Anais

20^o Simpósio Brasileiro de Redes de Computadores

III Workshop de Testes e Tolerância a Falhas

Búzios, RJ
21 de Maio de 2002

Editor
Avelino Francisco Zorzo

Organização
Núcleo de Computação Eletrônica
Universidade Federal do Rio de Janeiro

Promoção
Sociedade Brasileira de Computação - SBC
Laboratório Nacional de Redes de Computadores - LARC

Comitê de organização do SBRC

Coordenação Geral:

Luci Pirmez, NCE/UFRJ
Luiz Fernando Rust da Costa Carmo, NCE/UFRJ

Coordenação do Comitê de Programa:

Raimundo José de Araújo Macêdo, UFBA

Coordenação de Tutoriais:

Paulo Henrique de Aguiar Rodrigues, NCE/UFRJ

Coordenação de Minicursos:

José Ferreira Rezende, COPPE/UFRJ

Coordenação de Workshops:

Vitório Bruno Mazzola, UFSC

Coordenação do Workshop de Testes e Tolerância a Falhas:

Avelino F.Zorzo, PUCRS

Coordenação do Workshop em Segurança de Sistemas Computacionais:

Carlos Maziero, PUCPR

Coordenação do Workshop de Tempo Real:

Coordenador: Marcelo Ricardo Stemmer, UFSC
Vice-coordenador: Maria Luiza Sanchez, UFF

Coordenação do Workshop de TMN:

Coordenadora: Elizabeth Specialski, UFSC
Vice-Coordenador: Lisandro Z. Granville (UFRGS)

Coordenação de Palestras e Painéis:

Rogério Drumond, UNICAMP

Coordenação de Discussões Políticas:

Luiz Fernando Gomes Soares, PUC-RJ

Prefácio

O I e II Workshop de Testes e Tolerância a Falhas ocorreram respectivamente em 1998 e 2000, nas cidades de Porto Alegre e Curitiba, de forma a congregar profissionais da área nos anos que não acontecem o SCTF. A partir de discussões no último WTF em Curitiba, decidiu-se reunir a comunidade de testes e tolerância a falhas em conjunto com outro evento de porte nacional que tivesse alguma relação com o WTF. Para 2002 foi decidido realizar o evento em conjunto com o Simpósio Brasileiro de Redes de Computadores (Búzios-RJ), uma vez que o mesmo congrega diversos profissionais que trabalham em áreas correlatas.

Continuando o formato estabelecido no último WTF, e em concordância com os workshops realizados durante o SBRC, o formato do WTF contemplará atividades de apresentação de trabalhos técnicos e um painel de discussão sobre tolerância a falhas no Brasil. O principal objetivo deste evento é promover discussões e troca de idéias sobre trabalhos e projetos teóricos e práticos em andamento no país.

Nesta edição teremos um conjunto de 12 artigos selecionados pelo comitê de programa como artigos completos e 3 artigos selecionados como resumos. Espero que em conjunto com o painel, estes artigos e resumos gerem discussões sobre tolerância a falhas para que possamos fazer no ano de 2003 mais um excelente evento.

Aproveito este espaço para agradecer aos organizadores do SBRC2002 pela excelente infra-estrutura que vem sendo oferecida aos organizadores dos workshops e pelo excelente trabalho sendo realizado na organização do SBRC2002. Agradeço também aos membros do comitê de programa do WTF pela contribuição na seleção dos artigos que compõem o volume final do WTF e pelas discussões realizadas no momento da escolha destes artigos (sem eles o WTF não seria possível).

Avelino Francisco Zorzo
FACIN/PUCRS
Coordenador do III WTF

Comitê de Programa

- Avelino F. Zorzo (PUCRS) (coordenador)
- Marinho P. Barcellos (UNISINOS)
- Carlos A. Maziero (PUCPR)
- Cecilia M. F. Rubira (UNICAMP)
- Eliane Martins (UNICAMP)
- Elias P. Duarte Jr. (UFPR)
- Francisco Brasileiro (UFPB)
- Ingrid Jansch-Pôrto (UFRGS)
- Joni S. Fraga (UFSC)
- Rogério De Lemos (University of Kent, UK)
- Silvia R. Virgilio (UFPR)
- Taisy R. Weber (UFRGS)

Conteúdo

Sessão 4 - Recuperação de falhas e Modelagem

Sessão 5 - Testes e Simulação

Inserção Automática de Técnicas de Tolerância a Falhas em Descrições VHDL[1]

Ana Carla dos Oliveira Santos, Sérgio Vanderlei Cavalcante

{acos, svc}@cin.ufpe.br
Universidade Federal de Pernambuco - UFPE
Centro de Informática
Cx. Postal 7851 CEP 50732-970
Recife – PE – Brasil

Resumo

À medida que os usuários dependem mais do desempenho de máquinas, uma falha no funcionamento destas causa um prejuízo mais visível e mais grave. O uso de técnicas de tolerância a falhas pode aumentar a confiabilidade dos sistemas amenizando, assim, esse problema. Esse documento apresenta o desenvolvimento de uma ferramenta para inserir técnicas de tolerância a falhas em sistemas de hardware descritos em VHDL.

1 Introdução

Sistemas de computação vêm sendo mais empregados a cada dia, atingindo um maior número de usuários, que passam a depender mais fortemente do desempenho desses sistemas. À medida que mais pessoas são beneficiadas pelas máquinas, maior pode ser o prejuízo causado por problemas ocorridos no funcionamento destas.

Dessa forma, torna-se necessária a utilização de mecanismos para lidar com os problemas que potencialmente possam afetar o bom funcionamento dos sistemas. Tolerância a falhas é um desses mecanismos. Tolerar as falhas do sistema, implica em reconhecer que estas são inevitáveis e oferecer alternativas que permitam ao sistema manter o funcionamento desejado mesmo na ocorrência destas.

O desenvolvimento de sistemas embutidos vem sendo realizado de forma cada vez mais sistemática seguindo metodologias mais elaboradas de projeto. O desenvolvimento de técnicas de síntese de alto-nível e, agora mais recentemente, o emprego de metodologias de hardware/software co-design[2] vêm tornando o projeto de sistemas embarcados mais maduro e conseqüentemente diminuindo o tempo para comercialização dos produtos e aumentando a qualidade destes.

Uma forma de amadurecimento de metodologias é a utilização de ferramentas que auxiliem o projetista durante o desenvolvimento do produto. Uma vantagem do uso de ferramentas é que, geralmente, estas são bem testadas e robustas, e desse modo evitam a inserção de erros de projeto nas etapas em que atuam, que se tornam automáticas e corretas por construção.

Ferramentas automáticas envolvendo o projeto de hardware tolerante a falhas vêm sendo estudadas já há algum tempo. A ferramenta MEFISTO[4] por exemplo, foi elaborada para injetar automaticamente falhas em modelos VHDL, com o intuito de validar o mecanismo de tolerância a falhas utilizado. A injeção de falhas pode ser feita pela ferramenta por duas abordagens distintas: modificação do modelo VHDL, através da inserção de componentes sabotadores ou mutação dos compoonentes existentes; ou utilização de comandos nativos dos simuladores, para inserir erros nos sinais de saídas dos módulos. Outro

1 VHDL – Very High Speed Integrated Circuit Hardware Description Language

trabalho relacionado é a inserção automática de BIST (Built-In Self Test) em descrições VHDL, para aumento da testabilidade dos componentes, como apresentado em [5] .

Apesar do avanço em metodologias de automação de projeto nessa área, nota-se a ausência de ferramentas de auxílio à etapa de aplicação das técnicas de tolerância falhas, que ainda é feita manualmente. A análise da confiabilidade requerida e a escolha das técnicas a serem empregadas são ainda realizadas de forma intuitiva dependendo quase que exclusivamente da experiência da equipe de desenvolvimento do sistema. Como todo processo de desenvolvimento de sistemas computacionais, a tendência é que essas etapas de desenvolvimento sejam gradativamente automatizadas ou passem a receber auxílio de ferramentas para a escolha das técnicas e exploração das alternativas, que, somadas à capacidade de entendimento do problema e experiência prévia dos projetistas, tornem o projeto de sistemas embarcados críticos mais estável e confiável.

Além disso, após a escolha das técnicas, a implementação do sistema pode se tornar mais complexa devido à inserção de redundância que, eventualmente, pode exigir a utilização de protocolos de sincronização entre os módulos replicados para que as técnicas sejam corretamente aplicadas. Sistemas simples tendem a ser mais confiáveis que sistemas mais complexos; e partindo deste princípio, a aplicação de técnicas de tolerância a falhas a um projeto simples, pode torná-lo complexo o bastante de modo a não compensar a inserção de redundância já que a possibilidade de erros de projeto aumenta e a confiabilidade do sistema como um todo pode ser prejudicada. Uma forma sistemática e conhecidamente correta para a inserção das técnicas resolveria esse problema.

2 A ferramenta ToleranSE

Este documento vem apresentar o desenvolvimento da ferramenta ToleranSE – Tolerância a Falhas em Sistemas Embutidos – que visa a inserção automática de técnicas de tolerância a falhas em projetos de sistemas embutidos. Esse tipo de sistema, geralmente é constituído por módulos implementados em software e em hardware. Inicialmente, a ferramenta tratará da aplicação de técnicas na porção de hardware do projeto.

A especificação de hardware esperada como entrada para a ferramenta deve estar descrita na linguagem de descrição de hardware VHDL. Além da especificação do sistema em VHDL, a ferramenta também recebe como entrada a especificação das técnicas a serem implementadas no projeto. A técnica escolhida é especificada pelo usuário através da interface gráfica da ferramenta. A escolha da técnica fica sob a responsabilidade do projetista sendo apenas auxiliada pela ferramenta.

Como saída, a ferramenta gerará uma nova descrição VHDL do sistema, apresentando aspectos de tolerância a falhas, conferidos pela implementação das técnicas determinadas. A figura 1 mostra um esquema sobre o funcionamento geral da ferramenta apresentada.

Algumas técnicas de tolerância a falhas foram escolhidas para serem suportadas pela ferramenta. São elas: códigos de detecção e correção de erros de *Hamming*, *NMR*, *Mid-Value Select* e *Flux-Summing*[1] .

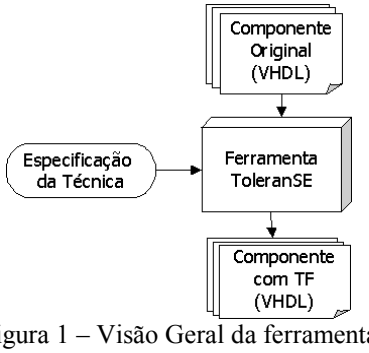


Figura 1 – Visão Geral da ferramenta ToleranSE

2.1 Técnicas de Tolerância a Falhas Abordadas

As técnicas a serem abordadas pela ferramenta foram escolhidas por serem muito utilizadas em projetos de hardware. Cada uma delas será brevemente descrita nesta seção.

