses de tipos de faltas, que definem a semântica de falhas prevista nas especificações de serviço deste sistema.

Obtenção e Validação da Segurança de Funcionamento

O desenvolvimento de um sistema seguro envolve a utilização conjunta de métodos de obtenção e de validação da segurança de funcionamento.

Obter a segurança de funcionamento significa dar ao sistema a capacidade de fornecer o serviço conforme especificação.

métodos de obtenção :

-prevenção a faltas : consiste em prevenir, por construção, a ocorrência de faltas.

-tolerância a faltas : visa capacitar o sistema a oferecer o serviço especificado, embora na presença de faltas, pelo uso de redundâncias.

Validar a segurança de funcionamento significa permitir que se tenha confiança na capacidade do sistema oferecer o serviço especificado.

métodos de validação :

-remoção de faltas : tem por objetivo minimizar a presença de faltas, através de verificação.

-previsão de faltas : consiste em estimar, mediante avaliação, a presença, a criação e as conseqüências de faltas.

Medidas da Segurança de Funcionamento

Há várias medidas visando expressar, em termos quantitativos a segurança de funcionamento. As duas medidas principais são a *confiabilidade* e a *disponibilidade* [Siewiorek, 84].

A confiabilidade é uma medida da continuidade do sistema executando o serviço conforme especificado, a partir de um instante inicial. A medida de tempo mais utilizada como indicador da confiabilidade é o tempo médio entre falhas (MTTF ou MTBF).

A disponibilidade é uma medida da probabilidade do sistema fornecer o serviço conforme especificado, considerando a alternância entre serviço dentro da especificação e serviço fora da especificação.

2.3-Tolerância a Faltas : aspectos gerais

A tolerância a faltas é uma propriedade que possibilita ao sistema fornecer o serviço de acordo com a especificação, mesmo na presença de faltas. As técnicas de tolerância a faltas estão fundamentadas na incorporação de redundâncias ao sistema. Redundâncias constituem recursos adicionais de hardware, de software ou de tempo, os quais não seriam necessários num sistema livre de faltas.

As redundâncias podem ser classificadas em ativas e passivas. O uso de redundâncias ativas é caracterizado pelo fato que as unidades replicadas têm um papel ativo durante o processamento. Já uma redundância passiva atua como simples unidade reserva.

As técnicas de tolerância a faltas são constituídas de duas fases principais: processamento de erros e tratamento de faltas [Laprie, 90].

2.3.1-Processamento de Erros

O processamento de erros tem por objetivo remover os erros do estado do sistema, visando evitar que venham a ocasionar falhas. Duas estratégias podem ser identificadas : recuperação de erros e compensação de erros. Em ambos os casos, a etapa inicial compreende a detecção de erros.

A detecção de erros é, normalmente, o ponto de partida de uma técnica de tolerância a faltas, uma vez que o sistema é incapaz de detectar uma falta diretamente, mas sim sua conseqüência, que é o erro.

A detecção de erros deve ser efetiva o suficiente para evitar a propagação de erros através do sistema. Além disto, a técnica de detecção utilizada deve basear-se apenas na especificação do sistema e ser o mais independente possível do mesmo. Desta forma, visa-se eliminar a possibilidade de que uma mesma falta venha a afetar tanto ao sistema quanto ao mecanismo de detecção de erros.

(a)Recuperação de Erros

A recuperação de erros consiste em substituir um estado errôneo do sistema por um estado consistente e isento de erros, a partir do qual o serviço poderá continuar sendo entregue de acordo com a especificação. Duas formas de recuperação de erros são identificadas na literatura : recuperação em retrocesso e recuperação em avanço.

Recuperação de Erros em Retrocesso

A recuperação de erros em retrocesso requer que informações acerca do estado do sistema sejam regularmente armazenadas em pontos de recuperação. Deste modo, quando um erro é detectado, um estado anterior do sistema (livre de erros) é restaurado, a partir das informações contidas no último ponto de recuperação estabelecido.

A obtenção dos pontos de recuperação pode se dar segundo vários critérios: armazenamento do estado completo do sistema; armazenamento de parte do estado do sistema (apenas a parcela do estado efetivamente modificada posteriormente ao estabelecimento do ponto de recuperação); armazenamento das operações processadas a partir do ponto de recuperação (permitindo que, caso seja necessário, tais operações possam ser desfeitas).

A recuperação de erros em retrocesso é um conceito bastante geral, aplicável a qualquer sistema, visto não exigir conhecimento da aplicação, e a qualquer tipo de erro. Os custos associados relacionam-se ao armazenamento de informações nos pontos de recuperação e à perda de processamento no caso de retorno. O compromisso entre esses dois aspectos deve ser observado no estabelecimento de pontos de recuperação, especialmente no caso de processos concorrentes e/ou aplicações de tempo real.

Recuperação de Erros em Avanço

A recuperação de erros em avanço utiliza o estado errôneo em que se encontra o sistema para levá-lo a um estado livre de erros. Desta forma, é necessário que haja um perfeito conhecimento da aplicação, de modo a permitir uma compreensão antecipada das possíveis faltas e suas conseqüências.

A recuperação em avanço deve ser projetada como parte integrante da aplicação, sendo inapropriada para a recuperação de erros oriundos de faltas não antecipadamente previstas. Entretanto, não apresenta os custos que estão associados à recuperação em retrocesso.

As duas principais técnicas de recuperação em avanço são : tratamento de exceções e compensação. Um tratador de exceções consiste num programa que é acionado a partir da sinalização de uma condição de exceção ocorrida em tempo de execução, objetivando levar o sistema a um estado consistente. A compensação consiste no envio pelo sistema, ao ambiente com o qual se comunica, de informação suplementar, visando corrigir os efeitos de uma informação errônea anteriormente emitida.

Cumpre ressaltar que recuperação em retrocesso e recuperação em avanço não constituem abordagens excludentes, mas sim complementares, de modo que seu uso conjunto num mesmo sistema pode melhorar o desempenho e a segurança do mesmo.

(b)Compensação de Erros

Esta estratégia visa fornecer o serviço conforme especificado, mesmo na presença de erros no sistema. Os efeitos dos erros são escondidos pela ação das redundâncias (mascaramento de erros). As técnicas de compensação de erros são baseadas em redundâncias ativas. Tendo em vista o posterior tratamento da falta, a detecção de erros também se faz necessária.

2.3.2-Tratamento de Faltas

O tratamento de faltas tem por objetivo eliminar faltas do sistema, visando evitar que voltem a se manifestar. Para tanto, faz-se necessário identificar os elementos faltosos, de modo a recuperá-los ou retirá-los do sistema. Podem ser identificadas duas fases no tratamento de faltas : diagnose das faltas e reparo do sistema.

A diagnose consiste em localizar e identificar a falta. Um sistema tolerante a faltas deve ser estruturado de modo a evitar a propagação de erros. Esta estruturação ocorre através de procedimentos como utilização de ações atômicas, definição e monitoração criteriosa das interfaces entre componentes e estruturação do sistema em módulos e camadas, por exemplo. O confinamento do erro a uma certa região do sistema favorece a localização da falta que lhe deu origem.

A primeira etapa do reparo do sistema consiste na adoção de procedimentos visando impedir a reativação da falta (técnicas de passivação de faltas). Na etapa seguinte, caso seja necessário, o sistema será reconfigurado, de forma manual ou automática, de modo a retornar ao funcionamento normal.

2.4-Tolerância a Faltas de Software

Inicialmente, o desenvolvimento de sistemas computacionais tolerantes a faltas atribuiu uma maior ênfase às faltas de hardware. Com o aumento no tamanho e na complexidade dos programas, a tolerância a faltas de software passou a ser um aspecto tão ou mais relevante que a tolerância a faltas de hardware.

Um programa não sofre desgastes físicos ou deteriora-se com o decorrer do tempo. Desta forma, uma vez tenha sido executado conforme especificado, deverá fazê-lo sempre, sob as mesmas condições de entrada, de ambiente ou de exigências do usuário. Contudo, na ocorrência de mudanças em qualquer destas condições, o programa poderá falhar.

O software tem como maior fonte de faltas o próprio projeto. As faltas de projeto poderão ser geradas em qualquer das fases que envolvem a concepção do software. O uso de metodologias adequadas para a produção de software, bem como os testes e verificações, reduzem consideravelmente a incidência de faltas de projeto. As faltas que persistirem, já que eliminá-las totalmente é praticamente impossível, serão faltas residuais e, provavelmente, manifestar-se-ão apenas em circunstâncias excepcionais.

Tendo em vista estruturar um sistema de software, dotando-o de meios de tolerância às suas próprias faltas de projeto, várias técnicas e modelos têm sido propostos. A maioria destes deriva dos conceitos de Blocos de Recuperação [Randell, 75] e Programação a N-versões [Avizienis, 85]. O

primeiro utiliza redundâncias passivas e fundamenta-se na detecção e recuperação de erros, enquanto o segundo utiliza redundâncias ativas e compensação de erros.

2.4.1-Programação a N-versões

A técnica da programação a N-versões [Avizienis, 84], [Avizienis, 85] consiste na geração independente de N (N >= 2) módulos de software (versões) funcionalmente equivalentes, a partir da mesma especificação inicial. Todas as versões são executadas e seus resultados são comparados, através de um processo de votação. Deste modo, há o mascaramento dos possíveis erros.

As N-versões devem ser projetadas da maneira mais independente possível. O projeto de cada versão deve estar a cargo, preferencialmente, de diferentes grupos de programadores, os quais não devem interagir. O elo comum entre eles deve ser apenas a especificação inicial. Desta forma, visa-se reduzir a possibilidade de que as versões tenham faltas de modo comum. A ausência de faltas de projeto comuns às versões é condição essencial para o sucesso da técnica.

A especificação inicial visa determinar, de forma completa e sem ambigüidades, os requisitos funcionais necessários, a partir dos quais as versões deverão ser concebidas, permitindo a maior liberdade possível, em termos de implementação das mesmas.

Execução das Versões

O controle da execução das N-versões é realizado através de um programa denominado ordenador ("driver"). O ordenador deve executar as seguintes funções : chamar cada versão, esperar que todas as versões completem a sua execução, comparar os resultados fornecidos pelas versões e agir sobre estes. É necessário que exista sincronização entre o ordenador e as versões. As versões esperam pela sinalização do ordenador para começarem sua execução e o ordenador, por sua vez, aguarda que as versões concluam o seu processamento e enviem suas respostas. Isto permite diferentes tempos de execução para as versões, de modo que se faz necessário que alguma condição de tempo máximo de espera ("timeout") seja associada ao mecanismo de sincronização, para evitar esperas excessivamente longas ou infinitas.

As versões devem enviar seus resultados para o ordenador num mesmo formato, para a realização do processo de votação. Uma outra exigência da técnica é que cada versão seja executada de maneira atômica, com relação às demais. É necessário também que, ao serem executadas, todas as versões tenham acesso aos mesmos dados de entrada.

Processo de Votação

A comparação dos resultados obtidos consiste no aspecto mais complexo da técnica das N-versões. Não existe um esquema de votação geral, que possa ser totalmente independente da aplicação e empregado em todas as situações. Quando se trata da manipulação de caracteres ou de aritmética inteira, por exemplo, isto é, nos casos onde pode ser esperado que um mesmo resultado seja fornecido pelas versões atuando sem faltas, a

comparação é bastante simples, consistindo de uma verificação de igualdade, na qual prevalece o resultado majoritário (votação exata). Nos casos onde tais características não estão presentes, o processo de comparação é mais complexo. Nesses casos, os resultados corretos esperados das versões não serão necessariamente iguais, mas a variação entre eles deverá estar situada numa certa faixa de valores ou não desviar significativamente de valores médios, por exemplo.

O sucesso da técnica de programação a N-versões fundamenta-se no fato de que os resultados majoritários sejam corretos. Se isto não ocorrer, a técnica em si não poderá impedir a ocorrência de erros, necessitando que estejam presentes outros meios para que sejam evitadas falhas. Em termos de desempenho, esta técnica não traz a perspectiva de atrasos e de perda de processamento, que estão associados a eventuais retornos nas técnicas que utilizam recuperação de erros. Contudo, seu desempenho é ditado pela versão mais lenta. Quanto à utilização de recursos, é pressuposta a existência de hardware suficiente para que as versões sejam executadas paralela e independentemente.

2.4.2-Blocos de Recuperação

A estrutura de um bloco de recuperação [Horning, 74], [Randell, 75], [Lee, 78a] consiste de um módulo primário, um ou mais módulos alternativos e um teste de aceitação comum a todos os módulos. A execução de um bloco de recuperação processa-se da seguinte forma :

(a)inicialmente, é estabelecido um ponto de recuperação;

(b)o modulo primário é executado;

(c)se nenhum erro em tempo de execução é detectado pelo suporte, inclusive violação a restrições de tempo, e os resultados fornecidos pelo módulo satisfazem ao teste de aceitação, a execução do bloco é encerrada (considerada correta);

(d)caso contrário, o estado do sistema deve ser restaurado, a partir do ponto de recuperação estabelecido (recuperação em retrocesso) e o módulo alternativo seguinte deve ser executado;

(e)a execução do bloco de recuperação prosseguirá até que um dos módulos possa ser considerado correto (conforme (c)) ou se esgotem as alternativas. Neste último caso, considera-se que o bloco de recuperação falhou e uma condição de exceção deve ser sinalizada.

Projeto dos Módulos

A técnica dos blocos de recuperação pode ser utilizada de duas diferentes formas :

-todos os módulos deverão apresentar exatamente os mesmos resultados;

-os módulos não devem apresentar necessariamente os mesmos resultados, mas resultados similares aceitáveis. Por exemplo, o módulo primário tentará fornecer os resultados desejados, enquanto os módulos alternativos tentarão produzir resultados progressivamente degradados, com relação aos resultados desejados, mas ainda aceitáveis.

No primeiro caso, os módulos devem ser projetados da forma mais independente possível, a partir de uma especificação inicial, visando evitar a ocorrência de faltas de projeto comuns. Para isto, podem ser requisitados programadores ou grupos de programadores diferentes, os quais não devem interagir.

No segundo caso, os projetos dos módulos são naturalmente diferentes, de modo que a ocorrência de faltas de projeto comuns é pouco provável. À medida em que o serviço oferecido afasta-se do desejado inicialmente (módulo primário), os algoritmos tornam-se mais simples e, provavelmente, menos sujeitos a faltas de projeto.

Em ambos os casos, deve ser ressaltado que, para o restante do programa, é indiferente qual dos módulos efetivamente forneceu os resultados. Um módulo alternativo não tem conhecimento das razões que levaram seu antecessor no bloco a falhar, nem tem acesso a quaisquer resultados intermediários obtidos pelo primeiro.

Para a implementação da técnica dos blocos de recuperação são necessários mecanismos para promover a passagem do controle entre os módulos e realizar a recuperação em retrocesso. Esses mecanismos devem ser fornecidos pelo suporte de execução, de forma transparente ao usuário. Ao iniciar sua execução, cada módulo deve ter acesso ao mesmo estado do sistema a que tiveram seus antecessores no bloco de recuperação.

Teste de Aceitação

O teste de aceitação constitui-se no aspecto mais crítico da técnica dos blocos de recuperação, cujo sucesso depende, em grande parte, da formulação de um teste eficiente. Idealmente, o teste deveria verificar a completa correção dos resultados fornecidos. Entretanto, esse tipo de teste nem sempre pode ser elaborado. Normalmente, os testes de aceitação não garantem que os resultados estejam absolutamente corretos, mas sim num nível aceitável.

O grau de rigor do teste de aceitação é uma decisão do projetista. É desejável que o teste seja o mais completo possível. Entretanto, deve ser considerado que, quanto mais completo for o teste, maior será a sua complexidade, o que o torna mais susceptível a faltas de projeto. Além disto, seu desempenho, em termos de tempo e utilização de recursos, poderá ser comprometido.

O objetivo do teste de aceitação é garantir que os resultados fornecidos pelo bloco de recuperação sejam aceitáveis para o restante do sistema. Por esta razão, o teste não deve ter acesso às variáveis locais dos módulos, mas apenas às variáveis globais do bloco. Normalmente, as variáveis globais poderão ser acessadas pelo teste de aceitação tanto em seus valores após a execução do módulo em questão, quanto em seus valores imediatamente anteriores ao início da execução do bloco.

O projeto do teste de aceitação é uma questão bastante complexa [Anderson, 81]. Primeiramente, não há uma metodologia geral a ser seguida

para a escolha do tipo de teste a ser adotado, bem como a melhor maneira de projetá-lo. O teste de aceitação deve ser projetado de maneira independente dos módulos, visando evitar faltas de projeto comuns. Espera-se que o teste seja bem menos complexo que os módulos, portanto menos sujeito a faltas residuais de projeto. No caso em que as alternativas fornecem um serviço progressivamente degradado, o teste de aceitação poderá ser, no máximo, tão rigoroso quanto para verificar a correção dos resultados do último módulo alternativo.

Um teste de aceitação pode ser desenvolvido em dois níveis: testes funcionais (alto nível) e testes estruturais (baixo nível) [Hecht, 86]. Os testes funcionais testam se as saídas do programa são consistentes com os requisitos funcionais, enquanto os testes estruturais testam seções de código, verificando a correção de funções e variáveis relevantes. A utilização de testes funcionais é sempre necessária. A adoção conjunta de testes estruturais possibilita a observação de comportamentos inesperados nos módulos, mesmo quando a degradação decorrente é sutil.

Em termos de desempenho, o tempo gasto no estabelecimento e descarte de pontos de recuperação, bem como na execução do teste de aceitação pode trazer atrasos consideráveis, especialmente se a operação de retorno efetivamente ocorrer, ocasionando a perda de todo o processamento realizado pelo módulo que falhou. Entretanto, pode se dizer que o tempo de execução de um bloco de recuperação é, potencialmente, ditado pelo módulo primário (supostamente o de melhor desempenho).

O problema do desempenho da técnica dos blocos de recuperação pode ser crítico para aplicações de tempo real. Além disto, a recuperação em retrocesso deve ser criteriosamente estabelecida quando se tem processos concorrentes. Isto faz com que sejam necessárias características adicionais ou algumas adaptações à técnica proposta originalmente, para que esta possa ser efetivamente eficiente em tais casos.

Em termos de utilização de recursos de processamento, deve ser notado que apenas um módulo é executado de cada vez, o que permite uma eficiente execução do bloco, mesmo em sistemas monoprocessadores.

2.4.3-Recuperação de Erros em Sistemas Concorrentes

Quando se trata de sistemas que possuem processos concorrentes é necessário considerar a influência que os procedimentos de tolerância a faltas de cada processo, especialmente no tocante à recuperação de erros, poderão vir a exercer sobre os demais. Os processos concorrentes podem ser classificados em: *independentes*, *competindo* ou *cooperando* [Anderson, 81].

Os processos são *independentes* quando não há qualquer interação entre os conjuntos de objetos a que cada processo tem acesso, de modo que a execução de cada processo é completamente independente dos demais. Desta forma, a recuperação de erros dá-se da mesma forma que em processos seqüenciais simples.

Quando processos concorrentes estão *competindo*, considera-se que há o compartilhamento de alguns recursos, mas não há troca de informações entre os processos. Uma vez que não há fluxo de informações entre os processos, estes podem ser vistos como independentes, desde que seja garantida a exclusão mútua na utilização dos recursos compartilhados. Isto leva a procedimentos de recuperação de erros similares aos utilizados para processos independentes.

Processos concorrentes são ditos *cooperando* quando existe um fluxo de informações entre eles. A recuperação de erros, nesse caso, torna-se mais complexa. No caso da recuperação em retrocesso, o estabelecimento de pontos de recuperação considerando individualmente os processos que cooperam, isto é, sem levar em conta as interdependências geradas pelas trocas de informações, pode resultar numa grande perda de processamento, causada pelo retorno generalizado e desordenado dos processos, o que constitui o chamado *efeito dominó* [Randell, 75].

As estratégias para a recuperação em retrocesso no caso de processos cooperando têm por objetivo obter um conjunto de pontos de recuperação consistente. Este conjunto deve permitir, no caso de detecção de erro em um processo, o retorno eficiente do processo em questão e de todos aqueles que poderiam ter sido afetados por ele, devido à troca de informações. Um conjunto de pontos de recuperação eficiente pode ser definido da seguinte forma :

- Sendo um subconjunto de processos P1,...,Pn, cujos pontos de recuperação mais próximos a um certo instante t são estabelecidos nos

instantes t1,...,tn, respectivamente, o conjunto de pontos de recuperação será consistente, nesse instante, se:

(a)Para quaisquer processos Pi e Pj, estes não se comunicam entre os instantes ti e tj;

(b)no período entre ti e t, o processo pi não se comunica com nenhum processo externo ao subconjunto.

Um conjunto consistente de pontos de recuperação ativos forma uma linha de recuperação [Lee, 78]. A cada processo, em qualquer instante, está associada uma seqüência de linhas de recuperação, cada uma das quais é um conjunto consistente de pontos de recuperação. Cada um desses pontos de recuperação pertence a um processo diferente, entre os quais está o processo em questão. A seqüência de linhas de recuperação associada a um processo pode ser vazia, em determinado instante. A obtenção de linhas de recuperação pode seguir duas abordagens: linhas pré-planejadas ou não.

Conversações

Uma *conversação* [Randell, 75] consiste numa extensão do conceito de blocos de recuperação a dois ou mais processos, mediante a imposição de restrições à comunicação entre processos, de forma a permitir uma recuperação de erros mais eficiente.

A entrada dos processos em uma conversação dá-se de forma assíncrona. Processos que possuem meios de se comunicar podem, a qualquer momento, estabelecer seus pontos de recuperação e entrar numa conversação. No interior da conversação é permitida a livre comunicação

entre os processos, sendo proibida a comunicação com processos externos à mesma.

A saída de uma conversação é realizada de forma síncrona. Ao final da conversação, os processos participantes deverão satisfazer seus respectivos testes de aceitação e aguardar que os demais façam o mesmo. Se algum dos processos não for aprovado em seu teste de aceitação, todos os processos participantes deverão restaurar seus estados, a partir dos pontos de recuperação estabelecidos e reiniciar a execução, utilizando módulos alternativos.

Conversações são vistas como operações atômicas e podem ser aninhadas de modo que os processos pertencentes a uma conversação interna constituam um subconjunto dos processos da conversação externa. Podem ser apontados como problemas a redução do paralelismo, ocasionada pela necessidade de saída síncrona, e a possibilidade de ocorrerem esperas indefinidas, por exemplo, quando um processo que é aguardado não consegue entrar na conversação ou chegar ao teste de aceitação (processos desertores). Uma forma mais restritiva de conversação, constitui-se no conceito de troca (exchange), onde os processos entram assim que são inicializados e saem quando todos tiverem terminado sua execução [Anderson, 83].

A partir da idéia inicial de conversação, várias implementações foram propostas, partindo do esquema básico e, eventualmente, introduzindo algumas adaptações. Algumas dessas propostas serão apresentadas a seguir.

Técnica de Russell

Esta técnica [Russell, 80] considera que um sistema assíncrono é constituído por um grupo finito de processos, cuja interação se dá através de um número finito de listas de mensagens, utilizando primitivas do tipo *send* e *receive*, de modo que não há dados globais ou compartilhados.

Sendo *m* uma lista de mensagens e *x* uma variável do mesmo tipo, a operação *SEND (m ,x)* coloca o valor de *x* na lista de mensagens *m*, enquanto *RECEIVE (m, x)* retira uma mensagem da lista *m*, colocando-a na variável *x*. A recepção de mensagens obedece à mesma ordem de envio. O estado do sistema será consistente se, para cada uma das listas de mensagens, a seqüência de mensagens recebidas preceder à seqüência de mensagens emitidas pela lista.

A detecção e recuperação de erros são realizadas para cada processo em separado. A cada processo está associada uma memória local que armazena os dados de recuperação (todo o estado do sistema ou apenas a parcela modificada após um ponto de recuperação) e registra as operações realizadas com listas de mensagens. A cada lista de mensagens está associado um registro das mensagens recebidas e enviadas.

As ações de recuperação de erros fundamentam-se na utilização dos comandos *mark*, *restore* e *purge*. O comando *mark* é usado para estabelecer pontos de recuperação. Os pontos de recuperação dividem a memória local em

regiões, de modo que a cada ponto de recuperação está associado um conjunto de dados de recuperação.

O comando *restore* é usado para realizar a restauração de estado, levando o sistema a um estado consistente e livre de erros. A execução de um comando *RESTORE (x)* faz com que o processo em questão restaure seu estado *x*, a partir das informações contidas na memória local, associadas ao ponto de recuperação correspondente. Assim, as operações realizadas são desfeitas, as mensagens recebidas são desconsideradas e as mensagens enviadas são revogadas, isto é, retiradas da lista de mensagens correspondente.

Esse retorno de um processo poderá levar o sistema a um estado inconsistente, devido às interações com outros processos. Processos que já receberam mensagens revogadas deverão restaurar seus estados, enquanto processos que haviam enviado tais mensagens não precisarão fazê-lo, já que as mesmas estão armazenadas nas listas correspondentes. Os retornos prosseguirão até que o sistema atinja um estado consistente, a partir do qual poderá retomar sua operação normal. O comando *purge* é utilizado para auxiliar a liberação de informações de recuperação antigas.

Técnica de Gregory-Knight

A técnica de Gregory-Knight [Knight, 85] propõe algumas modificações ao conceito básico de conversações, sendo fundamentada na introdução das construções linguísticas *diálogo* e *colóquio*.

Um *diálogo* é definido como uma ocorrência onde um conjunto de processos estabelecem pontos de recuperação individuais e comunicam-se entre si (e com nenhum outro processo não participante do diálogo), determinando se todos devem descartar seus pontos de recuperação (sucesso do diálogo) ou restaurar seus estados (falha do diálogo).

Diálogos podem ser adequadamente aninhados, de modo que o conjunto de processos participantes do *diálogo* interno deve ser um subconjunto dos participantes do *diálogo* externo. Ao ingressar num *diálogo*, cada processo tem algum objetivo (objetivo local) e ao *diálogo* em si pode estar associado um objetivo global.

Quando um *diálogo* falhar, os processos participantes deverão restaurar seus estados, a partir dos pontos de recuperação estabelecidos e utilizar seus algoritmos alternativos, para tentar realizar o mesmo objetivo local ou um objetivo modificado. Para tanto, o processo deverá entrar em outro *diálogo*, possivelmente envolvendo um grupo de participantes diferente do anterior. O *colóquio* permite expressar essa possibilidade de restauração de estado e ocorrência de diálogos alternativos.

Propostas de Kim

Em [Kim, 82] são propostas quatro maneiras de implementar conversações, partindo de extensões à linguagem Pascal Concorrente. Nessas implementações, a comunicação entre os processos é realizada através de monitores. Ao final da conversação, os processos participantes deverão se

submeter a um teste de aceitação global (teste de aceitação da conversação), podendo cada processo possuir, em adição, seu próprio teste de aceitação.

A mais simples das implementações consiste em dotar os blocos de recuperação de um campo identificador de conversações. Deste modo, um conjunto de blocos de recuperação, cada qual associado a um processo, constituirá uma conversação, caso seus identificadores sejam iguais. Portanto, a estrutura da conversação aparece diluída no programa.

Uma outra implementação baseia-se na utilização de um monitor especial, o *monitor-conversação*, dentro dos blocos de recuperação. O nome do monitor-conversação tem a mesma função do identificador da implementação citada anteriormente, de modo que a cada conversação está associado um monitor-conversação.

Uma terceira implementação fundamenta-se na criação de um tipo abstrato de dados *conversação*, que é constituído, basicamente, de monitores-conversação e de uma ou mais seções de interação.

A mais sofisticada das implementações propostas baseia-se na criação de blocos de recuperação concorrentes. Um bloco de recuperação concorrente possui a mesma estrutura de um bloco de recuperação comum, acrescida de declarações para a criação de processos-filhos. Um processo, ao entrar num bloco de recuperação concorrente, deverá inicializar os monitores e os processos-filhos que constituem a seção de interação. A seguir, passa a um estado de espera, até que todos os processos-filhos tenham terminado sua execução. Então, o processo-fonte deverá deixar o estado de espera e

executar o teste de aceitação, descartando os monitores associados. Essa implementação previne a ocorrência de processos desertores, visto que ao iniciar uma conversação, todos os processos participantes devem ser inicializados. Entretanto, as regras de execução necessárias podem trazer grandes custos em termos de desempenho e, em certos casos, a perda de concorrência pode ser considerável.

Em [Kim, 88a] é proposta uma implementação que visa isentar o projetista das preocupações associadas ao estabelecimento de pontos de recuperação, quando estão envolvidos processos concorrentes. Deste modo, o projeto das estruturas de detecção e recuperação de erros pode ser realizado de maneira independente para cada processo. O sistema processador é encarregado de, automaticamente, estabelecer pontos de recuperação adicionais àqueles que cada processo estabelece explicitamente (por programação). Esse estabelecimento de pontos de recuperação visa assegurar que:

(a)um processo nunca realize dois retornos consecutivos, sem que uma re-execução seja tentada;

(b)todos os retornos sejam de mínima distância.

A detecção e a recuperação de erros são estruturadas por blocos de recuperação e a comunicação entre processos se dá através de monitores.

2.4.4-Tolerância a Faltas de Software para Tempo Real

Em aplicações de tempo real é necessário que os resultados, além de estarem corretos, sejam fornecidos dentro dos limites de tempo especificados. As técnicas de tolerância a faltas têm associados custos em termos de desempenho. Portanto, sua aplicação a sistemas de tempo real deve ser cuidadosamente estudada, tendo em vista as restrições de tempo que tais sistemas apresentam.

As técnicas de tolerância a faltas de software propostas para sistemas de tempo real, em geral, atacam o problema das restrições de tempo através de mecanismos de watch-dog [Hecht, 76]. Serão apresentadas a seguir, algumas dessas técnicas, que constituem extensões ao conceito de blocos de recuperação.

Tempo Limite (Deadline)

Nesta técnica, descrita em [Anderson, 81], um sistema é considerado como um conjunto de processos independentes, que devem realizar seus respectivos serviços dentro de certos limites de tempo. A cada processo são associados um algoritmo primário e um alternativo. A fim de evitar uma falha, caso o algoritmo primário não consiga concluir sua execução dentro de um tempo determinado, o algoritmo alternativo deverá ser acionado. Esse algoritmo alternativo deve ser projetado para fornecer um serviço degradado, num tempo consideravelmente menor que o necessário para a execução do algoritmo primário. O tempo limite concedido para a execução do

algoritmo primário deve ser suficiente para que o algoritmo alternativo, caso seja necessário, sempre possa ser executado, tendo em vista o tempo limite para a entrega do serviço.

Técnica de Hecht

Na técnica proposta em [Hecht, 86], considera-se um bloco de recuperação com um módulo primário e um módulo alternativo. Ao bloco de recuperação é associado um temporizador. O suporte de execução contém um módulo *status*; um *flag*, que indica se o módulo primário falhou e um contador de execuções do módulo alternativo.

A execução do bloco de recuperação ocorre da seguinte maneira:

(a)Antes de iniciar o bloco de recuperação, o módulo *status* verifica o *flag*. Se este não estiver ativo (o que indica que o módulo primário não falhou), o módulo *status* chama o módulo primário. Ao ingressar no bloco de recuperação, o suporte chama os módulos primário e alternativo e carrega o temporizador com o tempo máximo de execução permitido para o módulo primário. O controle é passado ao módulo primário, que é executado. Quando o módulo primário completa sua execução, o teste de aceitação é realizado. Se os resultados são aceitos pelo teste e estão dentro do limite de tempo estabelecido, o controle retorna ao suporte, que recarrega o temporizador com um intervalo apropriado para a próxima operação e faz prosseguir a operação do programa.

(b)Se o módulo primário não é aprovado no teste de aceitação ou seus resultados não são fornecidos dentro do tempo estipulado, o módulo alternativo é chamado. O *flag* é ativado, o temporizador é ativado com o tempo máximo de execução previsto para o módulo alternativo e este é então executado. Quando o módulo alternativo completa sua execução, o teste de aceitação é realizado. Se o módulo alternativo é aprovado no teste de aceitação, o controle retorna ao suporte, que procede conforme descrito anteriormente.

(c)Se os resultados do módulo alternativo são rejeitados pelo teste de aceitação ou não são fornecidos em tempo hábil, uma condição de aborto é acionada. Quando o módulo alternativo fornece resultados não aceitáveis, mas dentro dos limites de tempo, o fato do *flag* ter sido ativado impede que a sua execução seja repetida.

(d)Quando, ao iniciar o bloco de recuperação, o *flag* estiver ativo, o módulo *status* deverá verificar o contador de execuções do módulo alternativo. Se o contador estiver abaixo de um certo valor limite (fixo do sistema ou específico da aplicação), o módulo *status* incrementa o contador e chama o módulo alternativo. Caso contrário, o *flag* é desativado, o contador é reinicializado e o módulo primário é chamado.

Blocos de Recuperação Distribuídos (DRB)

A técnica dos Blocos de Recuperação Distribuídos (DRB) é proposta em [Kim, 84] como uma abordagem uniforme para o tratamento de faltas de

hardware e de software em sistemas distribuídos de tempo real. Esta técnica combina o uso de redundâncias ativas e passivas. O sistema considerado é formado por múltiplas estações, que podem ser constituídas de um ou mais nós. Cada estação pode executar apenas um bloco de recuperação.

Em cada nó estão presentes os mesmos módulos de software e o mesmo teste de aceitação, isto é, cada nó implementa uma réplica do bloco de recuperação. Portanto, internamente aos nós, utilizam-se redundâncias passivas. O teste de aceitação inclui as restrições de tempo à execução dos algoritmos. Os nós primário e alternativos são executados concorrentemente (redundâncias ativas) e devem ter acesso aos mesmos dados de entrada. Ao nó primário cabe o envio de resultados ao exterior.

As funções dos módulos de software são, inicialmente, atribuidas de maneira oposta em cada nó, isto é, o algoritmo primário em um nó será alternativo em outro. Desta forma, tem-se a execução concorrente dos módulos de software que compõem o bloco de recuperação, o que produz um efeito de recuperação em avanço, reduzindo os custos associados ao processamento de erros.

A detecção e a recuperação de erros está fundamentada na cooperação entre os nós durante cada ciclo de processamento, o que ocorre em três pontos: após a recepção de dados de entrada e estabelecimento de um ponto de recuperação, quando há a informação dos resultados dos testes de aceitação e quando há a confirmação do envio de resultados à estação sucessora.

A condição dos nós (primário ou alternativo), bem como a função de cada módulo de software em um nó pode ser alterada em função da ocorrência de faltas.