NMR

A redundância modular ou NMR (*n-modular redundancy*) consiste na utilização de *N* módulos, com mesma funcionalidade, realizando a mesma computação e utilizando-se um mecanismo de voto por maioria para a escolha da saída correta.

Essa técnica considera que a probabilidade de mais de um módulo apresentar falhas durante a computação é muito pequena e desse modo a confiabilidade do sistema seria aumentada. Essa consideração exige que os módulos sejam independentes no que diz respeito às falhas[1] .

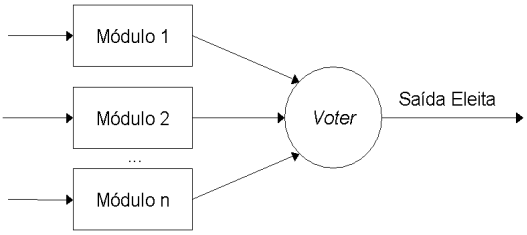


Figura 2 – Técnica NMR

Flux-Summing

Os sistemas de controle com realimentação, como por exemplo os controladores de temperatura, podem ser mais beneficiados pela técnica *flux-summing*, que utiliza propriedades inerentes desses sistemas (de controle de *loop* fechado) para a compensação de falhas.

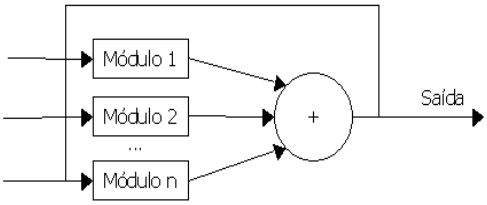


Figura 3 – Técnica Flux-Summing

A implementação desta técnica consiste em utilizar módulos redundantes e um transformador que recebe as saídas dos módulos como entrada, e sua saída é proporcional à soma das saídas dos módulos. Cada módulo é realimentado pela saída do transformador e desse modo a falha de um dos módulos é percebida e compensada pelos demais[1] .

Mid-Value Select

Em situações onde pequenas variações de valores da saída não são consideradas erro no sistema, uma variação conhecida como *mid-value select* pode ser utilizada. Nessa técnica, o sistema de votação majoritária é substituído por um componente que escolhe a saída que apresenta valor médio dentre as demais saídas do sistema.

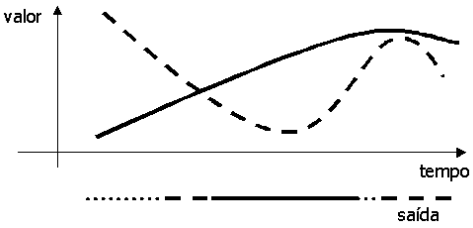


Figura 4 – Técnica Mid-Value Select

Um exemplo para a aplicação da técnica são os sistemas de conversão analógico-digital, onde as saídas podem sempre diferir nos bits menos significativos. A utilização da técnica TMR clássica neste caso não seria adequada, já que com muita freqüência acusaria erros inexistentes no sistema. Essa técnica só pode ser utilizada quando houver um número ímpar de módulos, tal que sempre uma saída seja a que se encontra no meio entre os valores considerados[1] .

Código de Hamming

Os códigos de detecção e correção de erros consistem na adição de informação redundante a conjunto de bits com o intuito de detectar e corrigir erros nos vetores de dados.

A informação redundante é produto de alguma operação realizada sobre os dados já existentes. Realizando novamente a operação e comparando o resultado com a informação redundante, pode-se detectar os erros.

No código de *Hamming*, bits redundantes são inseridos nas posições 2^n do dado codificado e as demais posições são preenchidas pelos bits originais. Cada bit de dados é utilizado para o cálculo dos bits de verificação cuja posição está contida em sua decomposição em potências de 2. Então um erro único em qualquer das posições de dados, alterará o valor dos bits redundantes que utilizaram o bit corrompido em seu cálculo. Dessa forma, para identificar a posição do bit corrompido, basta somar as posições dos bits de verificação que estão alterados. Identificada a posição do erro, o dado original pode ser facilmente recuperado.

2.2 Método de Aplicação Automática das Técnicas

A aplicação das técnicas propostas é realizada pela ferramenta através de diversas etapas. Primeiro, o código VHDL original desenvolvido pelo projetista da aplicação, que é o usuário da ferramenta, é tratado pelo analisador sintático que extrai informações necessárias para a geração do novo componente integrando os aspectos de tolerância a falhas. Essas informações, como a interface do componente, portas de I/O, e os sub-componentes instanciados, são armazenados na ferramenta. Foi utilizado, para a geração dos componentes específicos de tolerância a falhas – tais como os *voters* das técnicas NMR, Flux-Summing e Mid-Value Select, e os codificadores e decodificadores de *Hamming* – um gerador de componentes, que dependendo das características de cada aplicação, implementa o código VHDL dos componentes responsáveis pela implementação de cada uma das técnicas, específicos para a aplicação. Em seguida, a ferramenta gera as instâncias das réplicas do componente original, desenvolvido pelo projetista, além de instanciar os componentes específicos para tolerância a falhas.

É gerado ainda pela ferramenta todo o código VHDL adicional para a implementação da técnica escolhida, como as atribuições de sinais representando as ligações entre os módulos replicados e o componente de tolerância a falhas, além das conexões das instâncias com a interface do componente final, gerado pela ferramenta.

A figura 5 representa uma visão geral do funcionamento da ferramenta ToleranSE.

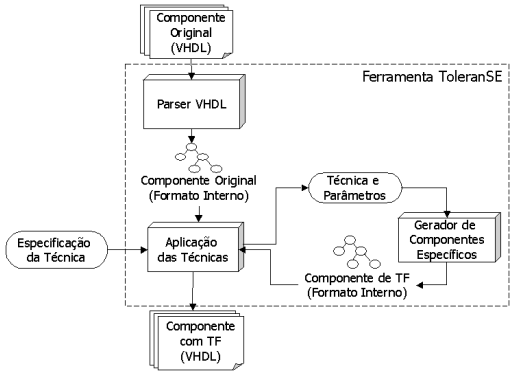


Figura 5 – Arquitetura da ferramenta ToleranSE

O método para geração da nova descrição VHDL do componente, incluindo tolerância a falhas, possui particularidades para cada uma das técnicas abordadas pela ferramenta. As técnicas que envolvem replicação de módulos – NMR, e suas variações, Mid-Value Select e Flux Summing – são aplicadas de forma similar, salvo pequenas particularidades apresentadas mais adiante.

É utilizada a arquitetura estrutural de VHDL para a geração do novo componente. O ideal é que o componente gerado possua a interface o mais parecida possível com a do componente original, minimizando o impacto da aplicação da técnica sobre o resto da implementação do sistema. Dessa forma, a interface do novo componente é inicialmente feita

igual à interface do componente original. A replicação dos módulos é feita instanciando-se o componente original tantas vezes quantas forem o número de réplicas utilizadas na técnica. Tal número será denominado *N*. Enquanto as entradas do componente original são distribuídas para as entradas dos módulos replicados, as saídas de cada réplica são agrupadas em um único vetor de bits, que será utilizado como entrada para o componente votante. Assim, são formados *N* vetores que agrupam as saídas de cada uma das réplicas. Uma instância do *voter* de *N* entradas é inserida no novo componente e os *N* vetores são fornecidos como entradas. A saída do *voter*, saída eleita dentre as saídas das réplicas, será também um vetor que representa um agrupamento de outros sinais e vetores. Esse vetor é então desagrupado nas saídas do componente gerado. Em alguns casos, o *voter* pode produzir saídas adicionais, indicando a ocorrência de falha (discordância entre os módulos) ou apontando qual dos módulos foi o discordante. Assim, essas saídas adicionais são acrescidas ao componente gerado, tendo seus valores simplesmente repassados para a interface.

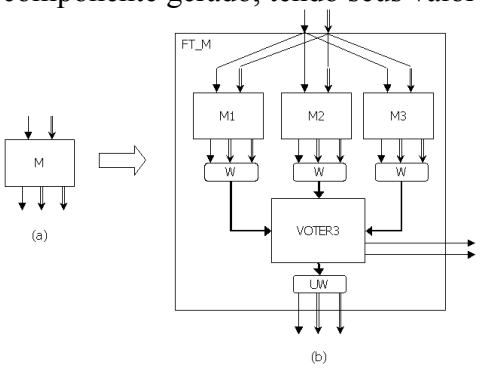


Figura 6 – Exemplo de Aplicação da Técnica com tripla replicação: (a) indica o componente original e (b) representa a arquitetura do componente gerado

A figura 6 representa o método geral de aplicação das técnicas que utilizam réplicas de componentes, considerando *N* igual à 3. Em (a) é representado o componente original, e em (b) o componente gerado pela ferramenta. Na figura, os componentes W representam os *wrappers,* responsáveis por agrupar os vetores, enquanto os componentes UW são os *unwrappers*, responsáveis pelo desagrupamento desses vetores. Tais componentes não são componentes físicos, mas apenas representam as atribuições de sinais, geradas por software e inseridas no código do novo componente, responsáveis pelo agrupamento e desagrupamento dos sinais.

Para aplicação das técnicas Mid-Value Select e Flux-Summing, nem todas as saídas dos módulos replicados devem ser levadas em consideração.

A saída do voter, nestes casos, depende operações de comparação ou soma sobre as entradas das réplicas. Assim, não faz sentido agrupar, no vetor utilizado como entrada para o voter, os sinais de saída do componente original que não representem valores numéricos. Sinais que não contenham valores numéricos, como flags por exemplo, não devem ser levados em consideração, para a correta aplicação das técnicas.

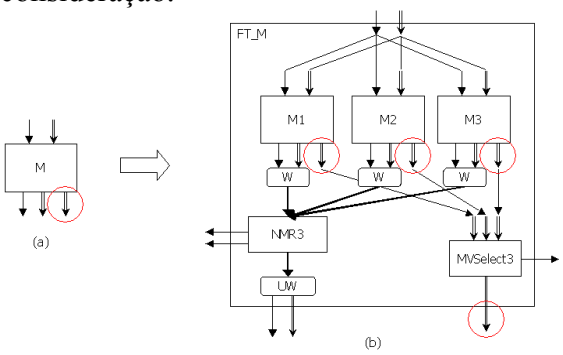


Figura 7 – Exemplo de Aplicação da Técnica Mid-Value com N = 3: (a)indica o componente original e (b) representa a arquitetura do componente gerado

Nestes casos, o projetista deve indicar a qual sinal de saída do componente original deve ser aplicada a técnica. Os demais sinais são então agrupados e votados por um voter NMR, onde os valores dos sinais não é importante. A figura 7 exemplifica o método de implementação da Mid-Value Select, com *N* igual à 3, destacando-se o vetor de saída do módulo original escolhido para aplicação da técnica.