As implementações de DRB propostas [Kim, 88] referem-se a sistemas distribuídos fortemente acoplados. O compartilhamento de memória agiliza a comunicação entre os nós, proporcionando uma recuperação de erros mais rápida. Esta característica é ideal para aplicações de tempo crítico (hard real time), entretanto, torna o modelo altamente dependente do hardware. Um outro fator a ser considerado é que a presença de um DRB numa estação não é transparente às demais, de modo que as estações predecessora (emissora dos dados de entrada) e sucessora (receptora dos resultados) precisam tratar com alguns aspectos do modelo de tolerância a faltas. Esta característica acrescenta complexidade à programação do DRB.

2.5-Conclusão

Neste capítulo foi apresentada a terminologia básica utilizada em segurança de funcionamento e tolerância a faltas. Foram também apresentados e discutidos os principais modelos e técnicas propostos na literatura visando implementar a tolerância a faltas de projeto, especialmente no tocante a sistemas distribuídos de tempo real.

No capítulo seguinte será apresentada a proposta de um modelo para programação e gerenciamento de redundâncias em sistemas distribuídos, voltado a aplicações em tempo real, o qual é o objeto deste trabalho.

CAPÍTULO 3

PROPOSIÇÃO DE MODELOS PARA A IMPLEMENTAÇÃO DE REDUNDÂNCIAS EM SISTEMAS DISTRIBUÍDOS

3.1-Introdução

Neste capítulo será apresentado um modelo para a programação de redundâncias em sistemas distribuídos. O modelo proposto está baseado no conceito de **Blocos de Recuperação Distribuídos** (DRB) [Kim, 84], utilizando como linguagem base a linguagem LIS [Fraga, 89], cujo paradigma de programação possibilita uma configuração flexível do sistema distribuído.

Na implementação do DRB na linguagem LIS podem ser identificados dois modelos: **modelo de tolerância a faltas** e **modelo de tratamento de faltas.** O modelo de tolerância a faltas está relacionado aos aspectos de detecção e recuperação de erros, enquanto o modelo de tratamento de faltas tem por objetivo a restituição de redundâncias perdidas durante a operação do sistema.

O paradigma de programação da linguagem LIS será apresentado a seguir. Posteriormente, serão discutidos os modelos de tolerância a faltas e de tratamento de faltas. Os principais aspectos de implementação e os resultados obtidos serão objeto do próximo capítulo.

3.2-Paradigma de Programação Distribuída na Linguagem LIS

Segundo o paradigma de programação da linguagem LIS, um programa distribuído consiste num conjunto de módulos, os quais são interconectados durante o processo de configuração do sistema. Os módulos, unidades básicas de configuração, têm visibilidade externa associada exclusivamente a seus portos. A ligação de portos de saída a portos de entrada forma canais de sincronização e comunicação. Estas conexões são definidas externamente aos módulos, durante o processo de configuração, permitindo a separação entre a programação desses módulos e a configuração do sistema. Esta característica está de acordo com o clássico princípio da decomposição modular [De Remer, 76]. A configuração do sistema pode ser modificada dinamicamente, isto é, durante sua operação, sem a parada do mesmo.

Neste modelo de programação, cada módulo encapsula uma ou mais tarefas, que são as unidades básicas de concorrência. Conceitualmente, tarefas em um módulo apresentam um forte acoplamento. A visibilidade externa das tarefas se dá apenas através de seus portos. A ligação desses portos a outros portos de tarefas define canais de comunicação internos ao módulo, enquanto que sua associação a portos do módulo encapsulador

possibilita a comunicação externa. As ligações internas entre tarefas, programadas nos módulos, podem ser alteradas em tempo de execução, por tarefas do próprio módulo. O paradigma de programação da linguagem LIS é mostrado na Fig. 3.1.

As abstrações desse modelo de programação são implementadas na linguagem LIS a partir de três declarações básicas: **MODULE**, **SYSTEM** e **CHANGE**. Estas declarações serão apresentadas no capítulo seguinte, juntamente com outros aspectos relevantes da linguagem.

A linguagem LIS possui características que favorecem a adoção de técnicas de tolerância a faltas, dentre as quais destacam-se : a decomposição modular, a configuração dinâmica e a possibilidade de reconfiguração interna do módulo.

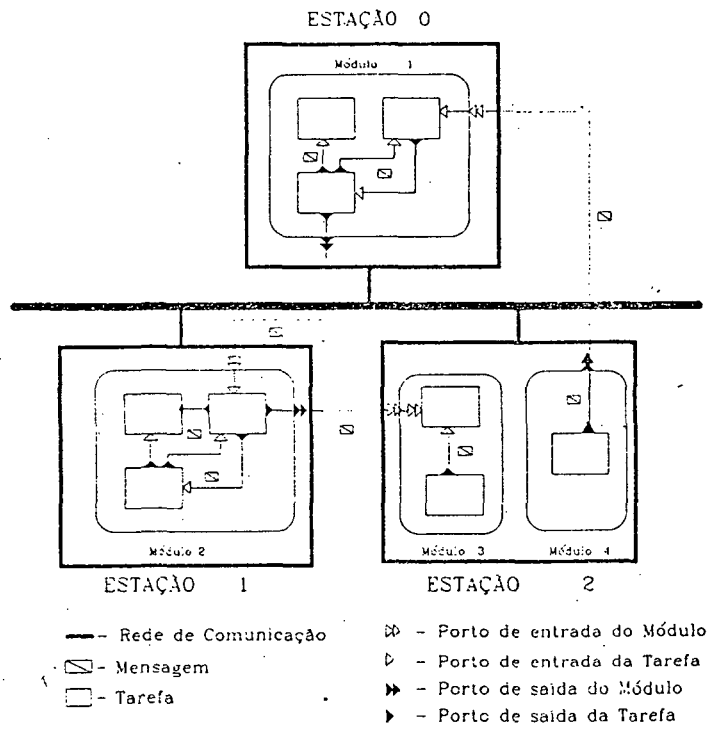


Fig. 3.1 - Paradigma de Programação LIS

3.3-Modelo de Tolerância a Faltas

Blocos de Recuperação Distribuídos (DRB) constituem uma abordagem uniforme para a tolerância a faltas de hardware e software em sistemas de tempo real. As implementações que têm sido propostas na literatura referem-se a sistemas distribuídos fortemente acoplados [Kim, 88], [Kim, 89], [Kim, 91] e [Kim, 91a]. O presente trabalho procura ampliar esta abordagem, através do desenvolvimento de ferramentas que permitam simplificar a programação de redundâncias. Desta forma, pretende-se tornar o modelo independente do hardware, possibilitando estender sua aplicação a sistemas fracamente acoplados.

A linguagem LIS recebeu extensões, com o objetivo de facilitar a programação de redundâncias, isentando o programador dos encargos associados à implementação do modelo. Desta forma, ao programador caberão apenas as atribuições que lhe são inerentes, isto é, a programação dos algoritmos alternativos e do teste de aceitação (incluindo as restrições de tempo), sendo que todas as funcionalidades necessárias ao modelo de tolerância a faltas serão geradas automaticamente, a partir de declarações da linguagem.

3.3.1-Estrutura do Modelo de Tolerância a Faltas

O modelo proposto fundamenta-se na criação, a partir da linguagem LIS, de um tipo módulo especial, **o tipo módulo tolerante a faltas (módulo** FT). Durante o processo de configuração do sistema, o tipo módulo FT

originará duas instâncias : instância primária e instância secundária. Essas instâncias serão carregadas em duas unidades de processamento distintas, atuando como redundâncias ativas e devendo, portanto, ter acesso aos mesmos dados de entrada (mensagens de um módulo emissor). À instância primária caberá enviar para o exterior (módulo receptor), os resultados do processamento do módulo FT. As instâncias primária e secundária constituem uma implementação do Bloco de Recuperação Distribuído (DRB).

A Fig. 3.2 mostra a estrutura do modelo de tolerância a faltas, considerando uma interface assíncrona. A versão que utiliza interface de comunicação síncrona possui estrutura similar, sendo os portos T1 e T4 substituídos por um único porto bidirecional. Cada instância do módulo FT é constituída pelas tarefas: A, B, TEST_I/O e CONTROLE.

As tarefas A e B implementam os algoritmos alternativos da aplicação. Esses algoritmos apresentam projetos de software diferentes, visando produzir resultados iguais ou similares para um mesmo problema. O algoritmo preferencial é representado pela tarefa A, enquanto a tarefa B constitui um algoritmo alternativo. As configurações internas das instâncias primária e secundária de um módulo FT são diferentes, no que se refere à situação das tarefas A e B, conforme a Fig. 3.2. A tarefa (A ou B) que estiver conectada à tarefa TEST_I/O estará ativa, visto que terá todos os recursos para que possa ser executada, exceto os dados de entrada. A tarefa que estiver desconectada permanecerá suspensa, funcionando como uma redundância passiva.

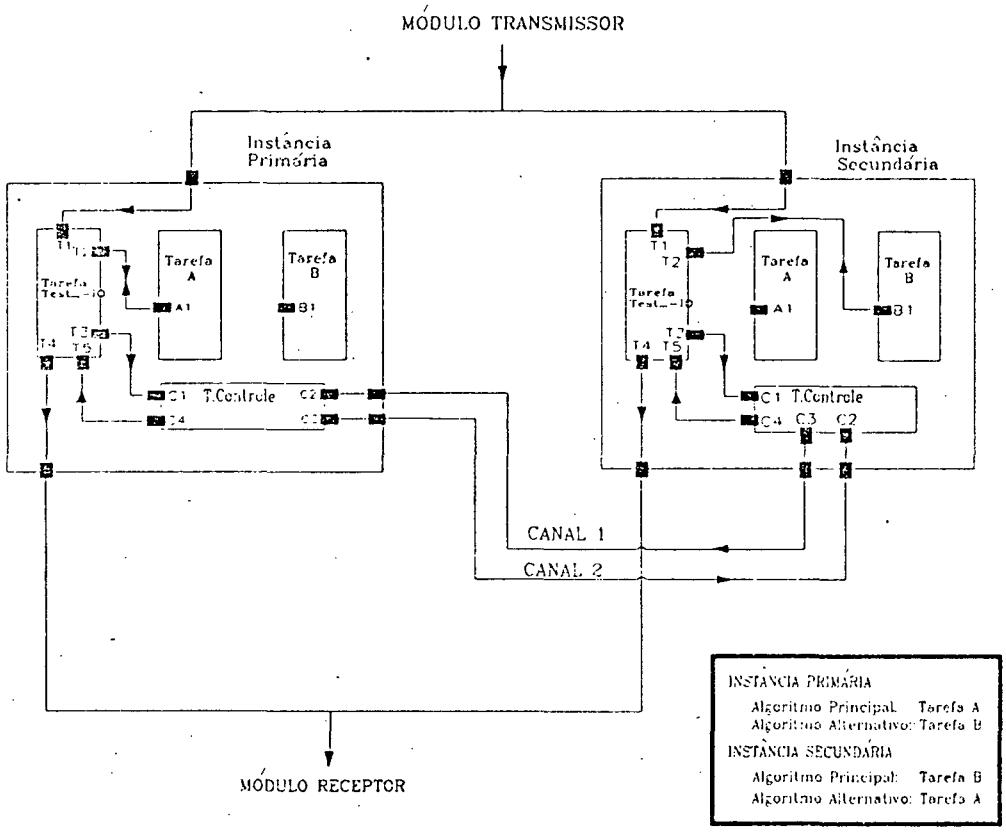


Fig. 3.2 - Modelo de Tolerância a Faltas

A tarefa TEST_I/O é responsável pela recepção dos dados de entrada, preparação do ciclo de processamento, execução do teste de aceitação e entrega dos resultados ao módulo receptor. Um *ciclo de processamento* no DRB tem início quando chega uma mensagem com dados de entrada e termina quando a instância primária envia ao módulo receptor o respectivo resultado. A cada ciclo de processamento estará associado um número, que corresponderá ao número da mensagem a ser tratada nesse ciclo.

A preparação do ciclo de processamento consiste em salvar o estado do DRB e enviar os dados de entrada à tarefa de aplicação ativa (tarefa A ou tarefa B). O estado do DRB é constituído pelas variáveis globais do módulo FT e pela fila de mensagens ainda não tratadas pelo porto de entrada, no caso assíncrono. As variáveis globais são variáveis cujos valores devem permanecer entre ciclos de processamento. Estas variáveis servirão como dados de entrada para o algoritmo de aplicação ativo (tarefa A ou B), juntamente com a mensagem de entrada a ser tratada no ciclo de processamento em questão. O fato do estado ser salvo (estabelecimento de um ponto de recuperação), a cada ciclo de processamento, permite a recuperação de erros em retrocesso, quando da falha do teste de aceitação. Neste caso, o estado DRB deverá ser restaurado e a instância deverá ser reconfigurada internamente (conexão/desconexão das tarefas A e B à tarefa TEST_I/O). O ciclo de processamento será então reiniciado, com o envio dos dados de entrada à nova tarefa ativa. Esta terá, então, acesso aos mesmos dados (mensagem de entrada e variáveis globais) a que teve acesso a tarefa anteriormente ativa.

Em aplicações de tempo real, a aceitabilidade dos resultados está associada não apenas à sua correção, mas também à sua entrega dentro de limites de tempo especificados. À tarefa TEST_I/O cabe verificar as restrições de tempo associadas à execução dos algoritmos de aplicação nas instâncias do módulo FT. Estas restrições de tempo são implementadas através de mecanismos de "timeout", disponíveis nas primitivas de comunicação da linguagem LIS. Uma violação aos limites de tempo nos canais internos de comunicação de uma instância FT é equivalente a uma falha no teste de aceitação do algoritmo ativo.

A tarefa CONTROLE é responsável por todas as comunicações entre as instâncias e pelas decisões sobre reconfiguração interna, troca de funções entre as instâncias e ainda pela decisão sobre o envio de resultados ao módulo receptor e pela sinalização de exceções ao gerenciador de faltas (a ser apresentado em 3.4.1). O algoritmo básico da tarefa CONTROLE é apresentado no apêndice deste trabalho, através de um pseudo-código.

3.3.2-Interações entre as instâncias

A realização de uma recuperação de erros eficiente, que explore toda a potencialidade do modelo, depende, em grande parte, de uma troca de informações efetiva entre as instâncias, a cada ciclo de processamento. A troca de informações deve ocorrer em pontos-chave da execução de um ciclo de processamento do DRB, de modo a proporcionar a cada instância um acompanhamento do comportamento de sua instância par.

A troca de informações entre as instâncias ocorre através do envio/recepção de mensagens através dos canais unidirecionais 1 e 2, mostrados na Fig. 3.2, os quais estão associados às tarefas CONTROLE de cada instância. As interações ocorrem em três pontos :

(a) *Interação 1* : após a recepção dos dados de entrada e estabelecimento de um ponto de recuperação; esta interação permite que sejam detectadas divergências entre as instâncias no que se refere ao número do ciclo de processamento.

(b) *Interação 2* : após a execução do teste de aceitação ou esgotamento do "timeout" associado à execução dos algoritmos de aplicação; esta interação permite que a instância tome conhecimento do sucesso ou não do teste de aceitação na instância par.

(c) *Interação 3* : após o envio dos resultados para o módulo receptor; neste ponto, a instância primária comunica à instância secundária que enviou os resultados ao módulo receptor.

Em todas as interações, a ausência de mensagens nos canais de comunicação é detectada através de mecanismos de "timeout" associados às operações de recepção. Os períodos de tempo especificados para estes mecanismos levam em consideração o desempenho do suporte de comunicação.

De acordo com as mensagens recebidas (ou esperadas e não recebidas) nos pontos de interação, a tarefa CONTROLE estará envolvida em decisões que poderão ocasionar a reconfiguração interna do módulo FT, a troca de funções entre as instâncias ou a sinalização de exceções ao módulo gerenciador de faltas.

3.3.3–Comportamento do modelo sob condições de falta

Ao início de cada ciclo de processamento, as duas instâncias FT irão diferir em sua configuração interna, de modo que a instância primária deverá executar a tarefa A como seu algoritmo principal, enquanto a

instância secundária executará a tarefa B. Na ausência de faltas, o teste de aceitação resultará positivo em ambas as instâncias, ficando a cargo da instância primária a entrega dos resultados do processamento ao módulo receptor. A Fig. 3.3.a mostra a configuração inicial das instâncias FT.

As condições de faltas podem ser resumidas da seguinte forma :

(a)Caso o teste de aceitação falhe na instância primária e seja bem-sucedido na instância secundária, as instâncias deverão trocar suas funções. Neste caso, a antiga instância primária será reconfigurada internamente e tentará executar seu algoritmo alternativo, no mesmo ciclo de processamento (Fig. 3.3.b).

(b)Caso o algoritmo principal da instância secundária falhe e a instância primária alcance um resultado positivo, não haverá troca de funções entre as instâncias. A instância secundária executará seu algoritmo alternativo, no mesmo ciclo de processamento (Fig. 3.3.c).

(c)Caso ambas as instâncias venham a falhar na execução de seus algoritmos principais, num mesmo ciclo de processamento, estas deverão ser reconfiguradas, passando a executar seus respectivos algoritmos alternativos. Neste caso, as instâncias permanecerão em suas funções de primária e secundária (Fig. 3.3.d).

(d)Quando, num mesmo ciclo de processamento, as execuções dos dois algoritmos alternativos falharem numa instância, uma exceção deverá ser sinalizada ao gerenciador de faltas.