A utilização de replicação de componentes para obtenção de tolerância a falhas, assume que os módulos replicados são independentes do que diz respeito à falhas. Para um melhor aproveitamento dessas técnicas, seria desejável que as réplicas fossem as mais independentes possíveis, sendo idealmente implementadas em diferentes abordagens, por diversas equipes de desenvolvimento e sintetizadas em chips separados. A implementação da técnica, encapsulando tanto as réplicas quanto o voter num único componente VHDL, a ser sintetizado num único chip, limita portanto o tipo de falhas que a configuração final deverá tolerar. Esse método, contudo, é eficaz para mascarar falhas parciais no chip ou componente programável onde o componente é sintetizado, além de servir como método de exploração e avaliação de alternativas para aplicação de tolerância a falhas, e para a prototipação rápida de projetos que, em ambiente de produção, serão desenvolvidos em diversos dispositivos.

Para a aplicação dos codificadores e decodificadores de *Hamming*, o procedimento utilizado é mais simples, sendo apenas necessário inserir uma instância do componente decodificador ou codificador de Hamming antes da entrada ou após a saída do componente original, dependendo das escolhas do projetista. As figuras 8 e 9 ilustram a aplicação dessa técnica a componentes genéricos. O componente *coder* recebe um vetor de dados e gera o vetor incluindo os bits redundantes para geração do código de *Hamming*, enquanto que o *decoder* faz a operação inversa. Desse modo, as interfaces dos componentes gerados diferem das interfaces dos componentes originais apenas pelo número de bits do vetor escolhido para aplicação da codificação/decodificação, que aumenta por conta da utilização dos bits redundantes.

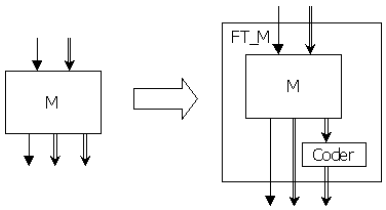


Figura 8 – Exemplo de Aplicação de um codificador de *Hamming*

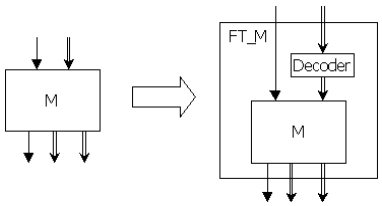


Figura 9 – Exemplo de Aplicação de um decodificador de *Hamming*

2.3 Gerador de Componentes Específicos

Um dos aspectos da inserção de tolerância a falhas é que cada solução é fortemente dependente da aplicação. Para a automatização da aplicação das técnicas, essa característica representa um problema à parte. A generalização das técnicas se torna necessária quando se pretende aplicá-las de forma automática, de modo que, para qualquer entrada fornecida pelo usuário, a ferramenta utilize um procedimento bem definido.

Assim, a implementação de uma biblioteca estática de componentes não é suficiente, já que não é possível prever todas as opções de componentes requeridas pelo usuário em ocasião da utilização da ferramenta. Sendo assim, um módulo da ferramenta desenvolvida é responsável por gerar dinamicamente, e sob demanda, os componentes VHDL específicos necessários para a aplicação da técnica escolhida pelo usuário.

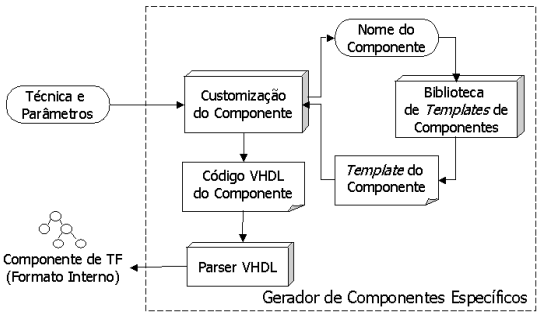


Figura 10 – Arquitetura Geral do Gerador de Componentes Específicos

O módulo gerador de componentes utiliza uma biblioteca de *templates* parametrizáveis de componentes VHDL, e instancia esses componentes de acordo com os argumentos passados por outros módulos da ferramenta. As saídas fornecidas pelo gerador representam os componentes específicos para a aplicação de tolerância a falhas – *voters* e codificadores, bem como os sub-componentes necessários para implementação destes.

2.4 Protocolo de Sincronização

Havendo replicação de módulos, é preciso garantir que os módulos replicados mantenham-se no mesmo passo da execução, evitando que a configuração tolerante a falhas – composta pelas réplicas e *voter* – detecte falhas inexistentes nos módulos. Assim, pode ser necessária a utilização de um protocolo de sincronização entre os módulos replicados e o *voter,* de forma que este não compute a votação entre as saídas nos intervalos em que uma das réplicas estiver ainda computando resultados ou uma das saídas encontre-se instável.

Nota-se que, sendo os *voters* das técnicas abordadas componentes puramente combinacionais, nos casos em que o componente original for implementado também de forma combinacional, não haverá necessidade de sincronismo, já que as réplicas receberão os mesmos sinais de entrada e portanto devem produzir as mesmas saídas sempre. Nesse caso podemos considerar que o efeito de não haver uma saída estável durante o período de computação do resultado final com o *voter* é semelhante ao efeito que se obtinha com o circuito original, apenas observando que o período que se deve esperar para se obter um resultado estável é igual ao maior período entre as N-versões acrescido do tempo utilizado pelo *voter*.

No caso de o componente original ter comportamento síncrono, regido por *clock*, há a necessidade de controle de sincronização e este é realizado através de um protocolo bem simples. O projetista deve implementar o componente original de forma que este indique através de um bit, denominado *ready*, que as saídas estão prontas para serem votadas. As saídas devem ser mantidas estáveis até que um outro bit, dessa vez de entrada, denominado *continue*, seja ativado. O bit de *continue* é enviado pelo *voter*, indicando que a votação foi realizada e que o módulo pode continuar executando sua máquina de estados. O *voter* por sua vez, espera que todas as réplicas ativem o bit de *ready*, indicando que suas saídas foram computadas e estão estáveis, para fazer a votação e liberar a saída da configuração tolerante a falhas. Após a votação ser realizada, o bit de *continue* deve ser enviado a todas as réplicas, e o componente volta a esperar pelos sinais de *ready*.

Para a implementação do protocolo de sincronização, alguns componentes VHDL adicionais são acrescentados aos *voters* para o controle de sincronização.

O principal destes componentes, o ControlSync, é regido por *clock*, que será conectado ao mesmo sinal de *clock* das réplicas, e é responsável pela execução de uma máquina de estados que implementa o protocolo. Além disso são utilizados um *timer*, para geração de um sinal de *timeout* quando uma das réplicas atrasar ou falhar no envio do sinal *ready,* e um registrador para manter a saída do *voter* estável até que ocorra a próxima votação. A figura 12 mostra um exemplo de *voter* 3MR encapsulado com o mecanismo de controle de sincronização em um único componente.

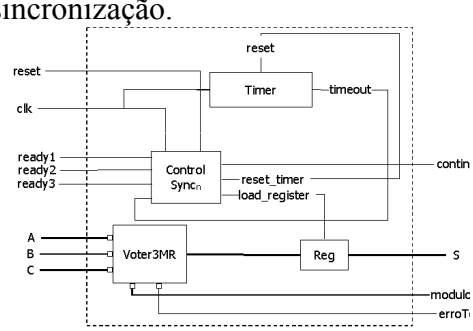


Figura 12 – *Voter* 3MR com controle de sincronização.

3 Resultados e Trabalhos Futuros

A ferramenta encontra-se atualmente em fase de testes. Um estudo de caso foi realizado aplicando-se a técnica 3MR no componente adicionador de um controlador de uma máquina de vender refrigerantes. O sistema foi validado através da injeção manual de falhas em vetores internos do projeto e a ferramenta de síntese de alto nível Max+plus II da Altera foi utilizada para simulação, que apresentou resultados satisfatórios. A tabela abaixo mostra o número de células lógicas utilizadas para a síntese e o atraso no caminho mais longo do projeto, antes e depois da aplicação automática da técnica. Pelos dados apresentados, nota-se que o número de células lógicas após aplicação de tolerância é pouco maior que três vezes o número original. Sendo a triplicação de recursos já esperada na implementação do 3MR, verifica-se que o *overhead* causado pela inserção da técnica é de 9 células lógicas. O aumento no atraso devido à utilização do *voter* foi de 5ns. Vale ressaltar que o componente utilizado para o estudo é bastante simples. Para componentes de mais alta complexidade, o *overhead*, de área e de tempo, causado pela utilização da técnica é ainda menos significativo, já que a complexidade do *voter* não é proporcional à complexidade do componente sendo replicado. Outros estudos de caso abrangendo todas as técnicas abordadas pela ferramenta deverão ainda ser realizados.

	Componente Original	Componente Tolerante a Falhas
Células lógicas utilizadas	21	72
Atraso no caminho mais longo	11,5 ns	16,5 ns

4 Conclusões

Com a automação da implementação das técnicas, o custo e o tempo de projeto dos sistemas de hardware tolerantes a falhas seriam diminuídos, além de a qualidade do sistema ser aumentada pela prevenção de inserção de erros de projeto que processos automáticos oferecem. A ferramenta também oferece auxílio ao projetista durante a fase de escolha das técnicas a serem aplicadas ao projeto, através da viabilidade de exploração no espaço de soluções. Num processo manual, mais lento e de custo mais elevado, a análise detalhada de qual técnica de redundância utilizar pode ser inviável. Assim, o projetista define a priori a técnica a ser utilizada podendo esta não ser a técnica ótima para o projeto. A ferramenta, oferecendo uma maneira de rápida implementação de várias técnicas, viabiliza a comparação do impacto que a aplicação de cada uma das técnicas pode causar na confiabilidade do sistema, permitindo uma busca mais eficiente da melhor técnica no espaço de soluções.

5 Referências

[1] Pradhan, D. K.; FAULT-TOLERANT COMPUTER SYSTEM DESIGN. Prentice Hall 1996.

[2] Vahid, F.; Givargis, T. EMBEDDED SYSTEM DESIGN: A UNIFIED HARDWARE/SOFTWARE APPROACH. Department of Computer Science and Engineering – University of California, 1999.

[3] Avizienis, A. TOWARD SYSTEMATIC DESIGN OF FAULT TOLERANT SYSTEMS. IEEE Computer, 30, No. 4 (1997), 51-58.

[4] Jenn, E.; Arlat J.; Rimen, M.; Ohlsson, J.; Karlsson, J. FAULT INJECTION INTO VHDL MODELS: THE MEFISTO TOOL. FTCS-24, pp. 66 – 75. IEEE, 1994.

[5] Kim, K.; Tront, J.G.; Ha, D.S. AUTOMATIC INSERTION OF BIST HARDWARE USING VHDL. IEEE Design Automation Conference, pp. 9 – 15. IEEE, 1988.