		Instância X	Instância Y
(a)	função e configuração iniciais	primária	secundária
		A	B
(b)	situação A	secundária	primária
		B	B
(c)	situação B	primária	secundária
		A	A
(d)	situação C	primária	secundária
		B	A

Fig. 3.3 - Funções e configurações internas das instâncias FT após a ocorrência de faltas

A ocorrência de faltas poderá alterar a condição das tarefas A e B (algoritmo principal ou alternativo) numa determinada instância e/ou a função das instâncias. Visando um melhor desempenho do modelo, visto que a tarefa A apresenta o algoritmo preferencial, foi adotada como estratégia que a instância primária terá sempre como seu algoritmo principal a referida tarefa. Portanto, ao final de cada ciclo de processamento, a tarefa CONTROLE verificará se a relação entre a função da instância (primária ou secundária) e o algoritmo principal (tarefa ativa) está de acordo com a estratégia adotada; caso contrário, será providenciada a reconfiguração interna da instância. Considerando, por exemplo, a condição de falta (a), ao final do ciclo de processamento, o DRB apresentará a configuração mostrada na Fig. 3.3.b. Para que, no ciclo subseqüente, a configuração interna das instâncias FT esteja de acordo com a estratégia adotada, será necessário que a instância Y, alçada à condição de primária, seja reconfigurada internamente,

passando a ter como algoritmo ativo aquele executado pela tarefa A (Fig. 3.4).

INSTÂNCIA X (secundária)	INSTÂNCIA Y (primária)
algoritmo B	algoritmo A

Fig. 3.4 - Configuração interna das instâncias FT no ciclo seguinte à ocorrencia de falta (situação a)

O modelo de tolerância a faltas proposto permite tolerar uma falta de software e uma falta de hardware. Em cada instância FT, faltas de projeto independentes presentes nos algoritmos alternativos serão toleradas; faltas de modo comum entre os algoritmos serão detectadas, enquanto as faltas de modo comum entre os algoritmos e o teste de aceitação não serão toleradas, nem detectadas. Podem ser identificadas no modelo características das arquiteturas *RB/1/1* e *NSCP/1/1*, descritas em [Laprie, 90]. Um estudo comparativo dos vários tipos de arquiteturas tolerantes a faltas pode ser encontrado em [Nacamura, 92]. Considerando as faltas quanto aos seus efeitos, o modelo tolera faltas de temporização (omissão, "crash") e faltas por valor, através do teste de aceitação. O comportamento arbitrário do módulo emissor pode ter seus efeitos tolerados no modelo em questão. O

comportamento arbitrário de uma das instâncias pode ser detectado e tratado, através do modelo de tratamento de faltas (seção 3.4).

3.3.4-Comunicações externas com o DRB

Um tipo módulo tolerante a faltas, da mesma forma que quaisquer outros módulos, tem acesso a todos os tipos de comunicação disponíveis no paradigma de programação da linguagem LIS. Para todos esses tipos de comunicação, há a possibilidade de utilização de endereçamento multi-destinação.

O modelo de tolerância a faltas utiliza as características de multi-destinação de mensagens da linguagem LIS para enviar os mesmos dados de entrada, em cada ciclo de processamento, a ambas as instâncias. O suporte utilizado na comunicação remota permite a difusão seletiva de mensagens ("multicast") e é dotado de um serviço de numeração de mensagens. Entretanto, o suporte de comunicação é considerado não-confiável, visto que permite que mensagens sejam perdidas. A ordenação das mensagens na recepção, fundamentada na numeração, não envolve a reposição das mensagens perdidas. O modelo de tolerância a faltas apresenta mecanismos para detectar a ausência de mensagens ou a inconsistência nos dados de entrada, através das ações associadas às interações (1), (2) e (3), apresentadas em 3.3.2.

O problema da consistência dos dados de entrada é especialmente crítico em comunicações assíncronas, pois o módulo emissor não é bloqueado após a emissão. Nesse caso, cada instância do DRB deve processar uma fila

de mensagens. A numeração de mensagens assegura que as redundâncias ativas (instâncias FT) tratarão com os mesmos dados de entrada, a cada ciclo de processamento. O número da última mensagem consumida irá determinar o número do ciclo de processamento. Através de primitivas do suporte de comunicação, a tarefa TEST_I/O tem acesso ao número da mensagem recebida.

A interação (1) permite que as instâncias troquem informações a respeito do número de ciclo e se sincronizem, ao início de cada ciclo. Dois problemas podem ser detectados nesse ponto : a recepção de uma fila de mensagens com descontinuidade (perda de mensagens) e possivelmente, como conseqüência, a divergência no número de ciclo de processamento. Nesse caso, a instância que possuir o número de ciclo correto deverá prosseguir sua execução e a outra deverá ser bloqueada até receber da instância par, ao final do ciclo, o estado do DRB, que inclui a fila de mensagens de entrada. Essa transferência é supervisionada pelo gerenciador de faltas e permite que no próximo ciclo de processamento as duas instâncias estejam em acordo.

A descontinuidade nas filas de mensagens de ambas as instâncias irá acarretar um comportamento que dependerá da aplicação. Em algumas situações poderá ser conveniente prosseguir a execução (considerando o menor número de ciclo de processamento, caso haja divergência na interação 1). Em casos críticos, onde a continuidade das mensagens é exigida, poderá ser necessário restaurar o DRB. A escolha da opção mais adequada a cada aplicação caberá ao programador.

No caso de interface síncrona, a inconsistência nos dados de entrada não é tão crítica. O módulo emissor (cliente do DRB) estará bloqueado durante o ciclo de processamento, aguardando os resultados. Desta forma, a principal função das interações (1), (2) e (3) passará a ser a detecção da ausência de dados de entrada em uma das instâncias. Neste caso, a instância considerada não comparecerá a nenhum dos pontos de interação definidos no modelo, o que será detectado pela instância par. Caso as duas instâncias não recebam a mensagem de entrada, a linguagem LIS oferece mecanismos que possibilitam que isto seja detectado pelo módulo emissor.

Em ambas as versões (síncrona e assíncrona), uma falha na instância primária, após enviar seus resultados ao módulo receptor e antes de executar a interação (3), poderá levar a instância secundária a também enviar resultados. O suporte de comunicação, através do sistema de numeração de mensagens efetuará o descarte da mensagem redundante, sempre que houver envios relacionados ao mesmo ciclo de processamento.

3.4-Modelo de Tratamento de Faltas

O modelo de tolerância a faltas, apresentado anteriormente, visa possibilitar que o módulo FT entregue resultados válidos (em termos de correção e cumprimento de restrições de tempo), mesmo na presença de faltas. No entanto, os efeitos da ocorrência de uma falta, dependendo da sua natureza, poderão vir a degradar o DRB, reduzindo sua capacidade de detecção e recuperação de erros e/ou comprometendo seu desempenho. Neste caso enquadram-se as faltas permanentes como, por exemplo, uma falta física

em uma estação (elemento processador), causando a perda de uma instância FT.

O modelo de tratamento de faltas proposto tem por objetivo básico a restauração do DRB, sempre que uma falta ou um conjunto de faltas vier a proporcionar uma degradação em seu comportamento. Os procedimentos de restauração poderão envolver desde uma simples transferência de estado de uma instância à outra até a parada do DRB, em casos extremos.

O modelo de tratamento de faltas fundamenta-se na interpretação e no tratamento de exceções sinalizadas a partir das tarefas CONTROLE das instâncias FT. Essas sinalizações de exceção indicam desvios no comportamento esperado do DRB, a cada ciclo de processamento.

3.4.1-Estrutura do Modelo

A estrutura do modelo de tratamento de faltas no DRB é mostrada na Fig. 3.5. O elemento central deste modelo é o **gerenciador de faltas** que, a partir da interpretação das exceções sinalizadas pelas tarefas CONTROLE, infere a provável falta ocorrida e, caso seja necessário, comanda o processo de restauração do DRB. O gerenciador de faltas mantém um histórico completo de todas as exceções ocorridas durante o funcionamento do DRB.

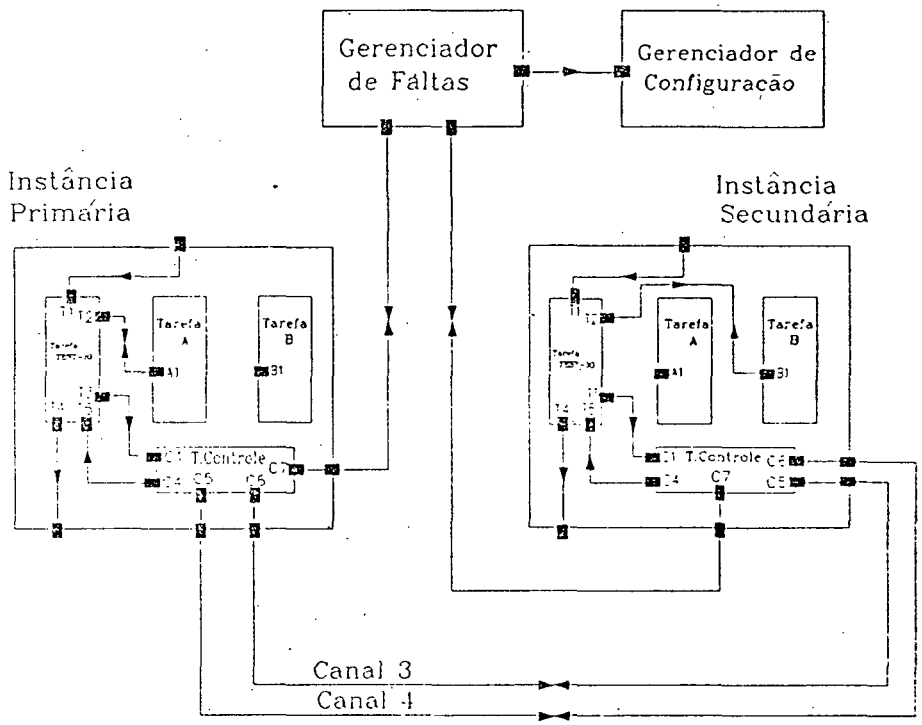


Fig. 3.5 - Modelo de Tratamento de Faltas

Quando, no decorrer de um ciclo de processamento, são detectadas situações de exceção, a tarefa CONTROLE da instância em questão sinaliza essa ocorrência ao gerenciador de faltas, através do envio de uma mensagem de exceção. Essa mensagem deve caracterizar perfeitamente, de forma explícita ou implícita, a exceção ocorrida.

Ao receber uma sinalização de exceção, o gerenciador de faltas deverá interpretá-la, contabilizá-la e, consultando o histórico de exceções por ele mantido, poderá tomar alguma decisão. Essa decisão resultará numa das três ações seguintes :

(a) *Transferência de Estado;*

(b) *Reconstrução do DRB;*

(c) *Parada do DRB.*

A parada do DRB está associada a casos críticos, que extrapolem ao escopo do modelo de tolerância a faltas, como, por exemplo, faltas de modo comum aos dois algoritmos alternativos. Já as duas primeiras ações citadas possibilitarão a restauração do DRB ao seu estado normal de funcionamento.

3.4.2-Transferência de Estado

O estado do DRB é constituído pelas variáveis globais e pela fila de mensagens de entrada ainda não tratadas, no caso assíncrono. Ao decidir que uma transferência de estado deve ser realizada, o gerenciador de faltas inicia o processo, através de uma sinalização às tarefas CONTROLE das instâncias. A instância secundária, ao ser sinalizada, envia uma mensagem à instância primária, solicitando o envio do estado. A transferência de estado à instância secundária ocorrerá ao final do ciclo de processamento da instância primária. O protocolo de comunicação entre as instâncias, necessário à troca de estado, é esboçado no diagrama da Fig. 3.6. Este protocolo utiliza os canais bidirecionais 3 e 4, mostrados na Fig. 3.5.

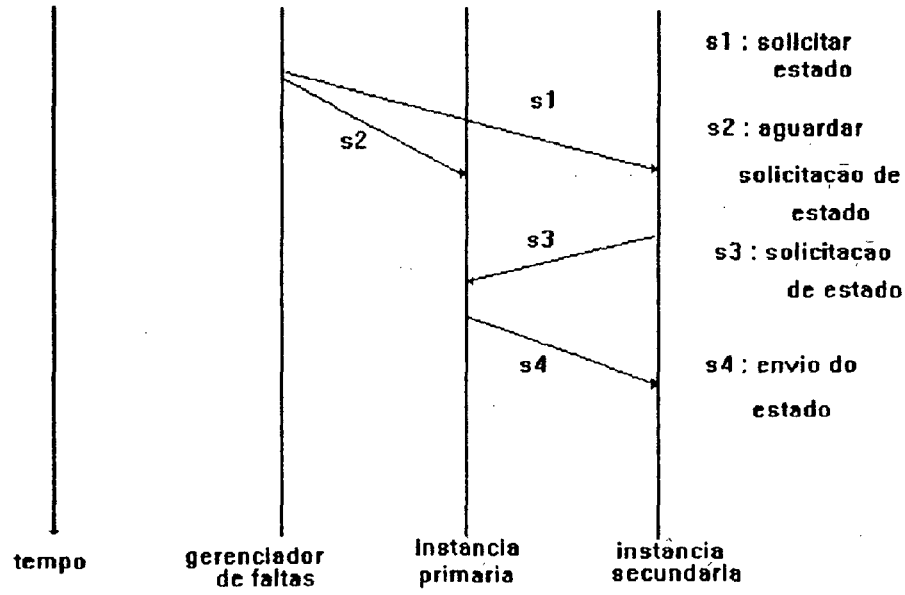


Fig. 3.6 - Protocolo de Transferência de Estado

3.4.3-Reconstrução do DRB

Diante da necessidade de reconstruir o DRB, em função da ocorrência de falta permanente em uma instância, o gerenciador de faltas enviará ao módulo gerenciador de configuração (Fig. 3.5) uma mensagem indicando que o sistema deverá ser reconfigurado (configuração dinâmica).

O módulo gerenciador de configuração, ao receber a sinalização do gerenciador de faltas, passará a interpretar uma lista de ações, gerada a partir dos resultados da tradução de uma declaração CHANGE da linguagem LIS pré-planejada (as declarações da linguagem LIS serão discutidas no próximo capítulo). A execução dessa lista de ações produzirá as operações de configuração necessárias à reconstrução do DRB.

Considerando, por exemplo, a restituição de uma instância perdida, o processo de reconstrução do DRB envolverá o recarregamento do módulo FT em uma estação do sistema distribuído e a criação a partir deste tipo de uma nova instância (a nova instância secundária). Além disto, deverá ocorrer o estabelecimento dos canais de comunicação associados, ligando a nova instância à instância que permaneceu ativa no DRB e aos módulos emissor e receptor.

De acordo com a natureza funcional do módulo FT, podem ser consideradas duas possibilidades para a reconstrução do DRB :

(a)caso as funções associadas ao DRB sejam independentes da localização, as funções correspondentes à unidade de processamento faltosa poderão ser realocadas em qualquer outra estação passível de reconfiguração;

(b)caso as funções associadas ao DRB dependam da localização no sistema, a unidade de processamento faltosa deverá ser reparada ou substituída por uma unidade reserva, para que a instância perdida possa ser reativada.

Para que a restauração do DRB seja consistente é necessário que a nova instância secundária tenha acesso aos valores atuais das variáveis globais e aos mesmos dados de entrada (filas de mensagens, no caso assíncrono) que a instância ativa (instância primária). Portanto, a última fase do processo de reconstrução do DRB envolverá a transferência de estado da instância primária para a instância secundária, o que se processará conforme descrito anteriormente.

Deve ser ressaltado que a instância primária prosseguirá sua execução de forma praticamente independente durante o processo de reconstrução do DRB. A única interferência no processamento da instância primária se dá pela solicitação de estado por parte da nova instância secundária. Os custos dessa transferência, em termos de desempenho, dependerão essencialmente da quantidade e do tamanho das variáveis globais e das filas de mensagens de entrada armazenadas em "buffers". Em aplicações típicas de automação industrial e controle de processos, o tamanho das mensagens contendo informações de estado não é significativo a ponto de comprometer o desempenho do modelo.

Além da sinalização relativa à reconstrução do DRB, o módulo gerenciador de faltas deve periodicamente enviar uma sinalização de controle ao módulo gerenciador de configuração. A ausência de mensagens oriundas do gerenciador de faltas por um tempo determinado será interpretada como falha do mesmo e levará o gerenciador de configuração a reconstruir integralmente o DRB (modelos de tolerância a faltas e tratamento de faltas).

3.4.4-Tratamento de Exceções

As mensagens de exceção sinalizadas pelas tarefas CONTROLE ao gerenciador de faltas deverão conter um número associado ao tipo de exceção e o número do ciclo de processamento em execução. De acordo com o porto pelo qual a mensagem de exceção é recebida, o gerenciador de faltas recebe também, de forma implícita, a informação sobre a localização da exceção sinalizada, isto é, em qual das instâncias a mesma ocorreu.

Estas informações (tipo de exceção, instância onde ocorreu e ciclo de processamento no qual ocorreu) caracterizam perfeitamente a exceção, permitindo a adoção da estratégia de tratamento adequada. Os diferentes tipos de exceções que podem se apresentar no sistema levarão o gerenciador de faltas a realizar diferentes ações de tratamento, conforme será discutido a seguir. As exceções tratadas e seus correspondentes tratamentos são sumarizados na tabela mostrada na Fig. 3.7.

(a)*Exceções E1, E2 e E3*

Estas exceções indicam, respectivamente, a ausência de mensagem nas interações (1), (2) e (3). Suas sinalizações decorrem do esgotamento de tempo ("timeout") na interação correspondente. Podem ser identificadas duas causas básicas : a falha na comunicação ou a falha na instância FT, que deixa de se comunicar com a sua instância par (falta de omissão).