Controle de Célula de Produção de Tempo Real com DMIs*

Leandro Azevedo Cassol, Avelino Francisco Zorzo
{cassol, zorzo}@inf.pucrs.br

FACIN - PUCRS - Av. Ipiranga, 6681 - 90619-900 - Porto Alegre - RS

Abstract. This paper presents the design of a controlling system for a production cell. The design uses an abstraction called Dependable Multiparty Interaction (DMI), which is used to enclose all interactions between the devices of the production cell. This paper also presents a short description of the production cell case study. The goal of the paper is to show that real time and fault tolerance requirements can be satisfied in a controlling software implemented with DMIs.

Keywords: Distributed Systems, Dependable Multiparty Interaction, Real Time, Fault Tolerance

1 Introdução

O aumento do uso dos computadores em quase todos os aspectos da vida moderna tem conduzido a uma necessidade de elevar a confiança dos sistemas de computadores e dos próprios computadores. Existem muitas áreas onde os computadores desempenham tarefas críticas. Um exemplo é a área de tempo real. Nesta área, uma falha nos computadores pode ocasionar resultados catastróficos, pois os resultados devem estar corretos não somente do ponto de vista lógico, mas também devem ser gerados no momento correto.

Alguns sistemas críticos e de tempo real envolvem atividades concorrentes complexas. Em alguns casos, estas atividades concorrentes podem trabalhar juntas, cooperando, para resolver um determinado problema. Em outros casos, as atividades podem ser completamente independentes ou podem ser essencialmente independentes apesar de necessitar concorrer para compartilhar recursos comuns do sistema. Na prática, diferentes espécies de concorrência podem coexistir em uma aplicação complexa, que irá necessitar de um mecanismo de suporte geral para controlar e coordenar as atividades concorrentes complexas.

Neste trabalho, mostra-se como projetar e implementar aplicações críticas de tempo real usando uma abstração de controle geral, chamada Interações Multi-Participantes Confiáveis [1]. Como estudo de caso, foi usado um modelo de uma célula de produção que descreve um problema real da indústria.

2 A Célula de Produção FZI

O modelo da Célula de Produção, usado nesta seção, foi desenvolvido pelo *Forschungszentrum Informatik* (FZI), Karlsruhe, Alemanha, como um estudo de caso que apresenta propriedades de tempo real [2]. Esta célula de produção é composta de duas esteiras transportadoras (esteira alimentadora e esteira depósito), um leitor de código de barras, quatro unidades de processamento (UP) e dois guindastes (ver Figura 1). A Célula de Produção FZI utiliza

* Pesquisa financiada pela HP-Brasil (Convênio CPAD/FACIN/HP)

um conjunto de atuadores e sensores. Os atuadores podem ser usados para alterar o estado do sistema, como ligar e desligar a esteira alimentadora, mover os guindastes, entre outros. Já os sensores fornecem informações sobre o estado do sistema, e.g. se há um bloco no final da esteira alimentadora ou quais são as posições dos guindastes.

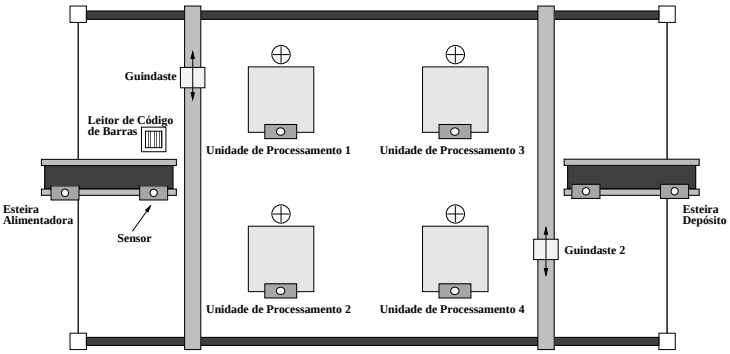


Figura 1. Estrutura da Célula de Produção FZI

Processamento dos Blocos. Os blocos entram no sistema através da esteira alimentadora e são transportados por ela até o sensor que está localizado no final dela detectar a presença de um bloco, ocasionando a parada da esteira. Nesse momento, o leitor de código de barras transmite as informações para o sistema de controle depois de ter lido de cada bloco.

Cada bloco possui um código de barras que contem informações para o seu processamento: *i)* quantas UP (no mínimo em uma e no máximo em quatro), quais e os tempos, mínimo e máximo, que são necessários para o processamento do bloco; *ii)* o tempo máximo que o bloco pode gastar em todo o sistema; e *iii)* se deve respeitar a ordem de processamento que foi informada no código de barras, pois um bloco pode ser processado sem respeitar essa ordem.

O controlador da célula deve garantir que os guindastes levem os blocos para a correta UP. Um bloco somente pode ser colocado em uma UP quando ela estiver desocupada. Existem dois tipos de UP: *i)* prensa: é ligada pelo controlador da célula e será desligada automaticamente quando ela terminar o processamento; e *ii)* forno: está sempre ligado. Um bloco é processado durante o tempo que permanecer nesse dispositivo. Depois de os blocos terem passado pelas UP, eles devem ser colocados na esteira depósito.

Propriedades. As propriedades do sistema estão organizadas em três classes: *safety properties*, *liveness properties* e *correctness properties*. Se todas as *safety properties* são satisfeitas, então nenhum dispositivo será danificado. Evitar colisões dos guindastes, evitar colisões entre os blocos e garantir que os blocos fiquem em áreas seguras são classificadas como *safety properties*. As *liveness properties* asseguram a ausência de *deadlocks* no sistema, isto é, todos os blocos são introduzidos no sistema pela esteira alimentadora e irão sair do sistema pela esteira depósito. Já as *correctness properties* asseguram que todas as informações lidas pelo controlador a partir do código de barras serão respeitadas [2].

3 Interações Multi-Participantes Confiáveis

Um mecanismo que abriga diversos processos executando um grupo de atividades em conjunto é chamado de interação multi-participantes (*multiparty interactions*) [3,4]. Em uma in-

teração multi-participantes, diversos processos (*threads*, objetos) de alguma forma "se reúnem" para produzir resultados combinados e intermediários, usam este estado para executar atividades em conjunto, e então abandonam a interação e continuam suas execuções normais. Mecanismos existentes para interações entre diversos participantes não fornecem recursos para tratar possíveis falhas que podem ocorrer durante a execução da interação. Em alguns, o sistema simplesmente pára como resposta a uma falha. Isto não é aceitável em diversas situações.

Em [1], o tratamento de exceções é adicionado ao mecanismo de interação multi-participantes. Este novo mecanismo é chamado de Interações Multi-Participantes Confiáveis (*Dependable Multiparty Interaction* - DMI). Uma DMI é uma interação entre diversos participantes que fornece recursos para: *i)* Tratar Exceções Concorrentes [5,6]; e *ii)* Garantir Consistência na Saída. Discussões a respeito das características de uma DMI podem ser encontradas em [1].

4 O Controlador da Célula de Produção

Para a construção de um controlador para o simulador da Célula de Produção FZI foi desenvolvido um projeto que satisfaz os requisitos de segurança *(safety)* e de tempo real (*correctness*) do estudo de caso (ver Seção 2), assegurando a ausência de *deadlocks* no sistema (*liveness*). O projeto foi separado em um conjunto de DMIs que controla as interações entre os dispositivos; um conjunto de controladores dos dispositivos que executam as DMIs; e um escalonador de DMIs que determina a ordem na qual as DMIs são executadas. Os requisitos de segurança são satisfeitos no nível de DMIs, enquanto os outros requisitos são atendidos pelos controladores dos dispositivos e pelo escalonador de DMIs (requisitos de tempo real).

Dessa forma, o *software* controlador para toda a Célula de Produção FZI consiste simplesmente em um conjunto de DMIs, em controladores dos dispositivos e em um escalonador de DMIs. Este controlador foi implementado na linguagem de programação Java.

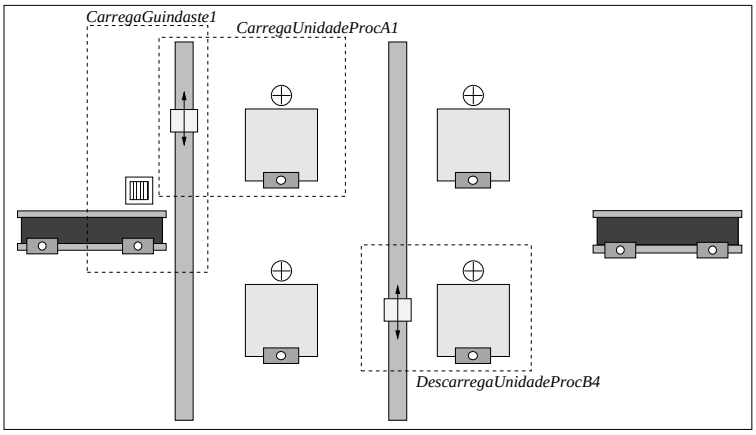


Figura 2. O conjunto de DMIs na Célula de Produção FZI

A Figura 2 mostra três das vinte e três DMIs que estão presentes na Célula de Produção. Cada DMI está representada por um retângulo pontilhado, a fim de salientar quais os dispositivos estão envolvidos na DMI. Devido aos possíveis processos de recuperação, duas DMIs, que possuem seus respectivos retângulos pontilhados sobrepostos, não podem ser executadas

concorrentemente, porque o mesmo dispositivo (ou bloco) não pode estar envolvido em mais do que uma DMI ao mesmo tempo [1].

Os movimentos de quase todos os dispositivos são desempenhados pelas DMIs e os dispositivos envolvidos na DMI são desligados antes do término dela. Assim, todos os dispositivos estão imóveis quando não estão sob o controle da DMI. O único dispositivo que não é controlado pelas DMIs é a esteira depósito, que é ativada pelo ambiente que engloba a célula.

4.1 Funcionamento

O conjunto de DMIs do controlador executa várias operações críticas, i.e. passagem de um bloco entre os dispositivos. A ativação destas DMIs e a ordem em que elas são executadas é a responsabilidade, respectivamente, dos controladores dos dispositivos e do escalonador de DMIs, que pode ser feito através de uma grande DMI. Os participantes neste outro nível de DMI, são ativados por *threads* de execuções externas, criadas imediatamente após o sistema iniciar sua execução.

Foram desenvolvidas vinte e três DMIs para controlar as interações entre os dispositivos: *CarregaCelula* (CC), *CarregaGuindaste1* (CG1), *RecuaGDT1* (RG1), *RecuaGDT2* (RG2), *CarregaUnidadeProcA*$_i$ (CUPA$_i$), *CarregaUnidadeProcB*$_i$ (CUPB$_i$), *DescarregaUnidProcA*$_i$ (DUPA$_i$), *Descarrega Unidade ProcB*$_i$ (DUPB$_i$), *ProcessaBloco* (PB), *CarregaEsteiraDeposito* (CED) e *DescarregaCelula* (DC). O índice i representa uma das quatro unidades de processamento, portanto existem quatro DMIs para cada DMI que possuir o índice i.