O módulo gerenciador de faltas realiza a contagem do número de ocorrências consecutivas de cada exceção. A cada tipo de exceção está associado um número máximo de ocorrências consecutivas permitidas. Quando esse limite é atingido, a instância em questão passa a ser considerada como faltosa (falta de "crash"). Neste caso, o gerenciador de faltas sinalizará ao configurador que o processo de reconstrução do DRB deve ser iniciado.

A ocorrência, num mesmo ciclo de processamento, das exceções E1, E2 e E3, referentes à ausência de mensagens de uma instância é tratada pelo gerenciador de faltas, segundo duas possibilidades :

-No caso de interface assíncrona, a ocorrência das três exceções pode ter várias causas e, numa primeira ocorrência, o gerenciador de faltas limita-se a contabilizá-la;

-No caso de interface síncrona, a ocorrência das três exceções no mesmo ciclo é interpretada como inconsistência nos dados de entrada (ausência de dados na instância omissa). O gerenciador de faltas deverá sinalizar para que, ao final do ciclo de processamento, ocorra a transferência de estado da instância correta para a instância omissa.

Tanto no caso síncrono como no caso assíncrono, a persistência das exceções E1, E2 e E3 no próximo ciclo representará um forte indício de falha na instância omissa. Desta forma, o gerenciador de faltas sinalizará ao configurador, iniciando o processo de reconstrução do DRB.

(b) *Exceção E4*

Esta exceção indica que ambos os algoritmos alternativos falharam numa mesma instância, durante um mesmo ciclo de processamento. A sinalização é realizada pela tarefa CONTROLE da própria instância envolvida.

Quando uma exceção E4 é sinalizada por uma instância, considera-se inicialmente a ocorrência de uma falta arbitrária. O gerenciador de faltas sinalizará, então, à tarefa CONTROLE dessa instância a necessidade da transferência de estado da instância par. Caso a exceção persista no próximo ciclo, inicia-se o processo de reconstrução do DRB. Se a exceção E4 for

sinalizada por ambas as instâncias, num mesmo ciclo de processamento, o DRB será bloqueado.

A ocorrência da exceção E4 numa instância pode ter como causa faltas transientes ou arbitrárias, enquanto a ocorrência simultânea nas duas instâncias poderá também ser fruto de faltas de modo comum a ambos os algoritmos alternativos ou de testes de aceitação mal dimensionados.

(c)*Exceção E5*

A ocorrência da exceção E5 está restrita a módulos FT que utilizam interface assíncrona, servindo para indicar descontinuidades nas filas de mensagens processadas pelas instâncias. A sinalização é efetuada pela própria instância envolvida, sempre que o número de ciclo de processamento associado ao número da mensagem consumida (dados de entrada) diferir do número esperado.

Se somente uma instância apresenta descontinuidade, esta será bloqueada logo após a execução da interação (1). Ao final do ciclo de processamento, a instância com número de ciclo correto enviará seu estado à instância bloqueada. Se ambas as instâncias apresentarem descontinuidade em suas filas de mensagens, há duas possibilidades : parar o processamento ou considerar como correto o menor número de ciclo, caso haja discordância. No segundo caso, a instância com número de ciclo de processamento maior (bloqueada) solicitará à instância par o envio do estado, ao final do ciclo, segundo o protocolo apresentado na Fig. 3.6.

Exceção	Tratamento
E1: Ausência de mensagem na interação 1 E2: Ausência de mensagem na interação 2 E3: Ausência de mensagem na interação 3	O gerenciador de faltas conta o número (N) de ocorrências consecutivas da exceção numa instância: Se N > NL, é assumida uma falta de "crash" na instância omissa (NL é um limite programado) e a estação deve ser reconfigurada (restauração do DRB).
E4: Ambos os algoritmos alternativos falham numa mesma instância	Há três possibilidades: a) E4 é sinalizado apenas por uma instância : é assumida uma falta arbitrária; o gerenciador de faltas solicita à instância correta que envie o estado à instância faltosa. b) E4 persiste no próximo ciclo; o gerenciador de faltas comanda uma restauração do DRB c) E4 é sinalizado por ambas as instâncias: o DRB é bloqueado (provavelmente houve uma falta de projeto no dimensionamento do teste de aceitação).
E5: DRB com interface assíncrona com descontinuidade no número de mensagens recebidas	Há duas possibilidades: a) Somente uma instância apresenta a descontinuidade; a instância que apresenta o numero correto é sinalizada a enviar seu estado à instância bloqueada, ao final do ciclo. b) As duas instâncias apresentam descontinuidade: caberá ao usuá-suário decidir sobre parar ou não o processamento. No segundo caso, se há discordância, o número de ciclo menor é considerado correto; o estado é transferido à outra instância, ao final do ciclo de processamento.
E1, E2, E3 : Ausência de mensagem nas interações (1), (2) e (3), num mesmo ciclo de processamento	Há duas possibilidades: a) Na primeira ocorrência, com interface síncrona: o estado e transferido à instância falha. b) Na segunda ocorrência em ciclos consecutivos: o DRB é restaurado.

Fig. 3.7 - Tabela de Exceções

Independentemente da ocorrência de exceções, ao final de cada ciclo de processamento, as instâncias FT sinalizam ao módulo gerenciador de faltas. Essa sinalização (ou sua ausência) irá auxiliar na adoção da ação de tratamento adequada a cada caso. Este aspecto é particularmente útil quando ocorrem combinações das exceções apresentadas e/ou estas ocorrem em ambas as instâncias FT.

Após a decisão por uma determinada ação de tratamento, o gerenciador de faltas sinalizará às instâncias FT. Essa sinalização, a ser recebida e processada pelas tarefas CONTROLE, ao final de cada ciclo de processamento, indicará um dos seguintes procedimentos :

(a)Prosseguir processamento normal;

(b)Solicitar estado à instância par;

(c)Transferir estado à instância par;

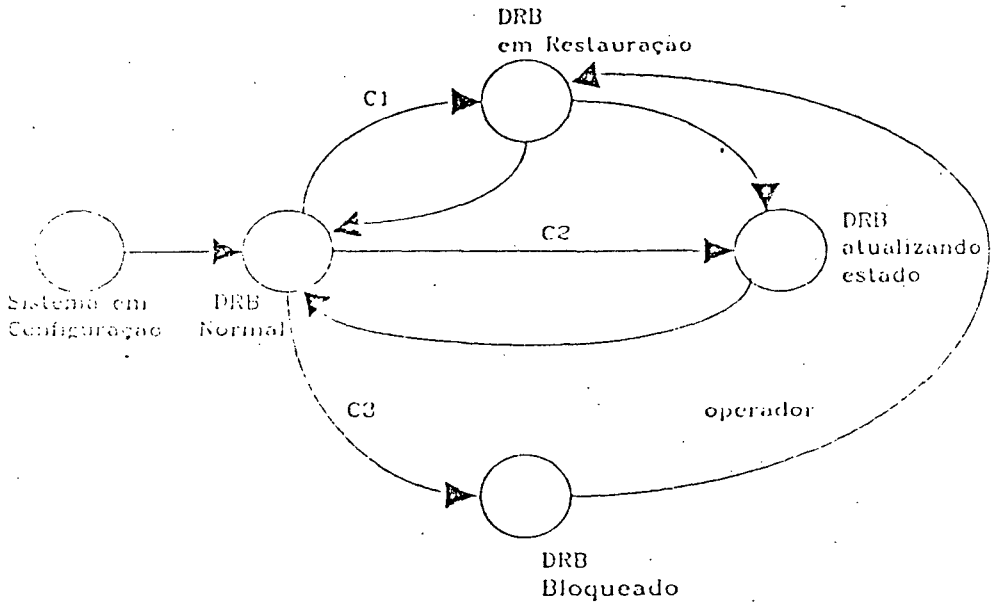
(d)Reconstrução do DRB;

(e) Parar processamento.

A sinalização indicada no item (d) informa à instância primária que o DRB encontra-se em processo de reconstrução, o que envolverá a configuração de uma nova instância secundária. Neste caso, até que venha a receber uma nova sinalização do gerenciador de faltas, indicando que o estado deve ser transferido, a instância primária prosseguirá sua execução, sem tentar se comunicar com a instância par.

Os estados possíveis do DRB e as condições para a transição entre esses estados são mostrados na Fig. 3.8, através de uma máquina de estados.

Deve ser observado que a ocorrência das exceções indicadas nas condições C1 e C2, em ambas as instâncias, indica dupla falta física, devendo o DRB ser bloqueado.



C1 ->
$$\left(\sum_{k=i}^{j} e_1(k) = N\right) \vee \left(\sum_{k=i}^{j} e_2(k) = N\right) \vee \left(\sum_{k=i}^{j} e_3(k) = N\right) \quad \text{onde } e_n(k)=1 \ \forall \ k \mid i <= k <= j$$
$$\vee$$
$$\sum_{k=i}^{i+1} e_4(k) = 2$$
$$\vee$$
$$\sum_{k=i}^{i+1} e_1(k) + \sum_{k=i}^{i+1} e_2(k) + \sum_{k=i}^{i+1} e_3(k) > 6$$
$$\vee$$
$$[e_5(k) = 1]_{\text{opcional}}$$

C2 ->
$$e_4(k) = 1$$
$$\vee$$
$$e_1(k) + e_2(k) + e_3(k) = 3$$
$$\vee$$
$$[e_5(k) = 1]_{\text{opcional}}$$

C3 ->
$$e_4(k) + \overline{e}_4(k) = 2 \quad \text{onde } \overline{e}_4 \text{ é sinalizada pela instância par}$$

OBS:

e (cp) é a função de ocorrência da exceção n por ciclo de processamento em uma instância de modulo e:

$$e_n(cp) = \begin{cases} 0 & \text{p/ não ocorrência de exceção} \\ 1 & \text{p/ ocorrência de exceção} \end{cases}$$

Fig. 3.8 - Máquina de Estados do DRB

3.5-Conclusão

Neste capítulo foram descritos os modelos de tolerância a faltas e de tratamento de faltas propostos, os quais visam a programação e o gerenciamento de redundâncias em sistemas distribuídos. A incorporação desses modelos à linguagem LIS permitiu a introdução de uma abordagem por linguagem para a implementação do DRB.

No próximo capítulo serão apresentados os principais aspectos da implementação dos modelos de tolerância a faltas e de tratamento de faltas na linguagem LIS. Além disto, serão discutidos os resultados obtidos.

CAPÍTULO 4

ASPECTOS DE IMPLEMENTAÇÃO E DISCUSSÃO DE RESULTADOS

4.1-Introdução

O modelo de tolerância a faltas e o modelo de tratamento de faltas, apresentados no capítulo anterior, foram implementados na linguagem LIS, seguindo o paradigma de programação dessa linguagem. Para obter o alto grau de transparência desejado dos algoritmos e mecanismos de tolerância a faltas, foram propostas extensões à linguagem.

Este capítulo tem por objetivo descrever os principais aspectos da implementação dos modelos de tolerância a faltas e de tratamento de faltas na linguagem LIS, bem como a discussão dos resultados obtidos. Inicialmente, serão apresentadas as principais características da linguagem LIS.

4.2-Linguagem LIS : Características Gerais

A Linguagem de Implementação de Sistemas - LIS [Fraga, 89] é parte integrante do Ambiente de Desenvolvimento e Execução de Software - ADES,

desenvolvido no LCMI-UFSC. Esta linguagem tem por objetivo facilitar a programação distribuída em aplicações de tempo real. Para tanto, a linguagem encapsula todos os aspectos da arquitetura do software, o que inclui suporte para concorrência, comunicação local e remota, além da configuração de sistemas. O modelo de programação da linguagem LIS constitui uma variação da abordagem do sistema CONIC [Magee, 87] para a construção de sistemas distribuídos.

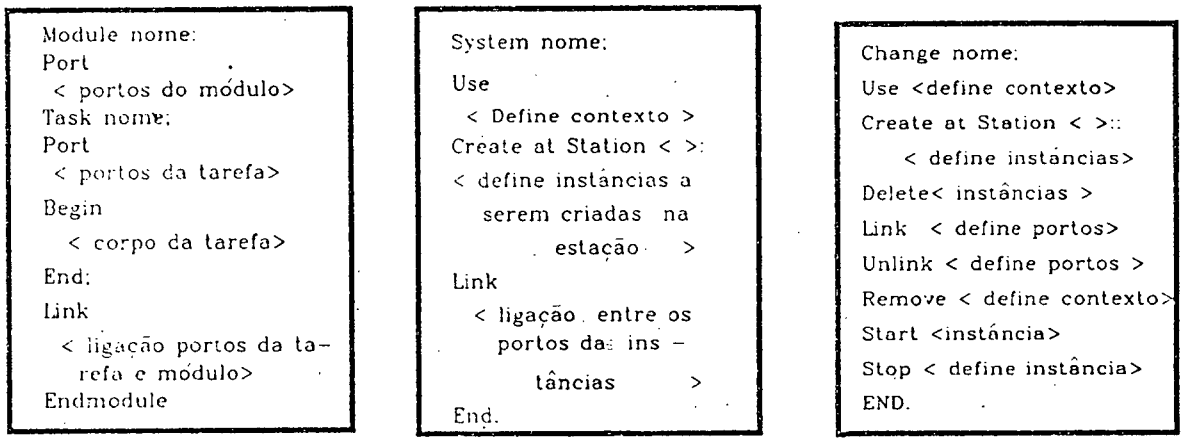
O paradigma de programação da linguagem LIS, conforme visto no capítulo anterior, segue o princípio da decomposição modular, separando as atividades de programação dos módulos das ações de configuração destes no sistema. Neste sentido, a linguagem LIS compõe-se de duas sub-linguagens : a Linguagem de Programação de Componentes Elementares - LINCE [Silva, 88] e a Linguagem de Configuração de Sistemas - LINCS [Souza, 88] e [Abreu, 91]. Ambas têm como suporte de execução o Núcleo de Tempo Real [Nacamura, 88].

Para implementar as abstrações de seu modelo de programação, a linguagem LIS dispõe de três declarações básicas, apresentadas a seguir :

(a)Declaração de Tipo Módulo :

O módulo, unidade básica de configuração, é instanciado a partir de um tipo módulo definido conforme mostrado na Fig. 4.1.a. As conexões internas, ligando portos de tarefa a portos de módulo ou de tarefa são especificadas no corpo do módulo. Essas ligações internas podem ser alteradas em tempo de execução por tarefas do próprio módulo, permitindo que o mesmo seja reconfigurado internamente. Os portos declarados no

módulo estão associados a um determinado tipo. A combinação das primitivas SEND/WAIT/RECEIVE/REPLY proporciona diferentes possibilidades em termos de canais de comunicação. Algumas dessas combinações podem ser vistas na Fig. 4.2.



(a) DECLARAÇÃO MÓDULO (LINCE) (b) DECLARAÇÃO SYSTEM (LINCS) (c) DECLARAÇÃO CHANGE (LINCS)

Fig. 4.1 - Declarações LIS

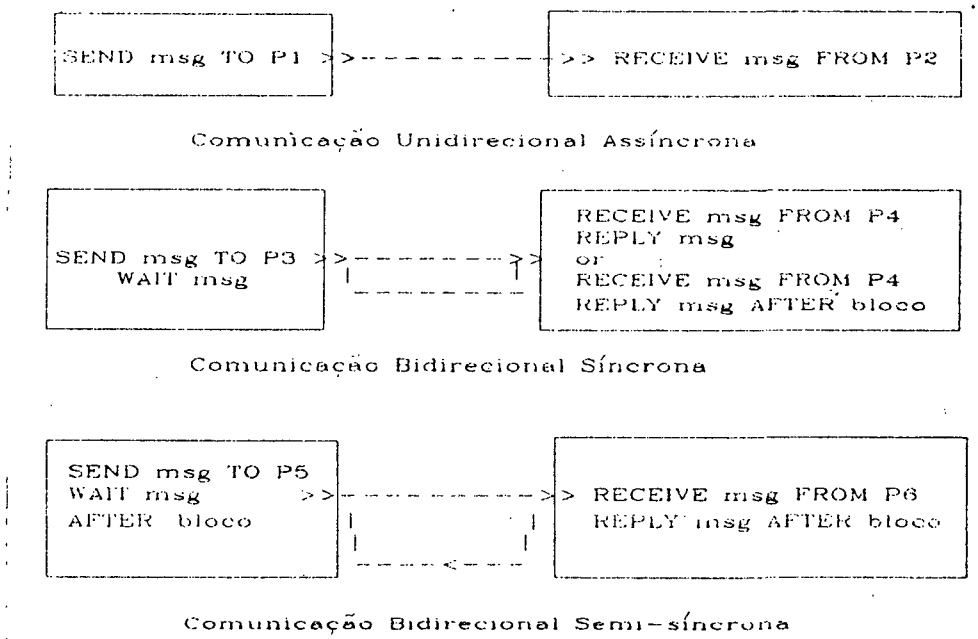


Fig. 4.2 - Tipos de Comunicação

(b)Declaração System :

A declaração SYSTEM define a configuração inicial do sistema (configuração estática). A configuração inicial deve incluir o suporte de tempo de execução, módulos do sistema operacional distribuído e módulos da aplicação. As declarações e os comandos da especificação SYSTEM permitem:

-a definição do contexto, isto é, da lista de tipos módulos que serão usados no programa distribuído;

-a criação de instâncias, nas diversas estações do sistema, a partir dos tipos módulos declarados;

-a interconexão dos portos de instâncias.

A sintaxe da declaração de especificação de sistema é mostrada na Fig. 4.1.b.

(c)Declaração Change :

A separação das atividades de programação em larga escala e pequena escala permite que a configuração do sistema seja alterada, durante a operação do mesmo - configuração dinâmica - [Kramer, 85].

A declaração CHANGE (Fig. 4.1.c) é similar à declaração SYSTEM, com o incremento de operações inversas. Estas operações inversas permitem que ligações sejam desfeitas, instâncias sejam destruídas e tipos módulos sejam removidos, possibilitando então a configuração dinâmica do sistema distribuído.

A execução da declaração CHANGE é controlada pelo módulo gerenciador de configuração, o qual está incluído no sistema operacional distribuído. As estratégias e o suporte de configuração dinâmica da linguagem LIS são abordados com maiores detalhes em [Abreu, 91].

4.3-Extensões à Linguagem LIS

A implementação dos modelos de tolerância a faltas e de tratamento de faltas na linguagem LIS foi desenvolvida visando obter simplicidade de programação, além de um alto grau de transparência, tanto em relação ao programador como em relação aos demais módulos do sistema. Desta forma, na programação de um módulo FT, o programador deverá incumbir-se apenas da implementação dos algoritmos da aplicação, da elaboração do teste de aceitação e da determinação das restrições de tempo associadas à execução dos algoritmos. Por outro lado, na especificação da configuração de um sistema não haverá distinções entre módulos FT e módulos comuns. O programador terá apenas que determinar a localização das instâncias FT, no caso de configuração explícita.

A linguagem LIS recebeu extensões a fim de permitir essa transparência na programação e configuração do módulo FT. As extensões propostas atingiram a ambas as sub-linguagens, LINCE e LINCS. A implementação das extensões deu-se com o auxílio das ferramentas YACC [Johnson, 78] e LEX [Lesk, 82].

4.3.1-Programação do Módulo FT

A sintaxe do módulo tolerante a faltas, codificada em linguagem LINCE é mostrada a seguir:

```
tipo_modulo_FT ::= MODULE "<" FT ">" ident [ "(" decl_par ")" ][ "<" heap ">"] ":"
                   [decl_use]
                   [decl_cte]
                   [dec_tipo]
                   [decl_estado]
                   [decl_subprogma]
                   [decl_porto]
                   [decl_Tarefa_A]
                   [decl_Tarefa_B]
                   [dec_Teste_Aceitacao]
                   [corpo_modulo]
                   [decl_link]
                   ENDMODULE"."
dec_Teste_Aceitacao ::= ACCEPTANCE TEST "[" tempo "." tempo "]"
                   BEGIN
                   lista_instrucao
                   END":"
      decl_estado ::= STATE lista_variavels ":"
```

A estrutura sintática de um módulo FT difere da mostrada na Fig. 4.1.a (módulo comum), basicamente, pela presença do símbolo terminal '<FT>' e pelas declarações de estado e de teste de aceitação, além da possibilidade de declaração de um corpo principal para o módulo.

Um módulo é identificado como tolerante a faltas através do símbolo terminal '<FT>'. Os portos declarados a nível de módulo, um único porto no

caso síncrono ou dois portos no caso assíncrono, representam os únicos pontos para o acesso externo às características de tolerância a faltas.

Num módulo FT são permitidas apenas duas tarefas, as quais representam os algoritmos da aplicação (tarefas A e B). Na declaração LINK, o(s) porto(s) de uma das tarefas é (são) associado(s) ao(s) porto(s) do módulo. Esta associação não será tratada pelo pré-compilador como uma conexão real, servindo somente para identificar o algoritmo preferencial, que será a tarefa ativa na instância primária.

As variáveis globais, que fazem parte do estado da instância FT, são declaradas em STATE, podendo ser inicializadas por código, no corpo principal do módulo.

Em ACCEPTANCE TEST é declarado o teste de aceitação ao qual os resultados dos algoritmos alternativos deverão ser submetidos. Os parâmetros declarados representam as restrições de tempo associadas à execução de cada algoritmo (tarefas A e B, respectivamente), isto é, o tempo máximo pelo qual as suas respostas permanecerão válidas, a partir do instante em que a tarefa correspondente recebeu os dados de entrada.

Mapeamento dos Módulos FT

O módulo FT, declarado da forma mostrada anteriormente, é mapeado pelo pré-compilador da linguagem LIS na estrutura de instância primária apresentada na Fig. 4.3. O teste de aceitação é inserido pelo pré-compilador na tarefa TEST I/O. O restante do código desta tarefa e a totalidade do

código da tarefa CONTROLE são também gerados automaticamente pelo pré-compilador.

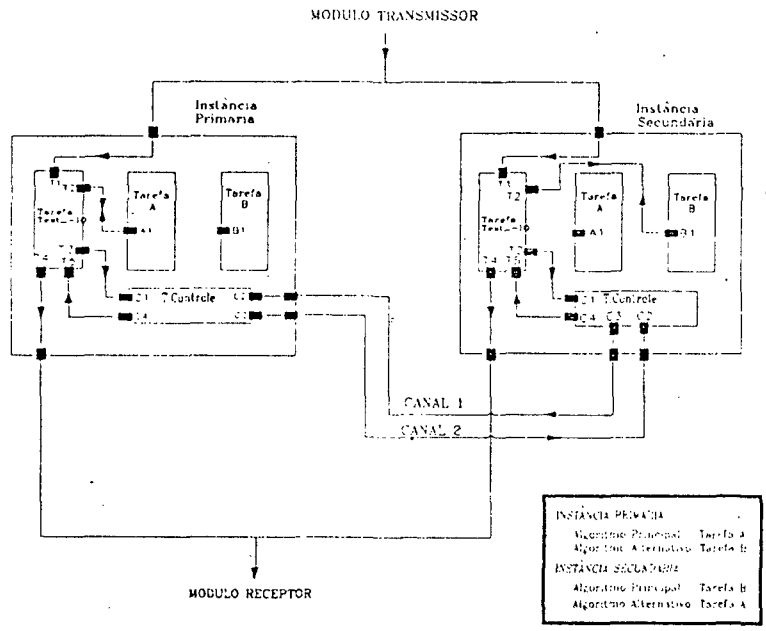


Fig. 4.3 - Modelo de Tolerância a Faltas

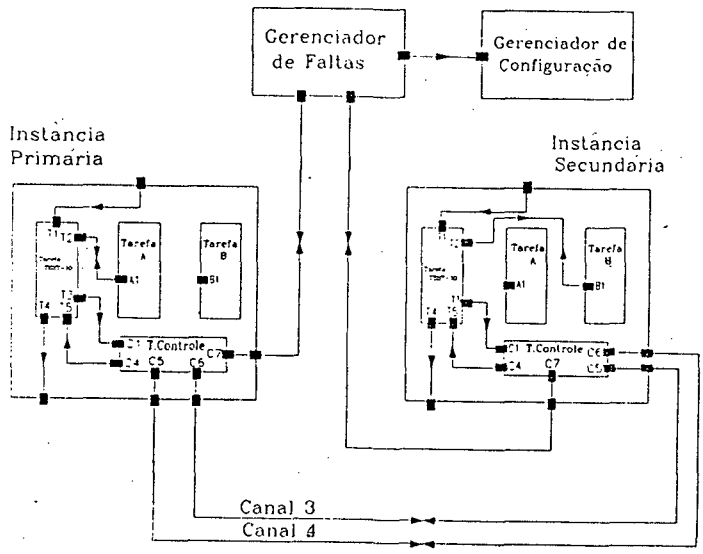


Fig. 4.4 - Modelo de Tratamento de Faltas

As variáveis globais declaradas em STATE são mapeadas em duas estruturas correspondentes a espaços distintos de memória : o estado temporário e o estado corrente. O estado temporário será utilizado como variável de trabalho pela instância durante cada ciclo de processamento. Ao início de cada ciclo de processamento, o estado temporário é copiado para a estrutura de estado corrente, o que significa o estabelecimento de um ponto de recuperação (salvar o estado da instância FT). Caso ocorra a reconfiguração interna da instância, num determinado ciclo, o estado corrente será copiado na estrutura de estado temporário, o que significa um retrocesso ao último ponto de recuperação (restauração do estado da instância FT). Deste modo, a nova tarefa ativa terá acesso ao estado anterior à execução do algoritmo que falhou. As operações de estabelecer um ponto de recuperação e restaurar o estado são realizadas através das primitivas do suporte de execução SAVE e RESTORE, respectivamente. A estrutura que contém o estado corrente é encapsulada em um registro que incorpora a fila de mensagens ainda não tratadas pelo porto de entrada do módulo, no caso de interface assíncrona. Este registro será enviado como uma mensagem, sempre que houver a solicitação do envio do estado à instância par (canais 3 e 4, Fig. 4.4), como parte de um processo de restauração do DRB, conforme descrito no capítulo anterior.

A partir das informações obtidas na declaração LINK é gerado automaticamente o código necessário para a conexão interna dos portos no módulo FT.

4.3.2-Configuração dos Módulos FT

Um módulo FT, ao ser configurado, deverá dar origem a duas instâncias, carregadas em diferentes estações de processamento. A criação das duas instâncias redundantes, estará, então, a cargo do aspecto configuração da linguagem LIS (sub-linguagem LINCS). Duas estratégias podem ser adotadas para a configuração de módulos FT :

-configuração explícita : o programador define explicitamente a localização das duas instâncias;

-configuração implícita : o programador define apenas a localização da instância primária, cabendo ao sistema a carga da instância secundária, de acordo com regras previamente definidas.

A indicação explícita das estações nas quais deverão ser carregadas as instâncias FT é adequada aos casos em que a localização das funções é relevante, isto é, onde a aplicação depende fortemente do ambiente externo. Nestes casos, o DRB precisa ser implantado em sítios específicos. Quanto à alocação implícita da instância secundária, várias regras podem ser implementadas, desde as mais simples como, por exemplo, carregar a instância secundária na estação seguinte àquela onde foi carregada a instância primária, até a adoção de critérios baseados na distribuição de carga do sistema.

Para possibilitar a configuração dos módulos FT, mantendo um nível elevado de transparência, algumas extensões foram propostas, no âmbito da

sub-linguagem LINCS. Em termos da configuração estática (declaração SYSTEM), as extensões tratam, essencialmente, dos seguintes aspectos :

-identificação dos módulos FT presentes na configuração;

-criação de parâmetros implícitos;

-criação das instâncias FT (configurações explícita e implícita);

-ligação dos portos de controle das instâncias FT e ligações destas com módulos externos ao DRB.

Identificação de Módulos FT

A identificação de um módulo como tolerante a faltas ou não, por parte do configurador, se dá através da verificação do campo correspondente no bloco descritor de módulo. Este bloco faz parte da base de dados representativa do estado de configuração do sistema, presente no configurador [Abreu, 91]. O preenchimento desse campo está a cargo do pré-compilador LINCE. O fato de um módulo ser tolerante a faltas exigirá do configurador um tratamento especial, conforme será descrito a seguir.

Criação de Parâmetros Implícitos

A um módulo FT estão associados dois parâmetros implícitos, que deverão ser passados às instâncias redundantes do modelo : a função da instância no DRB (primária ou secundária) e o índice dessa instância na estação onde será configurada. Ambos os parâmetros - "instância" e "índice" - serão tratados pelas tarefas CONTROLE de cada instância.

O resultado da compilação de um tipo módulo FT é um módulo cuja configuração interna corresponde à configuração de instância primária. Portanto, ao iniciar sua execução, cada instância deve ter conhecimento de sua função inicial. A partir do parâmetro "instância", a tarefa CONTROLE procederá, caso seja necessário (por se tratar de instância secundária), uma reconfiguração interna, antes do início da execução do DRB. O parâmetro "índice" é necessário para a execução das operações associadas à reconfiguração interna (religação de portos).

Criação das Instâncias FT

Conforme discutido anteriormente, a configuração das instâncias FT pode ser realizada de forma explícita ou implícita :

-Configuração Explícita : neste caso, o programador irá declarar ambas as instâncias, que deverão receber nomes semelhantes, diferindo apenas no último caractere (inst1 e inst2, por exemplo). O caractere final ('1' para a instância primária e '2' para a instância secundária) servirá para identificar a função de cada instância, permitindo ao configurador o envio correto do parâmetro implícito correspondente.

-Configuração Implícita : neste caso, o programador irá declarar apenas a instância primária. Foi implementada a estratégia de carregar, de forma automática, a instância secundária na estação com maior disponibilidade de memória dentre as estações presentes

na configuração (excetuando aquela onde foi carregada a instância primária).

Ligação de Portos

As ligações entre os portos de controle das duas instâncias, estabelecendo os canais de comunicação 1 a 4 (Fig. 4.3 e 4.4) são realizadas automaticamente. Quanto às ligações do DRB com módulos externos, o programador deverá apenas especificar as ligações relativas à instância primária. O configurador replicará essas ligações, considerando os portos correspondentes da instância secundária.

4.4-Sistema de Numeração de Mensagens

A necessidade de se adotar um sistema de numeração de mensagens, para dar suporte às comunicações envolvendo as instâncias do DRB e módulos externos, pode ser evidenciada pelos seguintes fatos :

-ambas as instâncias FT devem ter acesso aos mesmos dados de entrada;

-no caso de envio de mensagens redundantes, por parte das instâncias FT, uma dessas mensagens deverá ser descartada.

A comunicação remota no ambiente ADES é realizada com o auxílio do grupo módulo servidor de rede, que faz parte do suporte de tempo de execução [Souza, 88]. Este grupo módulo recebeu extensões, visando implementar a numeração de mensagens.

Servidor de Rede

O grupo módulo servidor de rede realiza a interface de comunicação entre as estações do sistema, permitindo que a comunicação remota ocorra de forma transparente quanto à distribuição dos módulos no sistema. Dentre os módulos pertencentes ao grupo destacam-se o *módulo servidor de envio (servenv)* e o *módulo servidor de recepção (servrec)*, encarregados, respectivamente, do envio e da recepção de mensagens remotas. A estrutura do servidor de rede é mostrada, de forma simplificada, na Fig. 4.5.

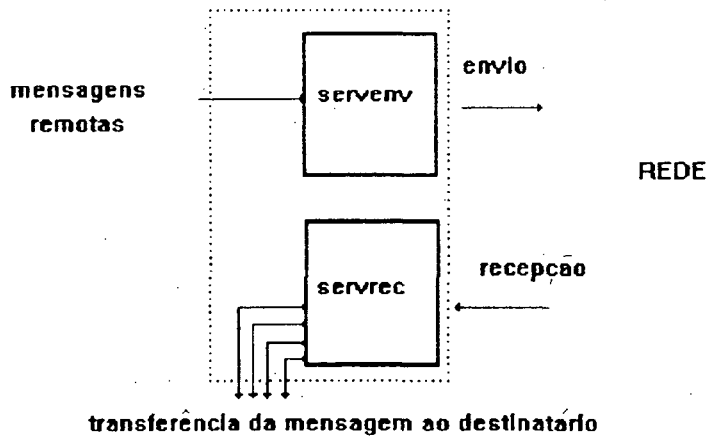


Fig. 4.5 - Servidor de Rede

Numa especificação de sistema, sempre que houver comunicação remota, o grupo módulo servidor de rede (gserv.grp) deverá ser declarado

entre os módulos do contexto e instanciado nas estações envolvidas com tal tipo de comunicação.

Durante o processo de configuração, a partir das ligações declaradas na especificação SYSTEM, a presença de comunicação remota é detectada pelas primitivas de ligação de portos do Núcleo de Tempo Real, mediante a comparação dos índices de estação associados aos portos de saída e entrada. Caso estes índices sejam diferentes, o que indica comunicação remota, o porto de saída será ligado ao porto de entrada da instância local do servidor de rede (módulo servidor de envio).

A recepção de mensagens remotas se dá através de uma das instâncias do servidor de recepção, a qual está associada a um nível de interrupção. A chamada ao servidor de recepção ocorre pela ativação da interrupção correspondente, por uma primitiva do núcleo. O servidor de recepção irá conectar-se ao porto destino e transferir a mensagem. No caso síncrono, o servidor de recepção aguardará é procederá o envio da mensagem de resposta. Desta forma, para possibilitar a recepção de novas mensagens pela estação, mesmo durante o bloqueio provocado por uma recepção síncrona, o grupo servidor de rede é formado por várias instâncias do módulo servidor de recepção [Souza, 88].

Extensões ao Servidor de Rede

A fim de permitir a numeração de mensagens, foram propostas extensões ao esquema do servidor de rede apresentado anteriormente. Estas extensões foram implementadas através dos módulos *nservenv* e *nservrec*, esquematizados na Fig. 4.6.

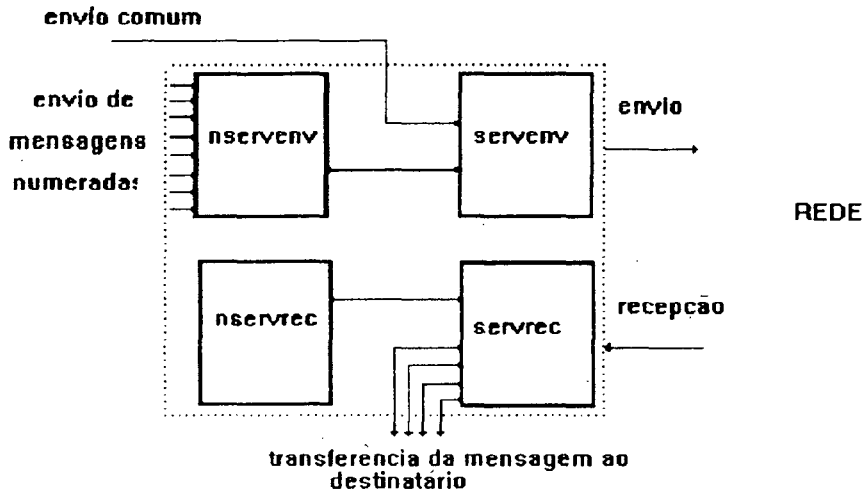


Fig. 4.6 - Servidor de Rede com extensões

(a)Envio de mensagens numeradas :

O servidor de rede com extensões permite dois tipos de envio : o envio comum e o envio com numeração. O envio comum processa-se como visto anteriormente. No envio com numeração, a mensagem é inicialmente tratada pelo módulo *nservenv*. Este módulo acrescenta à mensagem um campo, que representará o seu número. A seguir, a mensagem numerada é enviada ao módulo servidor de envio e daí transferida normalmente à estação destinatária. Para o módulo servidor de envio não há distinção entre mensagens numeradas e mensagens comuns.