O ciclo de produção completo de um bloco na Célula de Produção FZI é o seguinte: um bloco entra no sistema da célula de produção através do ambiente. Este bloco é colocado no dispositivo esteira alimentadora pela DMI *CarregaCelula*. Após, a DMI *CarregaGuindaste1* é responsável por fazer o dispositivo leitor de código de barras ler as informações necessárias sobre o processamento do bloco. Essas informações são transmitidas, por esta DMI, para o dispositivo guindaste 1. A DMI *CarregaGuindaste1* também é responsável por fazer o dispositivo guindaste 1 agarrar o bloco no final da esteira alimentadora.

Quando o bloco estiver no dispositivo guindaste 1, a DMI *CarregaUnidadeProcA*$_i$ levará o bloco até a UP informada no código de barras do bloco. Após a chegada do bloco nesta UP, ele deverá ser processado. A DMI *ProcessaBloco* será responsável por processar o bloco. Terminando esta etapa, o *software* controlador poderá: *i)* fazer o dispositivo guindaste 1 pegar o bloco através da DMI *DescarregaUnidProcA*$_i$. Através desse caminho, o bloco deverá, mais tarde, ser colocado em outra UP, ocasionando um processo recursivo; ou *ii)* fazer o guindaste 2 pegar o bloco. Esta operação será realizada pela DMI *DescarregaUnidadeProcB*$_i$.

Quando o bloco estiver no dispositivo guindaste 2, ele poderá: *i)* voltar para uma UP através da DMI *CarregaUnidadeProcB*$_i$, para continuar o seu processamento; ou *ii)* ser colocado no dispositivo esteira depósito pela DMI *CarregaEsteiraDeposito.*

Após o bloco ser colocado na esteira depósito, a DMI *DescarregaBloco* será responsável por fazer o bloco sair da Célula de Produção. Neste ciclo de produção não foram citadas as DMIs *RecuaGDT1* e *RecuaGDT2*. Estas duas DMIs são utilizadas para os guindastes irem para áreas seguras. Estas são áreas onde somente um guindaste poderá chegar. Dessa forma, no momento em que um dos guindastes estiver executando a sua DMI de recuo, estará garantido que ele não poderá colidir com o outro guindaste.

A maioria das DMIs projetadas para o estudo de caso da Célula de Produção FZI, têm dois participantes: um que recebe o bloco como um argumento de entrada e o outro que retorna o bloco como um argumento de saída. O dispositivo que tem o bloco como um argumento de entrada, passa para o dispositivo que tem o bloco como um argumento de saída.

A DMI *CarregaGuindaste1* usa três objetos externos e é composta de três papéis[1]: *EsteiraAlimentadoraRole*, *LeitorCodigoBarrasRole* e *Guindaste1Role*. Na Figura 3, a passagem de um bloco de um papel para outro é representada por uma seta sólida com uma direção. Como pode ser visto nessa figura, o papel *EsteiraAlimentadoraRole* recebe o objeto Bloco como um parâmetro de entrada. Depois, é realizada a passagem física do Bloco da esteira alimentadora para o leitor de código de barras, que neste momento faz a leitura do código de barras e depois envia esse objeto para o papel *Guindaste1Role*, que representa o guindaste que está pegando fisicamente o objeto. A partir desse momento, o papel *Guindaste1Role* retorna esse objeto como um parâmetro de saída. Os acessos aos objetos externos bloco, esteira alimentadora, leitor de código de barras e guindaste 1 são representados por setas tracejadas. Já as atividades desempenhadas pelos papéis (*roles*) são representadas por quadrados pontilhados.

Durante a implementação do projeto da Célula de Produção FZI encontraram-se alguns problemas. Para resolver estes problemas foram tomadas as seguintes decisões: *i)* realizar o processamento dos blocos na Célula de Produção, sempre respeitando a ordem que foi informada no código de barras; e *ii)* para evitar a ocorrência de *deadlocks*, o número máximo de blocos que a célula de produção pode processar ao mesmo tempo foi estipulado em dois.

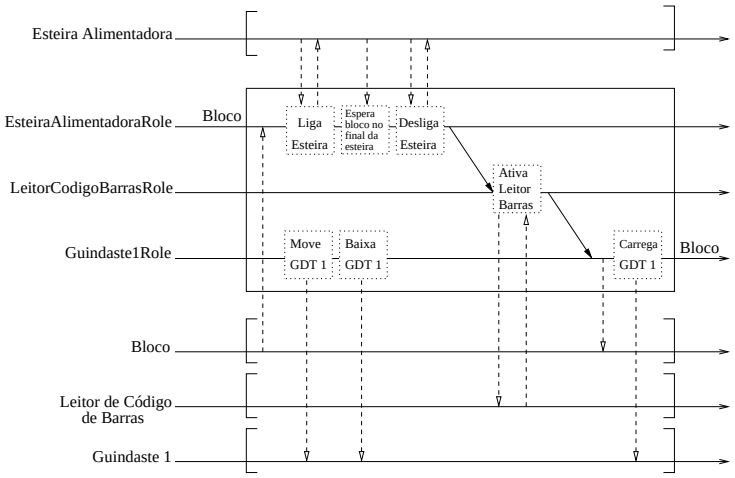


Figura 3. A DMI *CarregaGuindaste1*

4.2 Escalonador de DMIs

De forma a fazer com que as DMIs sejam executadas de maneira correta e que respeitem os requisitos de tempo real necessários para essa célula, foi necessária a implementação de um escalonador em vez de adicionar novas características de tempo real dentro de uma DMI.

O escalonador de DMIs é responsável por determinar a ordem em que as DMIs são executadas. Assim, os controladores dos dispositivos se comunicam com o escalonador de DMIs para agendar a ordem de execução das DMIs. Essa comunicação é realizada através da troca de mensagens entre eles, isto é, os controladores informam ao escalonador o estado dos dispositivos ou

[1] Cada papel é responsável por abrigar as instruções de cada participante de uma DMI.

algum pedido e ficam aguardando uma resposta do escalonador para executarem alguma DMI. Por sua vez, o escalonador recebe as mensagens dos controladores, verifica qual ação deve ser tomada e, após, envia mensagens para os controladores dos dispositivos que participarão da ação, a fim de que eles executem uma determinada DMI.

O estudo de caso apresenta dois tipos de requisitos de tempo. O primeiro refere-se ao tempo máximo que cada bloco pode gastar em toda a célula, já o segundo refere-se ao tempo mínimo e máximo necessários para o processamento de cada bloco nas UP. Para satisfazer o primeiro tipo de requisito, toda a vez que um guindaste estiver carregando um bloco, será verificado se o *deadline* desse bloco é menor que o *deadline* do outro bloco que está presente na célula de produção. Dessa forma, o bloco que possuir o menor *deadline* será processado primeiro. Já para satisfazer o segundo tipo de requisito, quando existirem dois blocos sendo processados ao mesmo tempo, os *deadlines* desses blocos são comparados para verificar qual bloco tem o menor *deadline*, ou seja, qual bloco deve ser retirado primeiro de uma UP.

4.3 Requisitos de Tempo

Os requisitos de tempo do estudo de caso foram satisfeitos através do escalonador de DMIs. Porém, para ter uma maior segurança quanto ao cumprimento desses requisitos, foram realizadas diversas medições de tempo de execuções das DMIs desenvolvidas para o estudo de caso. Para realizar essas medições foi utilizado um computador PC IBM Pentium Celerom, de 330 MHz, com 64 MB de memória RAM e tendo como sistema operacional o Linux Red Hat 7.0 (kernel 2.2.16-22). As medições dos tempos de execução das DMIs foram realizadas a partir de iterações compostas pelo processamento de quatro blocos. Os valores foram obtidos em duas etapas, sendo que em cada uma houve a repetição de cada iteração mil vezes, totalizando duas mil vezes o processamento de quatro blocos.

Como o processamento de cada bloco envolve a execução de algumas DMIs, após as duas etapas de medição dos tempos, obteve-se vários valores correspondente ao tempo de execução de cada DMI. Para cada DMI foram descartados os valores que estavam destoando em relação aos demais valores. Com base nesses novos valores, determinou-se que os maiores valores de cada DMI corresponderiam ao tempo máximo de execução de cada uma (ver Tabela 1).

DMIs	Tempo Máximo (seg.)
CG1	5
CUPA1 e CUPA2	5,8
CUPA3 e CUPA4	7,9
DUPA1 e DUPA2	3
DUPA3 e DUPA4	5,9
CUPB1 e CUPB2	5,1
CUPB3 e CUPB4	5,4
DUPB1 e DUPB2	7,7
DUPB3 e DUPB4	5,2
CED	7,6

Tabela 1. Tempo máximo de execução das DMIs

Unidades de Processamento	Tempo Mínimo (seg.)
1	26,1
2	26,1
3	25,7
4	25,7
(1,2) ou (2,1)	34,9 e 38,9
(1,3) ou (1,4)	34,5 e 36,7
(2,3) ou (2,4)	34,5 e 36,7
(3,1) ou (3,2)	39,9 e 38,5
(4,1) ou (4,2)	39,9 e 38,5
(3,4) ou (4,3)	39,5 e 36,3

Tabela 2. Tempo mínimo de processamento de um bloco

Com base nos tempos da Tabela 1, foi realizada a soma dos tempos de execução das DMIs que estão envolvidas no processamento de um bloco em uma e duas UP. Para cada UP foi obtido um valor. Todos os valores podem ser visualizados na Tabela 2.

Após a obtenção dos tempos da Tabela 1 determinou-se que os tempos mínimos necessários para realizar o processamento de cada bloco nas UP devem ser maiores que os valores apresentados na Tabela 2, pois se o *deadline* de um bloco for menor que esses tempos, a célula de produção poderá não cumprir o *deadline* em uma determinada UP. Um outro detalhe a ser acrescentado é que o tempo gasto para realizar o escalonamento foi desconsiderado por ser muito menor que o tempo gasto para executar uma DMI. Por exemplo, para executar o escalonamento leva-se microseg./miliseg. enquanto que o processamento mecânico de um bloco leva segundos ou até minutos.

Em algumas linhas da segunda coluna, da Tabela 2, têm-se dois valores. O primeiro valor foi obtido com o guindaste 1 fazendo a maior parte da operação na célula de produção, deixando só a retirada do bloco da célula de produção para o guindaste 2. Já o segundo valor que aparece nesta coluna foi obtido com o guindaste 2 fazendo a maior parte da operação, deixando somente a inclusão do bloco na célula de produção para o guindaste 1.

4.4 Tratamento de Exceções

O controlador da célula de produção irá levantar uma exceção toda a vez que o *deadline* de um bloco for menor que o tempo mínimo de processamento determinado na Tabela 2. Assim, quando um bloco irá ser processado, o escalonador verifica quanto tempo levará para processá-lo. Após, será levantada uma exceção se o tempo necessário para realizar o processamento do bloco for: *i)* menor que o tempo mínimo determinado no código de barras do bloco; *ii)* maior que o *deadline* do bloco na UP; e *iii)* maior que o *deadline* do bloco em todo o sistema.

Se uma exceção for levantada, o controlador irá interromper a execução do sistema. Nesse trabalho não foi realizado o tratamento das exceções levantadas durante a execução do controlador, pois o objetivo é mostrar que as DMIs podem ser usadas em sistemas de tempo real.

5 Conclusões

Em sistemas de tempo real existe uma dificuldade em compatibilizar dois objetivos fundamentais: garantir que os resultados sejam produzidos no momento desejado e dotar o sistema de flexibilidade para adaptar-se a um ambiente dinâmico e, assim, aumentar sua utilidade. Dessa forma, o uso de um escalonador deve assegurar que os *deadlines* das tarefas sejam cumpridos. Em alguns sistemas, mesmo em caso de falha de algum componente, as tarefas que nele estavam sendo executadas devem ser asseguradas. Para isso, o escalonador deve implementar tolerância a falhas. Um exemplo de um sistema que engloba estas duas características, tolerância a falhas e tempo real, é o estudo de caso da Célula de Produção FZI [2].

Para projetar e implementar o sistema de controle para a Célula de Produção FZI (estudo de caso) foi utilizado o mecanismo proposto em [1], DMI. Porém, o mecanismo de DMI não fornece, de maneira direta, as propriedades necessárias para poder ser usado em interações entre diversos participantes que possuem requisitos de tempo real.

Um método para atender os requisitos de tempo real pode ser a inclusão, nas próprias DMIs, dessas características. Este tipo de filosofia foi adotada em [7] e [8], onde foi estudado um método para incluir requisitos de tempo real em um mecanismo similar às DMIs, as *CA Actions* [9,10]. Uma outra forma de atender estes requisitos é a utilização de um escalonador de DMIs, ao invés de incluir novas propriedades nas DMIs. Dessa forma, é possível atender aos requisitos

de tempo real, para problemas similares aos da célula de produção, sem incluir estes requisitos nas DMIs. Parte desses resultados estão apresentados em [11] e [12].

As propriedades descritas na Seção 2 foram atendidas da seguinte maneira: *i)* as *safety properties* foram satisfeitas pelas DMIs; *ii)* as *liveness properties* foram atendidas pela maneira como foi projetado o sistema controlador e também pelo escalonador de DMIs; e *iii)* as *correctness properties* foram atendidas pelo escalonador de DMIs, por exemplo, o escalonador não deixa a DMI $DUPA_2$ retirar uma peça da UP 2 antes dela ter sido totalmente processada, ou ainda, a ordem com as quais as peças têm que passar pelas UP é respeitada pelo escalonador.

Neste trabalho, mostrou-se como usar as DMIs para projetar um sistema crítico: o da Célula de Produção FZI. Foi demonstrado que aplicando as DMIs para este estudo de caso ajudou-se a melhorar a construção do projeto para esse sistema, pois o uso de DMIs para implementar o sistema permite garantir todos os requisitos relacionados com as atividades concorrentes da Célula de Produção FZI [13]. Foi mostrado que a utilização de um mecanismo que pode confinar os erros e, conseqüentemente, fornecer tolerância a falhas também pode ser utilizado em sistemas de tempo real. Embora um projeto baseado em DMIs possa diminuir o desempenho de alguns sistemas, acredita-se que os benefícios ganhos por um projeto simples, usando componentes reutilizáveis e fornecendo ao sistema uma disciplina de tolerância a falhas, possibilitem construir aplicações críticas de tempo real com mais eficiência.

Referências

1. A. F. Zorzo. *Multiparty Interactions in Dependable Distributed Systems.* PhD thesis, University of Newcastle Upon Tyne, UK, 1999.
2. A. Lötzbeyer and R. Muhlfeld. Task Description of a Flexible Production Cell with Real Time Properties. Technical report, Forschungszentrum Informatik, Karsruhe, Germany - http://www.fzi.de/divisions/prost/projects/korsys/korsys.html, 1996.
3. M. Evangelist, N. Francez, and S Katz. Multiparty interactions for interprocess communication and synchronization. *IEEE Transactions on Software Engineering*, 15(11):1417–1426, 1989.
4. Y. J. Joung and S. A. Smolka. A comprehensive study of the complexity of multiparty interaction. *Journal of ACM*, 43(1):75–115, 1996.
5. R. H. Campbell and B. Randell. Error Recovery in Asynchronous Systems. *IEEE Transactions on Software Engineering*, 12(8):811–826, 1986.
6. A. Romanovsky, J. Xu, and B. Randell. Exception Handling and Resolution in Distributed Object-Oriented Systems. In *16th IEEE Int. Conf. on Distributed Computing Systems*, pages 545–552, Hong Kong, 1996. IEEE CS Press.
7. A. Burns, B. Randell, A. Romanovsky, R. Stroud, A. J. Wellings, and J. Xu. Temporal Constraints and Exception Handling in Object-Oriented Distributed Systems. *Design for Validation (DeVa) - Third Year Report, Esprit LTR Project 20072*, pages 3–25, Dezembro 1998.
8. A. Romanovsky, J. Xu, B. Randell, R. J. Stroud, and A. Burns. Analysis and Design of the Real-Time Production Cell. Technical report, Department of Computing Science (University of Newcastle Upon Tyne), UK, 1998.
9. B. Randell, A. Romanovsky, R. J. Stroud, J. Xu, and A. F. Zorzo. Coordinated Atomic Actions: from Concept to Implementation. Technical Report 595, Department of Computing Science (University of Newcastle Upon Tyne), UK, 1997.
10. J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. J. Stroud, and Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In *Proceedings of the 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25)*, pages 450–457, Pasadena, USA, 1995. IEEE CS Press.
11. L. A. Cassol. Projeto de uma Célula de Produção com Requisitos de Tempo Real usando DMIs. In *Workshop de Teses e Dissertações em Computação Tolerante a Falhas (SCTF'01)*, pages 37–42, Florianópolis, SC, Brasil, Março 2001.
12. L. A. Cassol. Tempo Real em Interações Multi-Participantes Confiáveis. Master's thesis, Departamento de Informática (PUCRS), Porto Alegre, RS, Brasil, 2001.
13. A. F. Zorzo, A. Romanovsky, J. Xu, B. Randell, R. J. Stroud, and I. S. Welch. Using Coordinated Atomic Actions to Design Safety-Critical Systems: a Production Cell Case Study. *Software - Practice and Experience*, 29(8):677–697, 1999.

Um Toolkit para Avaliação da Intrusão de Métodos de Injeção de Falhas

Patrícia Pitthan A. Barcelos[1] Taisy Silva Weber[2] Roberto Jung Drebes[3]

{pitthan@exatas.unisinos.br}
Instituto de Informática, UNISINOS
C. Postal 275, São Leopoldo – RS, Brasil

{pitthan, taisy, drebes@inf.ufrgs.br}
Instituto de Informática, UFRGS
C. Postal 15064, Porto Alegre – RS, Brasil

Abstract

This paper presents implementation and experiments of fault injection used to validate mechanisms of fault tolerance in communication protocols with temporal restrictions. It describes two approaches. The first implements fault injection through modifications in the application libraries, which ones support the target protocol. The second approach uses operating system resources. Also, the paper shows practical experiments using the fault injectors to validate a timeout mechanism of a communication protocol. The fault injectors are part of the INFIMO project, a toolkit to fault injection experimentation developed under Linux platform.
Keywords: Fault-tolerance, Validation, Fault Injection, Communication Protocol, Linux.

1 Introdução

Técnicas de tolerância a falhas visam aumentar a dependabilidade dos sistemas nos quais são empregadas. Entretanto, há necessidade de garantir a confiança na capacidade do sistema em fornecer o serviço especificado. A validação possui como objetivo propiciar esta garantia. Uma técnica de validação bastante utilizada é a injeção de falhas, que consiste na introdução controlada de falhas no sistema para observar seu comportamento. Injeção de falhas acelera a ocorrência das falhas em um sistema. Ao invés de esperar pela ocorrência espontânea das falhas, pode-se introduzi-las intencionalmente, controlando o tipo, a localização, o disparo e a duração. Injeção de falhas pode ser implementada por hardware, software ou simulação.

O artigo apresenta a validação, através da injeção de falhas por software, de um protocolo de comunicação. Enfoque especial é dado à intrusão resultante da inclusão do injetor junto ao protocolo, já que o mesmo apresenta restrições temporais que podem ser comprometidas pela atuação do injetor. Um *toolkit* para experimentos de intrusão da injeção de falhas é apresentado. O *toolkit*, denominado INFIMO (***IN****trusiveless* ***F****ault* ***I****njector* ***MO****dule*), visa analisar, de forma experimental, a intrusão temporal de injetores de falhas sobre um protocolo com característica tempo real.

A seção seguinte apresenta aspectos referentes a injeção de falhas por software e a intrusão de injetores de falhas. A seção 3 descreve o INFIMO *toolkit* e a arquitetura dos injetores de falhas. Na seção 4 é apresentado o protocolo INFIMO_TAP, enquanto nas seções 5 e 6 são descritos os injetores INFIMO_LIB e INFIMO_DBG, respectivamente. A seção 7 descreve experimentos de intrusão e a seção 8 esboça algumas conclusões.

2 Injeção de Falhas por Software

Injeção de falhas por software emula falhas de hardware através de software corrompendo o estado do programa em execução. Sob o ponto de vista de implementação, o injetor é um processo, conjunto de rotinas ou objeto, que interrompe ou altera a execução do sistema e executa seu próprio código, emulando falhas pela inserção de erros no sistema.

Em sistemas com característica tempo real, onde as aplicações por eles controladas devem ocorrer em instantes de tempo relativos a uma base de tempo externa, a injeção de

[1] Doutora em Ciência da Computação (UFRGS, 2001)
[2] Doutora em Informática (Univ. Karlsuhe, 1986)
[3] Bacharel em Ciência da Computação (UFRGS, 2001)

falhas se torna mais crítica. As restrições temporais impostas pelos sistemas aliadas a sobrecarga provocada pelo injetor de falhas justificam este fato.

2.1 Intrusão

Por representar uma carga extra no sistema, o injetor de falhas pode alterar o tempo de execução do protocolo, comprometendo suas restrições temporais. Esta execução de carga extra é referenciada na literatura como intrusão [CUN00]. Assim, durante o desenvolvimento de um injetor, cuidados especiais devem ser tomados para determinar a carga que esse representa e minimizar seus efeitos. Na maioria dos modelos de falhas, há necessidade de executar um código adicional no mesmo processador que executa a aplicação alvo. Como conseqüência, pode-se chegar a situações onde alguns limites de tempo são perdidos devido ao tempo de execução extra por parte das atividades de injeção de falhas [CAR98].

A intrusão do injetor de falhas no protocolo alvo pode ser observada de forma temporal e espacial. Na intrusão temporal tem-se o tempo de execução aumentado em virtude do acréscimo das atividades de injeção de falhas, as quais devem ser executadas juntamente com o protocolo alvo. Já na intrusão espacial a necessidade de modificar o código do protocolo alvo pode consistir em comportamento intrusivo. O artigo enfoca a análise da intrusão temporal do injetor de falhas sobre o protocolo alvo.