O envio de mensagens numeradas está reservado aos casos onde o módulo destinatário de uma comunicação remota é um módulo FT. Esta informação é obtida pelas primitivas de ligação de portos do Núcleo de

Tempo Real, a partir de um campo da tabela de configuração gerada quando o sistema é configurado. Deste modo, caso se trate de comunicação remota e o módulo destinatário seja FT, o porto de saída do módulo emissor deverá ser ligado a um dos oito portos do módulo *nservenv*.

A existência de vários pontos de acesso ao módulo *nservenv* permite que sejam criadas várias seqüências de numeração, possibilitando a comunicação com múltiplos DRB. O campo relativo ao número de uma mensagem é formado pelo número da seqüência e pelo número da mensagem propriamente dito. O número de seqüência é um inteiro, associado ao índice do porto de entrada do módulo *nservenv*. O número da mensagem é um número inteiro entre "0" e "9". Ao atingir o número máximo, a numeração é reciclada.

A recepção das mensagens numeradas, efetuada nas estações remotas onde se encontram as instâncias FT, ocorre do mesmo modo que a recepção de mensagens comuns, já que à tarefa TEST_I/O de cada instância cabe tratar com os aspectos da numeração durante o ciclo de processamento correspondente.

(b)Recepção de mensagens numeradas :

As mensagens de resposta provenientes das instâncias FT são enviadas pelo servidor de rede como mensagens comuns, embora sejam implicitamente numeradas. Ao descritor de tais mensagens é associado um campo indicando a numeração implícita.

A recepção dessas mensagens requer um tratamento especial. Por esta razão, o módulo servidor de recepção recebeu uma modificação, de modo que, sempre que receber uma mensagem, deverá realizar o teste do campo correspondente à numeração implícita. Caso o teste resulte positivo, o que indica que o módulo origem é FT, a mensagem será enviada ao módulo *nservrec*.

O módulo *nservrec* verificará a numeração da mensagem, retirando-a e descartando as mensagens repetidas. As mensagens não repetidas, já sem numeração, serão retornadas ao módulo servidor de recepção, para que sejam entregues ao módulo destinatário da comunicação remota.

4.5-Discussão de Resultados

Neste item serão apresentados os resultados mais relevantes obtidos na implementação do DRB através da linguagem LIS, bem como serão descritos os testes realizados. Esses resultados serão discutidos no âmbito dos modelos de tolerância e de tratamento de faltas.

4.5.1-Modelo de Tolerância a Faltas

A primeira etapa da implementação do modelo de tolerância a faltas na linguagem LIS consistiu na realização de um protótipo, a partir de uma versão não estendida da linguagem. O objetivo desta etapa foi obter uma primeira avaliação do comportamento do modelo DRB num ambiente

distribuído fracamente acoplado, bem como testar os algoritmos das tarefas TEST_I/O e CONTROLE.

Numa segunda etapa, foram introduzidas na linguagem LIS as características de transparência na programação e configuração de módulos FT, implementadas através das extensões descritas anteriormente.

Para verificar seu comportamento sob diversas condições simuladas de falta, o modelo foi submetido a uma série de testes. O esquema básico de testes utilizado é mostrado na Fig. 4.7. Neste esquema, o módulo FT realiza um cálculo simples, a partir dos dados de entrada supridos pelo módulo EMISSOR, sendo os resultados do processamento entregues ao módulo RECEPTOR. Testes semelhantes foram realizados considerando uma interface síncrona para o DRB. A inclusão de situações simuladas de falta deu-se através dos seguintes procedimentos :

-perda de determinadas mensagens, tanto em canais internos como externos às instâncias FT;

-injeção de faltas de projeto nos algoritmos alternativos;

-dimensionamento incoerente de testes de aceitação;

-interrupções na comunicação entre instâncias FT;

-desligamento de estações de processamento, simulando faltas físicas.

A partir da utilização dos procedimentos citados e da combinação destes, um grande número de casos foi elaborado e testado. Os resultados obtidos foram satisfatórios, tendo o modelo de tolerância a faltas se comportado de forma compatível com o que foi descrito no capítulo anterior (seção 3.3.3).

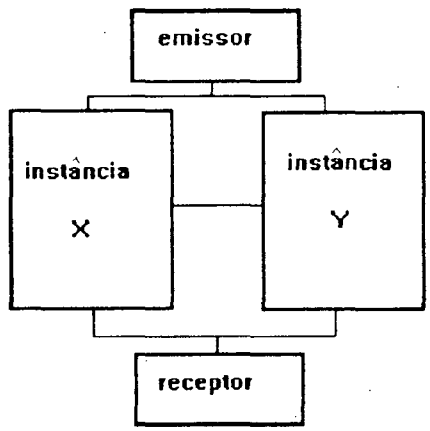


Fig. 4.7 - Esquema de Testes do Modelo de Tolerância a Faltas

4.5.2-Modelo de Tratamento de Faltas

O passo inicial para a implementação do modelo de tratamento de faltas foi a concepção e o desenvolvimento do módulo gerenciador de faltas.

A validação do modelo de tratamento de faltas foi realizada de forma não exaustiva, através de um exemplo que será descrito a seguir. Este exemplo tem por objetivo oferecer um cenário, onde várias situações de falta são simuladas, permitindo avaliar o comportamento do modelo. A estrutura do exemplo implementado é mostrada na Fig. 4.8. Além das instâncias FT e do módulo gerenciador de faltas, estão presentes os módulos EMISSOR, RECEPTOR e MEIO.

Os algoritmos de aplicação implementados pelas instâncias FT são extremamente simples, realizando a multiplicação de números inteiros, cujos operandos são recebidos como mensagem de entrada. Tendo em vista os objetivos do exemplo, estas aplicações simples mostram-se adequadas, muito embora, obviamente, problemas dessa natureza não justificariam a adoção de técnicas de tolerância a faltas.

O módulo EMISSOR tem por função o envio das mensagens de entrada às instâncias FT. O módulo RECEPTOR recebe, a cada ciclo, os resultados do processamento do DRB e uma sinalização proveniente do módulo gerenciador de faltas, indicando quais exceções ocorreram e que ações de tratamento foram executadas. Estas informações são apresentadas na tela.

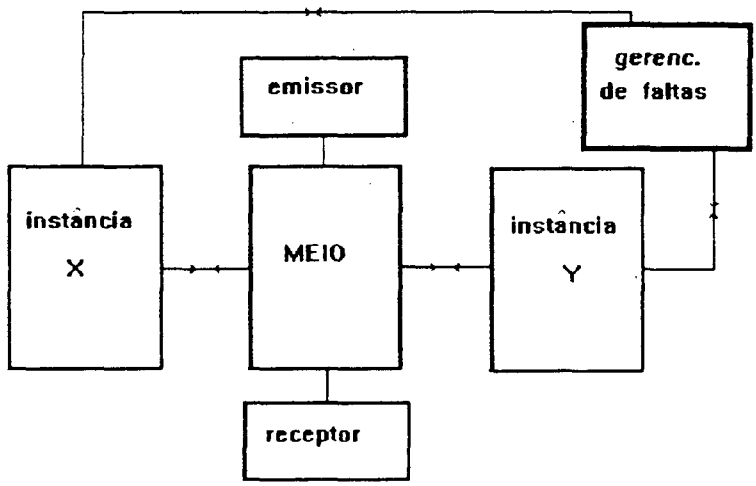


Fig. 4.8 - Esquema de Teste do Modelo de Tratamento de Faltas

A criação do cenário de faltas é centralizada pelo módulo MEIO, no qual são geradas as situações que simulam essas faltas. O módulo MEIO simula o suporte de comunicação em um sistema distribuído, atuando como intermediário nas comunicações entre as instâncias FT e entre estas e os módulos EMISSOR e RECEPTOR. O algoritmo do módulo MEIO envolve, basicamente :

(a)Recepção das mensagens de entrada, provenientes do módulo EMISSOR;

(b)Envio das mensagens de entrada às instâncias FT;

(c)Recepção/envio das mensagens de troca de informações entre as instâncias (interações 1, 2 e 3, apresentadas no capítulo anterior);

(d)Recepção dos resultados do processamento provenientes das instâncias FT;

(e)Envio dos resultados ao módulo RECEPTOR.

Nos pontos (b) e (c) podem ser simuladas várias situações de falta, que causarão a sinalização de exceções. Foram elaboradas 14 situações, procurando abranger todo o escopo das possíveis faltas e propiciar a ocorrência de todas as exceções apresentadas no capítulo anterior e ainda de combinações destas. As situações de falta implementadas, juntamente com as correspondentes exceções ocasionadas e respectivas ações de tratamento (expressas através das sinalizações às instâncias FT) são discutidas a seguir.

Cenários de Faltas

(a)Cenário 1:

situação de falta : a mensagem de entrada é enviada apenas a uma instância (por exemplo, instância X), num determinado ciclo.

exceções sinalizadas : no ciclo de processamento considerado, a instância X sinalizará as exceções E1, E2 e E3. No ciclo seguinte, a instância X sinalizará as exceções E1, E2 e E3, enquanto a instância Y sinalizará a exceção E5.

sinalizações às instâncias : no ciclo considerado, o gerenciador de faltas apenas contabilizará as exceções sinalizadas e sinalizará que o processamento deve prosseguir. No ciclo seguinte, a

instância X será sinalizada a transferir o estado e a instância Y, a solicitar a transferência de estado.

(b)Cenário 2 :

situação de falta : idem ao caso anterior, por dois ciclos consecutivos.

exceções sinalizadas : nos dois ciclos considerados, a instância X sinalizará as exceções E1, E2 e E3.

sinalizações às instâncias : no primeiro ciclo considerado, o gerenciador de faltas irá contabilizar as exceções sinalizadas. No ciclo seguinte, a instância X receberá a sinalização de que o DRB está em reconstrução, enquanto a instância Y será sinalizada a parar o processamento.

(c)Cenário 3 :

situação de falta : a mensagem de entrada deixa de ser enviada a ambas as instâncias FT, num determinado ciclo de processamento.

exceções sinalizadas : no ciclo seguinte, as duas instâncias sinalizarão a exceção E5.

sinalizações às instâncias : o gerenciador de faltas sinalizará que as instâncias FT devem prosseguir o seu processamento.

(d)Cenário 4 :

situação de falta : as mensagens de entrada endereçadas às instâncias FT são perdidas alternadamente, isto é, no ciclo *n* não é enviada a mensagem de entrada à instância X , no ciclo *n + 1*

não é enviada a mensagem de entrada à instância Y, e assim, sucessivamente.

exceções sinalizadas : no primeiro ciclo considerado, a instância Y irá sinalizar as exceções E1, E2 e E3. No ciclo seguinte, a instância X sinalizará as exceções E1, E2, E3 e E5. Nos ciclos subsequentes, essas quatro exceções serão sinalizadas pelas instâncias Y e X, de forma alternada.

sinalizações às instâncias : no primeiro ciclo considerado, o gerenciador de faltas contabilizará as exceções sinalizadas e indicará que as instâncias deverão continuar o seu processamento. No ciclo seguinte, em função da ocorrência da exceção E5, a instância X será sinalizada a solicitar a transferência de estado e a instância Y, a transferir o estado. Nos próximos ciclos, essas sinalizações prosseguirão, de forma alternada. Entretanto, a quarta ocorrência consecutiva do grupo de exceções (duas vezes para cada instância, em ciclos alternados) levará o gerenciador de faltas a sinalizar a ambas as instâncias que parem seu processamento.

(e)Cenário 5 :

situação de falta : uma mensagem de entrada sem sentido é enviada a uma das instâncias FT (por exemplo, instância Y), num determinado ciclo.

exceções sinalizadas : a situação de falta descrita acima simula uma falta arbitrária. A existência de dados de entrada sem sentido será detectada pelo teste de aceitação, levando ambos os

algoritmos alternativos da instância a falhar. Portanto, a instância Y irá sinalizar a exceção E4.

sinalizações às instâncias : a instância X será sinalizada a transferir seu estado e a instância Y será sinalizada a solicitar a transferência de estado.

(f)Cenário 6 :

situação de falta : idem a situação anterior, em dois ciclos consecutivos.

exceções sinalizadas : nos dois ciclos, a instância Y sinalizará a exceção E4.

sinalizações às instâncias : no primeiro ciclo considerado, a instância X será sinalizada a transferir seu estado e a instância Y será sinalizada a solicitar a transferência de estado. No ciclo seguinte, a instância X receberá sinalização indicando que o DRB está sendo reconstruído, enquanto a instância Y será sinalizada a parar seu processamento.

(g)Cenário 7 :

situação de falta : uma mensagem de entrada sem sentido é enviada, num determinado ciclo, a ambas as instâncias FT.

exceções sinalizadas : as duas instâncias irão sinalizar a exceção E4.

sinalizações às instâncias : as instâncias X e Y serão sinalizadas a parar seus processamentos.

(h)Cenário 8 :

situação de falta : a mensagem de entrada é enviada a uma das instâncias FT (por exemplo, X), num determinado ciclo, com um número de ciclo incorreto (mensagem repetida).

exceções sinalizadas : a instância X sinalizará a exceção E5, enquanto a instância Y sinalizará as exceções E1, E2 e E3.

sinalizações às instâncias : a instância X será sinalizada a solicitar a transferência de estado. A instância Y será sinalizada a transferir seu estado à instância par.

(i)Cenário 9 :

situação de falta : as mensagens relativas às interações 1, 2 e 3, provenientes de uma das instâncias FT (por exemplo, X) são perdidas num determinado ciclo.

exceções sinalizadas : a instância Y sinalizará as exceções E1, E2 e E3.

sinalizações às instâncias : as exceções serão contabilizadas e as instâncias, sinalizadas a prosseguir seu processamento normal.

(j)Cenário 10 :

situação de falta : idem ao caso anterior, em dois ciclos consecutivos.

exceções sinalizadas : nos dois ciclos, a instância Y irá sinalizar as exceções E1, E2 e E3.

sinalizações às instâncias : no segundo ciclo considerado, a instância X será sinalizada a parar seu processamento, enquanto a instância Y será comunicada de que o DRB será reconstruído.

(k)Cenário 11 :

situação de falta : as mensagens relativas às interações 1, 2 e 3 são perdidas em ambos os sentidos, num determinado ciclo de processamento.

exceções sinalizadas : ambas as instâncias sinalizarão as exceções E1, E2 e E3.

sinalizações às instâncias : as exceções serão contabilizadas e as instâncias FT serão sinalizadas a continuar seu processamento.

(l)Cenário 12 :

situação de falta : idem ao caso anterior, por dois ciclos consecutivos.

exceções sinalizadas : ambas as instâncias irão sinalizar, nos dois ciclos considerados, as exceções E1, E2 e E3.

sinalizações às instâncias : no segundo ciclo considerado, ambas as instâncias FT serão sinalizadas a parar seu processamento.

(m)Cenário 13 :

situação de falta : as mensagens enviadas por uma das instâncias FT (por exemplo, Y), referentes à interação 1 são perdidas.

exceções sinalizadas : a instância X irá sinalizar a exceção E1.

sinalizações às instâncias : antes de ser atingido o limite de ocorrências consecutivas permitido, o gerenciador de faltas apenas contabilizará essas ocorrências e sinalizará a continuidade do processamento normal do DRB. Quando o limite (adotado como cinco ocorrências) for atingido, a instância X será informada que

o DRB iniciará um processo de reconstrução e a instância Y será sinalizada a parar seu processamento.

(n)Cenário 14 :

situação de falta : idem ao caso anterior, considerando as mensagens emitidas pelas duas instâncias FT.

exceções sinalizadas : ambas as instâncias FT sinalizarão a exceção E1.

sinalizações às instâncias : antes de ser atingido o valor limite, o procedimento será idêntico ao caso anterior. Quando o limite for atingido, as duas instâncias serão sinalizadas a parar seu processamento.

O comportamento descrito pelos cenários 13 e 14 será semelhante se forem consideradas as mensagens referentes às interações 2 e 3 (exceções E2 e E3, respectivamente).

Quanto às ações de restauração do DRB, a transferência de estado já se encontra implementada. O próximo passo deverá ser a implementação do processo de reconstrução do DRB. Para isto, deverá ser realizada a integração entre o modelo de tratamento de faltas e o suporte de configuração dinâmica da linguagem LIS [Abreu, 91].

O processo de restauração do DRB, quando da necessidade de reconstrução de estações faltosas, certamente irá se constituir num processo lento, em termos de aplicações de tempo real. Contudo, pode-se afirmar que a aplicação em si não sofrerá grandes prejuízos, já que a execução da

instância primária ocorrerá de forma praticamente independente desse processo. Somente na fase final, a transferência de estado, é que a instância primária será envolvida.