A intrusão temporal é avaliada de acordo com o *tempo de execução do protocolo*. No trabalho apresentado neste artigo, este tempo é medido através do relógio local do processador no qual o protocolo está executando. Para cada experimento toma-se o valor do relógio antes da execução do protocolo e após o término da mesma. Para se analisar a carga do sistema executa-se, para os mesmos dados de entrada, os seguintes experimentos: execução somente do protocolo alvo, execução do protocolo alvo e do injetor inativo e execução do protocolo alvo e do injetor ativo.

3 INFIMO *Toolkit*

O INFIMO (***IN****trusiveless* ***F****ault* ***I****njector* ***MO****dule*) é um *toolkit* para experimentos de injeção de falhas na validação da comunicação em protocolos com restrições temporais. O principal objetivo da implementação do INFIMO é analisar a intrusão imposta pelas atividades de injeção de falhas no comportamento temporal do protocolo alvo.

Este trabalho compreende dois métodos de validar um protocolo de comunicação através da injeção de falhas. Com isso, busca-se analisar a interferência das atividades de injeção de falhas no comportamento do protocolo alvo, o qual possui característica tempo real. O primeiro método de validação utiliza-se das API's do protocolo alvo (biblioteca JRTPLIB), enquanto o segundo faz uso de recursos do sistema operacional.

O INFIMO é implementado em plataforma Linux. Os experimentos apresentados, assim como a implementação do *toolkit*, foram realizados em computadores PC Pentium MMX 233Mhz, 64MB de RAM, conectados via Ethernet 10Mbps, rodando Linux 2.2.14.

O *toolkit* é formado por três componentes: um protocolo alvo e dois injetores de falhas. O protocolo, denominado INFIMO_TAP (***IN****trusiveless* ***F****ault* ***I****njector* ***MO****dule* – ***TA****rget* ***P****rotocol*), foi desenvolvido para experimentos com o *toolkit* e encontra-se descrito na seção 4. Os injetores de falhas INFIMO_LIB (***IN****trusiveless* ***F****ault* ***I****njector* ***MO****dule – by* ***LIB****rary modifications*) e INFIMO_DBG (***IN****trusiveless* ***F****ault* ***I****njector* ***MO****dule – using* ***DeBuG*** *resources*) são apresentados nas seções 5 e 6.

Neste contexto, cabe ressaltar três objetivos: do INFIMO, do protocolo alvo e dos injetores de falhas. O INFIMO visa analisar a intrusão dos injetores de falhas INFIMO_LIB e INFIMO_DBG sobre o protocolo alvo INFIMO_TAP. O protocolo INFIMO_TAP viabiliza a

troca de pacotes entre processos, controlando, através do mecanismo de detecção do *timeout*, a recepção destes pacotes. Os injetores INFIMO_LIB e INFIMO_DBG têm como objetivo validar o mecanismo de detecção do *timeout* implementado pelo protocolo alvo.

3.1 Arquitetura do INFIMO

A Figura 1 apresenta a arquitetura dos injetores INFIMO_LIB e INFIMO_DBG, baseada na arquitetura genérica para injetores de falhas proposta por Hsueh [HSU97].

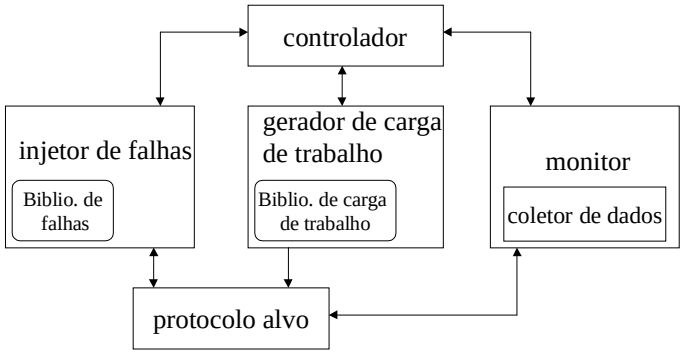


Figura 1 – Arquitetura de Injeção de Falhas do INFIMO

Os injetores do INFIMO implementam os componentes da arquitetura da Figura 1 através de procedimentos incorporados aos códigos dos mesmos. O protocolo INFIMO_TAP oferece procedimentos que realizam as funções do *gerador de carga de trabalho* e da *biblioteca de carga de trabalho*. Os injetores INFIMO_LIB e INFIMO_DBG possuem procedimentos que executam as funções dos componentes: *injetor de falhas*, *biblioteca de falhas*, *controlador*, *monitor* e *coletor de dados*.

O *protocolo alvo* é a um protocolo de comunicação que apresenta restrições temporais e possui um mecanismo de tolerância a falhas, o mecanismo de detecção de *timeout*.

Em ambos os injetores de falhas, a *biblioteca de falhas* pertence ao *injetor de falhas*. Esta biblioteca possibilita a especificação do modelo de falhas, composto pelas informações referentes ao tipo, localização, duração e disparo das falhas.

Para o INFIMO_LIB, o *injetor de falhas* é implementado através de alterações nas funções da biblioteca JRTPLIB, a qual é implementada sobre o protocolo RTP. As alterações realizadas visam a injeção de falhas de acordo com a especificação da biblioteca de falhas. No INFIMO_DBG, o *injetor de falhas* usa o *ptrace( )* para injetar as falhas no protocolo alvo. O *ptrace( )* permite que o injetor controle e manipule a execução do protocolo.

O *gerador de carga de trabalho* estabelece o número de processos participantes da comunicação e o número de pacotes envolvidos. O gerador de carga de trabalho conta com o auxílio da *biblioteca de carga de trabalho*.

O *controlador* é um procedimento que dispara a injeção de falhas. No INFIMO_LIB o disparo é feito a cada acesso à biblioteca alterada. No INFIMO_DBG o disparo é realizado a cada chamada ao *ptrace( )*. Em ambos os injetores, o disparo é realizado com base na especificação do modelo de falhas a ser validado.

O *monitor* é um procedimento que observa o andamento do experimento, coleta dados sobre o mesmo e salva-os em arquivos de *log*. O *coletor de dados* pertence ao *monitor*.

Em ambas as implementações, os injetores de falhas são processos criados pelo sistema operacional através da linha de comando. O processo que implementa o injetor de falhas INFIMO_LIB incorpora em seu código fonte o código do protocolo INFIMO_TAP. O

processo que implementa o injetor INFIMO_DBG cria um processo filho, o processo a ser monitorado. Este processo executa o código do protocolo alvo INFIMO_TAP.

4 Protocolo alvo: INFIMO_TAP

O INFIMO_TAP é o alvo da validação por injeção de falhas. O INFIMO_TAP é um protocolo coordenado de troca de pacotes, que possui restrições temporais e tolerância a falhas. Para sua implementação, faz-se uso das funções da biblioteca JRTPLIB [JRT99], a qual é implementada sobre o protocolo RTP (*Realtime Transport Protocol*) [SCH97]. O mecanismo de tolerância a falhas do INFIMO_TAP consiste na detecção do *timeout* associado aos pacotes. O objetivo da validação é a detecção do *timeout*.

A execução do protocolo é disparada pela linha de comando, através da qual o sistema cria os processos do protocolo. Um processo do INFIMO_TAP corresponde a um conjunto de *threads*. Para cada processo existe uma *thread* principal, uma *thread* de resposta e *n*-1 *threads* de envio, onde *n* é o número de processos. Associado à *thread* principal há o *signal_handler*, uma espécie de vetor de interrupções responsável pela coordenação do *timer*.

O *gerador de carga de trabalho* é responsável pela criação das *threads*, sua associação aos processos, e geração do número de pacotes a serem enviados. O destino dos pacotes é irrelevante. Após a geração da carga de trabalho, dá-se início à comunicação propriamente dita. A Figura 2 ilustra a seqüência de envio de um pacote A do processo 2 para o processo 4. O envio se dá através de uma *thread* de envio do processo origem (processo 2), enquanto a recepção é feita pela *thread* de resposta do processo destino (processo 4). A *thread* de envio do processo 2, responsável pelo destino 4 é a *thread* de envio 4.

Conforme a Figura 2, no tempo t1, a *thread* de envio 4 (processo 2) envia o pacote A. Ao enviar o pacote, a *thread* 4 adiciona um *timer* ao pacote e coloca este *timer* em uma fila. No tempo t2, a *thread* de resposta (processo 4) recebe o pacote A, armazena-o e envia um ACK ao processo 2. O ACK é recebido pela *thread* de resposta (processo 2) no tempo t3. Então, a *thread* de resposta (processo 2) remove o *timer* do pacote A da fila. Toda a seqüência de envio de pacote e recebimento do ACK é realizada antes do *timeout* expirar (tempo t4). As ações de cada processo do protocolo são armazenadas em arquivos de *log*.

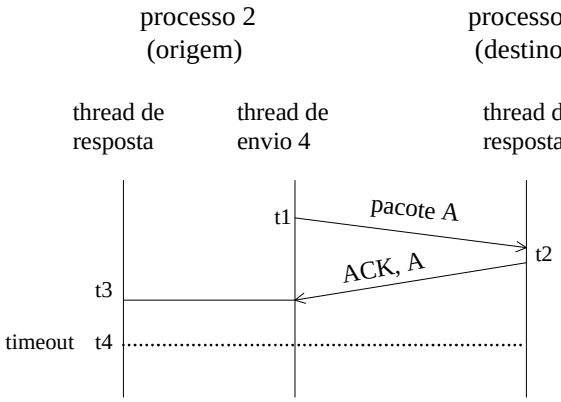


Figura 2 – INFIMO_TAP: Envio de pacote

5 INFIMO_LIB

O INFIMO_LIB é um injetor de falhas implementado através de alterações nas funções da biblioteca JRTPLIB sobre a qual está implementado o protocolo alvo (INFIMO_TAP). Assim como no INFIMO_TAP, no INFIMO_LIB também existe um conjunto de processos, onde cada um possui um conjunto de *threads*. Entretanto, enquanto os

processos do INFIMO_TAP utilizam as funções da biblioteca JRTPLIB, os processos do INFIMO_LIB usam funções modificadas da JRTPLIB.

Enquanto o INFIMO_TAP corresponde a um protocolo de troca de pacotes, o INFIMO_LIB corresponde ao INFIMO_TAP acrescido de atividades de injeção de falhas. O INFIMO_LIB mantém a estrutura de funcionamento do INFIMO_TAP referente a: disparo dos processos, criação das *threads*, geração da carga de trabalho, seqüência de envio e reenvio, manipulação dos *timers* e geração de *logs*.

A execução do INFIMO_LIB é disparada pela linha de comando. O disparo do INFIMO_LIB deve ser feito para cada processo do protocolo. Ao ser executado, o INFIMO_LIB, através da *biblioteca de falhas*, interpreta o modelo de falhas, possibilitando assim, a execução efetiva da injeção de falhas. Os processos do INFIMO_LIB executam conforme os processos do INFIMO_TAP. De acordo com a especificação do modelo de falhas, a comunicação se dá através da execução de funções da JRTPLIB modificadas, ou seja, são executadas as funções de injeção de falhas do INFIMO_LIB.