O grau de interferência do processo de transferência de estado na execução da instância primária está ligado essencialmente à quantidade e ao tamanho das variáveis globais e das filas de mensagens armazenadas em buffers (caso assíncrono).

4.6-Comentários Adicionais

O conceito de Blocos de Recuperação Distribuídos (DRB) propõe uma abordagem uniforme para o tratamento de faltas de hardware e de software, através da utilização combinada de redundâncias ativas e passivas.

As implementações de DRB que têm sido propostas na literatura restringem-se a sistemas distribuídos fortemente acoplados. Em tais sistemas, o compartilhamento de memória proporciona um alto desempenho nas trocas de informações entre as redundâncias ativas. Outro aspecto a ser ressaltado é que as estações que executam o DRB não atuam de forma transparente às demais estações do sistema, de modo que parte dos procedimentos associados ao modelo fica a cargo das estações predecessora e sucessora. Portanto, as implementações propostas por Kim credenciam o modelo, por suas características de desempenho, a ser utilizado em aplicações onde estão presentes severas restrições de tempo (hard real time), tornando-o, porém,

dependente do hardware. Além disto, o envolvimento de outras estações torna a programação de redundâncias complexa e pouco flexível.

A proposta deste trabalho visa ampliar a abordagem de DRB, tornando o modelo independente do hardware e estendendo-o a sistemas distribuídos fracamente acoplados. Para dar suporte a essa implementação de DRB foi utilizada uma linguagem de programação - linguagem LIS. A abordagem por linguagem simplifica e introduz maior flexibilidade à programação de redundâncias. Além disto, a linguagem utilizada como base apresenta características que favorecem a adoção de técnicas de tolerância a faltas. Dentre essas características podem ser apontadas a decomposição modular (separação dos aspectos programação dos módulos e configuração do sistema), a configuração dinâmica e a possibilidade de reconfiguração interna dos módulos em tempo de execução. Na presente proposta, sob o ponto de vista dos demais módulos do sistema, não há distinções entre módulos FT e módulos comuns.

Outro aspecto considerável nesta proposta é o envolvimento mínimo do programador com as funcionalidades do modelo de tolerância a faltas, supridas através de declarações da linguagem LIS. Esses requisitos de flexibilidade e transparência favorecem também a manutenção do sistema (restituição de redundâncias). Em termos de desempenho, considerando sistemas fracamente acoplados, a velocidade das trocas de informações entre as redundâncias ativas é fortemente influenciada pelo suporte de comunicação. Desta forma, a implementação de DRB aqui proposta mostra-se mais adequada a sistemas de tempo real onde as restrições de tempo são menos rígidas (soft real time).

4.7-Conclusão

A linguagem LIS possui características que favorecem a adoção de técnicas de tolerância a faltas. Com o objetivo de assegurar um grau elevado de transparência na programação e gerenciamento de redundâncias foram propostas extensões à linguagem. As principais características da linguagem LIS e os aspectos mais representativos acerca da implementação do DRB nessa linguagem foram apresentados neste capítulo.

Foram também destacadas as principais características do sistema de numeração de mensagens, proposto com o intuito de possibilitar a consistência das comunicações do DRB com módulos externos. Finalmente, foram apresentados e discutidos os principais resultados obtidos, no âmbito dos modelos de tolerância a faltas e de tratamento de faltas.

No capítulo seguinte, serão apresentadas as conclusões finais deste trabalho.

CAPÍTULO 5

CONCLUSÃO

Neste trabalho são propostos modelos para a programação e o gerenciamento de redundâncias em sistemas distribuídos de tempo real. A proposta apresentada pode ser sintetizada em dois modelos : o modelo de tolerância a faltas e o modelo de tratamento de faltas. O modelo de tolerância a faltas trata dos domínios da detecção e recuperação de erros, enquanto o modelo de tratamento de faltas está relacionado à restituição de redundâncias perdidas durante a operação do sistema.

Ambos os modelos foram implementados na linguagem LIS, do ambiente ADES, seguindo o paradigma de programação desta linguagem. O paradigma de programação da linguagem LIS, fundamentado no princípio da decomposição modular, favorece a adoção de técnicas de tolerância a faltas. Essa característica, aliada à possibilidade de configuração dinâmica do sistema e de reconfiguração interna dos módulos introduz uma maior flexibilidade e simplifica a programação e a manutenção de redundâncias. Para dar suporte ao conceito de DRB, foram propostas extensões à linguagem LIS. Essas extensões tiveram por objetivo obter um alto grau de transparência na programação e na configuração dos módulos tolerantes a faltas .

Embora a implementação proposta tenha sido baseada em sistemas distribuídos fracamente acoplados, cujo desempenho é significativamente limitado pelo suporte de comunicação, o modelo proposto é independente do hardware. Deste modo, poderia ser utilizado tanto em sistemas distribuídos com fraco acoplamento como em sistemas fortemente acoplados.

A validação dos modelos de tolerância a faltas e de tratamento de faltas propostos, realizada através de testes que simulavam determinadas condições relevantes de falta, pode ser considerada satisfatória, tendo em vista os objetivos definidos para este trabalho. Uma validação completa dos modelos poderia ser obtida através da sua utilização em uma aplicação real. Pode ser também realizado um estudo analítico dos modelos, visando obter resultados qualitativos acerca de seu comportamento.

A adoção de técnicas de tolerância a faltas (inclusão de redundâncias) sempre acarreta custos, em termos do desempenho da aplicação. Para avaliar esses custos, há a necessidade de que sejam realizados testes de desempenho, comparando a execução remota de um algoritmo sem redundâncias e do mesmo algoritmo como parte de um DRB. Muito embora algumas medidas tenham sido realizadas, não houve, no presente trabalho, um estudo sistemático desta questão.

Pode ser apontada como próxima etapa no desenvolvimento deste trabalho, a implementação em versão final do modelo de tratamento de faltas, mediante a integração ao suporte de configuração dinâmica da linguagem LIS.

BIBLIOGRAFIA

[Abreu, 91] W. Abreu, J.S. Fraga e E.S. Silva, "Configuração Dinâmica no Sistema ADES", 9o. Simpósio Brasileiro de Redes de Computadores, pág. 247-258, Florianópolis, 1991.

[Anderson, 81] T.Anderson, P.A.Lee, "Fault Tolerance, Principles and Practice", Prentice-Hall International, London, 1981.

[Anderson , 83] T.Anderson e J.C.Knight, "A Framework for Software Fault Tolerance in Real-Time Systems", IEEE Trans Software Eng., vol SE-9, no. 3, pp. 355-364, May 1983.

[Avizienis, 84] A. Avizienis and J.P. Kelly, "Fault Tolerance by Design Diversity : Concepts and Experiments", IEEE Computer, vol. 17, no. 8, pp. 67-80, August 1984.

[Avizienis, 85] A. Avizienis, "The N-Version Approach to Fault-Tolerant Systems", IEEE Trans. on Software Eng., vol. SE-11, no. 12, pp. 1491-1501, December 1985.

[De Remer, 76] F. De Remer and H. Kron, "Programming-in-the-Large versus Programming-in-the-Small", IEEE Trans. on Soft. Eng., vol. SE-2, no. 2, pp. 80-86, June 1976.

[Fraga, 89] J.S. Fraga, J.M. Farines, W.M. Abreu, E.S. Silva, L. Nacamura Jr. e O. Coelho Fo., "ADES: Ambiente de Desenvolvimento e Execução de Software Distribuído", Actes du Séminaire Franco-Brésilien de Systèmes Informatiques Répartis, Florianópolis - SC - Brazil, September 1989.

[Hecht, 76] H. Hecht, "Fault-Tolerant Software for Real-Time Applications", Computing Surveys, pp. 391-407, December 1976.

[Hecht, 86] H. Hecht and M. Hecht, "Fault Tolerant Software", Fault-Tolerant Computing: Theory and Techniques. Ed. D.K.Pradhan, Prentice-Hall International, 1986.

[Horning, 74] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith and B. Randell, "A Program Structure for Error Detection and Recovery", Lecture Notes in Computer Science, vol. 16, pp. 171-187, NY Springer - Verlag, 1974.

[Johnson, 78] S.C. Johnson, "YACC : Another Compiler Compiler", Bell Laboratories, July 1978.

[Kim, 82] K.H.Kim, "Approach to mechanization of the conversation scheme based on monitor", IEEE Trans. Software Eng., vol. SE-8, no. 3, pp 189-197, May 1982.

[Kim, 84] K.H. Kim, "Distributed Execution of Recovery Blocks: an Approach to Uniform Treatment of Hardware and Software Faults", Proc. 4th Int.Conf. on Distributed Computing System, pp.526-532, May 1984.

[Kim, 88] K.H. Kim,and J.C. Yoon, "Approaches to Implementation of a Repairable Distributed Recovery Block Scheme", 18th International Symposium of Fault-Tolerant Computing, Pittsburgh, June 1988.

[Kim, 88a] K.H.Kim , "Programmer-Transparent Coordination of Recovering Concurrent Processes : Philosophy and Rules for Efficient Implementation", IEEE Trans. Software Eng, vol. 14, no. 6, June 1988.

[Kim, 89] K.H. Kim, "An Approach to Experimental Evaluation of Real-Time Fault-Tolerant Distributed Computing Schemes", IEEE Trans. on Soft. Eng., vol. SE-15, no. 6, pp. 715-725, June 1989.

[Kim, 91] K.H. Kim, W.J. Guan, A. Damm and J.A. Rohr, "Approaches to Design of Temporary Blackout Handling Capabilities and an Evaluation with a Real-Time Tightly Coupled Network Testbed", Proc. IEEE Computer Society's Symp. on Fault Tolerant Computing , Montreal, June 1991.

[Kim, 91a] K.H. Kim and B.J. Min, "Approaches to Implementation of Multiple DRB Stations in Tightly Coupled Computer Networks", Proc. IEEE Computer Society's Computer Software and Applications Conf., Tokio, September 1991.

[Knight, 85] J.C.Knight and S.Gregory, "A New Linguistic Approach to Backward Error Recovery", Digest of Papers of 15th IEEE FCTS, Ann Arbor, Michigan, pp. 404-409, June 1985.

[Kramer, 85] J. Kramer and J. Magee, "Dynamic Configuration of the Distributed Computer Systems", IEEE Trans. on Soft. Eng., vol. SE-11, no. 4, pp. 425-436, April 1985.

[Laprie, 85] J.C. Laprie, "Dependable Computing and Fault Tolerance : Basic Concepts and Terminology", Proc. 15th IEEE Int. Symp. on Fault-Tolerant Computing, pp. 2-11, Ann Arbor, Michigan, June 1985.

[Laprie, 89] J.C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and Analysis of Hardware and Software Fault-Tolerant Architectures", LAAS Report N.89257, July 1989.

[Laprie, 90] J.C. Laprie, B. Randell, W.C. Carter and J.E. Dobson, "Dependability Concepts and Terminology", Esprit Bra Project 3092 - Predictable Dependable Computing Systems, First Year Report, Task A, vol. 1, May 1990.

[Lee, 78] P.A. Lee, B. Randell and P.C. Trealeaven, "Reliability Issues in Computing System Design", Computing Surveys, vol. 10, no. 2, June 1978.

[Lee, 78a] P.A. Lee, "A Reconsideration of the Recovery Block Scheme", Computer Journal, vol. 21, pp. 306-310, November 1978.

[Lesk, 82] M.E. Lesk and E. Schimidt, "LEX : A Lexical Analyser Generator", Bell Laboratories, August 1982.

[Magee, 87] J. Magee, J. Krammer and M. Sloman, "Constructing Distributed Systems in CONIC", Imperial College Research Report DOC 87/4, March 1987.

[Nacamura, 88] L. Nacamura Jr., "Projeto e Implementação de um Núcleo de Sistema Operacional Distribuído com Mecanismos para Tempo Real", Dissertação de Mestrado, CPGEEL, UFSC, Florianópolis, Julho 1988.

[Nacamura, 92] L. Nacamura Jr., "Proposição de um Modelo de Tolerância a Faltas Baseado em Alta Replicação", Monografia / Exame de Qualificação para Doutorado, CPGEEL, UFSC, Abril 1992.

[Powell, 90] D. Powell, "Fault Assumptions and Assumption Coverage - a Contribution to the Fundamental Concepts of Dependability", LAAS Report no. 90074, February 1990.

[Randell 75] B. Randell, "System Structure for Software Fault Tolerance", IEEE Trans. on Software Eng., Vol. SE-1, N.1, pp 220-232, June 1975.

[Russell, 80] D.L.Russell, "State Restoration in Systems of Communicating Processes", IEEE Trans. Software Eng., vol. SE-6, pp 183 - 194, March 1980.

[Siewiorek, 84] D. Siewiorek, "Architecture of Fault-Tolerant Computers", IEEE Computer, pp. 9 - 18, August 1984.

[Silva, 88] E.S. Silva, "Uma Linguagem de Programação de Componentes Elementares para Aplicações Distribuídas em Tempo Real : Projeto e Implementação", Dissertação de Mestrado, CPGEEL, UFSC, Florianópolis, Agosto 1988.

[Souza, 88] L.E. Souza, "Um Suporte para Configuração Estática de Sistemas Distribuídos Utilizando Abordagem por Linguagem : Projeto e Implementação", Dissertação de Mestrado, CPGEEL, UFSC, Florianópolis, Dezembro 1988.

APÊNDICE

ALGORITMO DA TAREFA CONTROLE

Será apresentado a seguir o algoritmo da tarefa CONTROLE, através de um pseudo-código. A pseudo-linguagem utilizada para este fim baseia-se na linguagem PASCAL.

A estrutura de tarefa é representada por um procedimento (PROCEDURE). O envio e a recepção de mensagens são expressos através de extensões, da seguinte maneira :

envio : **ENVIA** msg **PARA** destino;

recepção : **RECEBE** msg **DE** origem;

recepção seletiva : **RECEBE** msg **DE** origem

bloco

OU TIMEOUT

bloco;

O campo msg corresponderá a uma variável a ser processada como mensagem. os campos origem e destino indicarão o módulo ou tarefa envolvidos na comunicação com a tarefa CONTROLE.

Descrição das variáveis utilizadas

variáveis do tipo booleano :

inst_ativa : indica se a instância está em execução.

inst_par_ativa : idem, relativamente à instância par.

primaria : indica se a instância em questão é primária (TRUE) ou secundária (FALSE).

princ_exec : indica se o algoritmo executado é o principal (TRUE) ou o alternativo (FALSE).

variáveis do tipo inteiro :

num_ciclo : indica o número de ciclo de processamento atual.

num_ciclo_par : idem, relativamente à instância par.

num_ciclo_esp : indica o próximo número de ciclo esperado.

teste : indica o resultado do teste de aceitação (positivo : 1; negativo : 0).

teste_par : idem, relativamente à instância par.

env_res : indica se os resultados do processamento foram enviados ao módulo receptor (positivo : 1; negativo : 0).

```
PROCEDURE  CONTROLE;

VAR
   inst_ativa, inst_par_ativa, primaria, princ_exec : boolean;
   num_ciclo, num_ciclo_par, num_ciclo_esp, teste, teste_par, env_res : integer;


BEGIN
   inst_par_ativa := TRUE;
   num_ciclo_esp := 1;

   WHILE  TRUE  DO
      BEGIN
         princ_exec := TRUE;
         env_res := 0;
         RECEBE  num_ciclo  DA instância par;
         inst_ativa := TRUE;
         IF num_ciclo <> num_ciclo_esp THEN

             <sinaliza exceção E5>

         IF inst_par_ativa = TRUE THEN
            BEGIN
               ENVIA  num_ciclo  PARA  instância par;
               RECEBE  num_ciclo_par  DA  instância par;

                 <compara os números de ciclo>


               OU TIMEOUT

                  <sinaliza exceção  E1>

            END;
         WHILE  inst_ativa = TRUE  DO
            BEGIN
               RECEBE  teste  DA  tarefa TEST_I/O;
               IF  teste = 0  THEN
                  BEGIN
                     IF  princ_exec = TRUE  THEN

                         <reconfiguração interna>

                     ELSE
                        BEGIN

                           <sinaliza exceção E4>

                           inst_ativa := FALSE;

                        END;
               IF int_par_ativa = TRUE  THEN
                  BEGIN
                     ENVIA  teste  PARA  instância par;
                     RECEBE  teste_par  DA instância par;
                        BEGIN
                           IF  primaria = TRUE  THEN
                              BEGIN
                                 IF (teste = 0) AND (teste_par = 1) THEN
```

```
                                <instância troca de função>

                            END
                         ELSE
                            BEGIN
                               IF (teste = 1) AND (teste_par = 0) THEN

                                  <instância troca de função>

                               END;
                         END

                      OU TIMEOUT

                          <sinaliza exceção E2>

                   END;

                IF (primaria = TRUE) AND (teste = 1) THEN
                   BEGIN
                      RECEBE env_res DA tarefa TEST_I/O;
                      IF inst_par_ativa = TRUE THEN
                         ENVIA env_res PARA instância par;
                   END;

                IF primaria = FALSE THEN
                   BEGIN
                      IF inst_par_ativa = TRUE THEN
                         BEGIN
                            IF teste_par = 1 THEN
                               BEGIN
                                  RECEBE env_res DA instância par
                                  OU TIMEOUT

                                       <sinaliza exceção E3>

                               END;
                         END;
                   END;

                IF teste = 1 THEN inst_ativa := FALSE;

             END;
          num_ciclo_esp := num_ciclo + 1;

            <sinaliza fim de ciclo>
            <recebe sinalização do gerenciador de faltas>

       END;

END;
```