Em ambos os injetores de falhas, para emular uma falha de *crash* no processo alvo, todos os pacotes oriundos deste processo são desviados para um destino não válido. O mesmo acontece na emulação de uma falha por omissão, entretanto, neste caso o desvio não ocorre para todos os pacotes do processo alvo, porém somente para alguns. O atraso pode ser emulado através da retenção do pacote por um determinado período de tempo.

O histórico de ações de cada processo do INFIMO_LIB é armazenado em arquivos de *log*. Este procedimento é realizado pelo *coletor de dados*, que atua em conjunto com o *monitor*. O *controlador* é o responsável por todo o gerenciamento do INFIMO_LIB, desde a inicialização do protocolo alvo até a coleta de dados do experimento.

Um aspecto a ser considerado na implementação desta abordagem se refere à integridade e, neste sentido, destacam-se duas preocupações. A primeira relaciona-se à integridade do protocolo alvo. Alterações nas funções da JRTPLIB não devem interferir no contexto do protocolo. A segunda preocupação refere-se à manutenção da semântica das funções da biblioteca JRTPLIB. Deve-se garantir que as alterações impostas pelas atividades de injeção de falhas não comprometam o comportamento das funções originais. Outro aspecto a ser considerado é a portabilidade da abordagem. A idéia é poder utilizar esta abordagem para validação de outros protocolos alvo, implementados também sobre a JRTPLIB.

Entretanto, a abordagem apresenta diminuição do controle da carga de trabalho, uma vez que se está implementando o injetor em um nível mais alto. Assim, a carga de trabalho é provavelmente mais alta do que se a implementação fosse desenvolvida em baixo nível, como no nível do sistema operacional.

6 INFIMO_DBG

O INFIMO_DBG baseia-se na utilização dos recursos de depuração apresentados pelo sistema operacional. O injetor utiliza o *ptrace( )* (padrão Unix) para desviar das operações normais do protocolo alvo, o INFIMO_TAP, para as atividades de injeção de falhas.

O mecanismo de injeção do INFIMO_DBG é manipulado através da interceptação da comunicação de dois processos que executam concorrentemente: processo injetor de falhas e processo alvo. O controle de execução exercido pelo injetor de falhas permite a injeção das falhas através das chamadas de sistema executadas pelo processo alvo.

A inicialização do INFIMO_DBG é similar a do INFIMO_LIB. A execução do INFIMO_DBG é disparada pela linha de comando. O disparo do INFIMO_DBG deve ser feito para cada processo do protocolo.

Ao ser executado, o INFIMO_DBG, através da *biblioteca de falhas*, interpreta o modelo de falhas. Um dos processos do protocolo assume a responsabilidade de injetar falhas,

tornando-se o processo injetor de falhas. O processo injetor de falhas tem total controle sobre o processo alvo podendo, portanto, atuar sobre o mesmo.

A Figura 3, que ilustra o funcionamento do INFIMO_DBG, apresenta um conjunto de processos: processo injetor de falhas, processo alvo e demais processos do protocolo. O processo injetor de falhas cria, através da chamada de sistema *fork( )*, o processo alvo. O processo alvo executa o mesmo código dos demais processos do protocolo. As ações do processo alvo podem ser interceptadas pelo processo injetor de falhas. Através da interceptação, o processo injetor de falhas decide, com base no modelo de falhas, sobre as atividades de injeção de falhas.

Na Figura 3 são mostradas duas situações de intercepção do processo alvo por parte do processo injetor de falhas. Na primeira, o injetor de falhas decide pela omissão do pacote enviado pelo processo alvo para um dos processos do protocolo. Esta situação ilustra os tipos de falha *crash* ou omissão. Na segunda interceptação, o injetor supõe o atraso no envio do pacote. Esta situação ilustra uma falha de temporização.

Ao ser executado, o processo alvo recebe do processo injetor de falhas os argumentos referentes ao modelo de falhas, os quais possibilitam a execução efetiva da injeção de falhas. O histórico de ações de cada processo do protocolo é armazenado em arquivos de *log*. O *monitor* e o *coletor de dados* são os responsáveis por esta função.

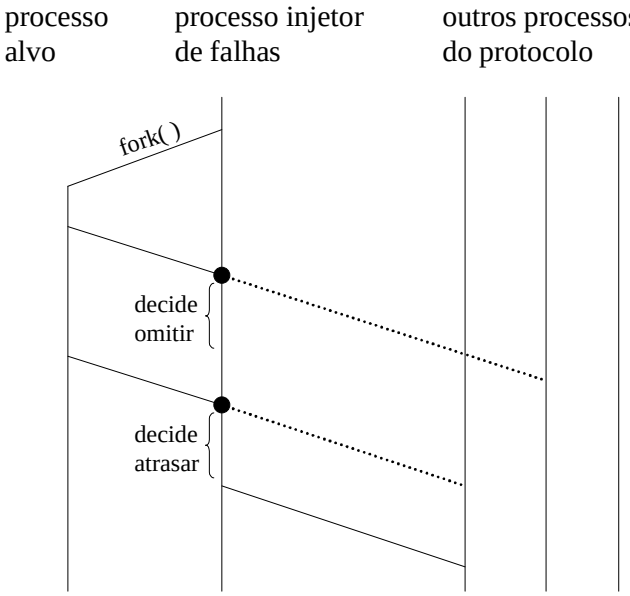


Figura 3 – Interceptação do processo alvo

O INFIMO_DBG baseia-se no uso do depurador do sistema operacional através da execução do protocolo alvo da validação até a chamada de sistema *sendto( )*. No momento em que encontra a chamada *sendto( )*, o INFIMO_DBG pode alterar os parâmetros da mesma para injetar a falha desejada. A coordenação das atividades do INFIMO_DBG é realizada pelo *controlador*, de acordo com a arquitetura exibida na seção 3.1.

Para utilizar o *ptrace( )*, o INFIMO_DBG alterou a estrutura de *threads* do INFIMO_TAP. Para o INFIMO_TAP há uma *thread* principal, que cria uma *thread* de resposta, a qual cria *n threads* de envio. A necessidade do *ptrace( )* atuar sobre o nível mais alto de *threads* fez com que as *threads* de envio fossem incorporadas à *thread* principal.

Esta abordagem de implementação apresenta como vantagens a portabilidade e a integridade. Por ser um injetor embasado no padrão Unix e utilizar recursos do sistema operacional para a injeção de falhas, é possível sua utilização para validação de outros

protocolos, implementados ou não sobre a JRTPLIB, porém compatíveis com o ambiente Unix. Quanto à integridade pode-se afirmar que o contexto do protocolo alvo não sofre interferência em função da subordinação ao injetor de falhas.

7 Intrusão temporal dos injetores de falhas

Esta seção descreve experimentos que mostram os *tempos de execução* dos injetores INFIMO_LIB e INFIMO_DBG, ou seja, a carga imposta pelas atividades dos injetores de falhas sobre o protocolo INFIMO_TAP. Consideram-se 3 processos, onde os processos livres de falhas enviam 27 e 24 pacotes e processo alvo envia 30 pacotes.

Os experimentos baseiam-se no modelo de falhas apresentado na Tabela 1. O tipo de falha corresponde ao que corromper e como corromper. O INFIMO enfoca falhas de comunicação, cuja localização inclui processos ou *links*. As falhas de comunicação ocorrem no envio de pacotes. A forma pela qual um processo (ou *link*) pode ser corrompido compreende a classe de falhas a ser validada, que pode ser falha de *crash* ou omissão. O INFIMO possui ainda suporte para implementação de falhas de temporização.

O disparo da falha relaciona-se a uma condição espacial ou temporal. Uma falha disparada espacialmente é ativada quando o protocolo alcança determinado estado. Uma falha disparada temporalmente torna-se ativa após um determinado período de tempo. O INFIMO possibilita ainda a definição da duração da falha, que pode ser transiente ou intermitente.

Tabela 1 – Modelo de falhas dos experimentos de intrusão

Argumento	**Descrição**
Tipo de falha	Falhas por omissão (falhas simples)
Localização da falha	Processo
Disparo da falha	Disparo espacial (a cada 2 pacotes)
Duração da falha	Falhas intermitentes

A Tabela 2 apresenta os experimentos relativos a intrusão dos injetores de falhas INFIMO_LIB e INFIMO_DBG. Os valores de tempo de execução exibidos são expressos em milisegundos. O valor definido para o *timeout* é de 500 ms.

Tabela 2: Experimentos de intrusão dos injetores

	INFIMO_LIB		**INFIMO_DBG**	
	Processos livres de falha	Processo alvo	Processos livres de falha	Processo alvo
a) somente protocolo	37,443	-	37,443	-
b) protocolo + injetor inativo	36,876	-	40,177	-
c) protocolo + injetor ativo				
1 falha	40,091	666,641	40,152	647,589
2 falhas	39,590	691,139	39,825	678,759
4 falhas	39,748	674,899	39,818	679,683
8 falhas	39,321	682,603	39,680	687,102
12 falhas	39,516	703,460	38,809	688,042

O experimento “a” (linha 3 da Tabela 2) apresenta a carga do protocolo alvo, ou seja, é medido o tempo de execução do protocolo alvo. Todos os processos estão livres de falhas, uma vez que o protocolo não está sendo validado por nenhum injetor de falhas.

No experimento “b” (linha 4 da Tabela 2) o protocolo e o injetor estão executando, porém o injetor de falhas executa com máscara de injeção nula. Assim, todas as atividades do injetor são realizadas, exceto a injeção propriamente dita. Em ambos os injetores de falhas, os

valores de tempo de execução mantém-se razoavelmente próximos. O INFIMO_DBG apresenta um pequeno acréscimo no seu tempo de execução, porém não implica em sobrecarga significativa. Este acréscimo está provavelmente relacionado à criação do processo injetor de falhas e as trocas de contexto entre os processos participantes do experimento.

O experimento "c" (linhas 5-9 da Tabela 2), apresenta a carga imposta pela injeção de falhas. Tanto no INFIMO_LIB como no INFIMO_DBG, para os processos livres de falhas, os tempos de execução obtidos para o envio de 27 e 24 pacotes, mantêm valores aproximados. Para todos os experimentos, os quais introduzem de 1 a 12 falhas, o *timeout* expirou. Considerando o processo alvo da injeção de falhas, percebe-se que os valores de tempo apresentados sofrem um elevado acréscimo logo após a injeção da primeira falha. Nesse acréscimo há que se considerar o valor do *timeout* (500 ms). Logo, o processo leva, em média, 183,748 ms para manipular a falha. Esta manipulação inclui, além da detecção da falha, a cópia do pacote, a remoção de seu *timer* da fila de *timers*, o reenvio do pacote e o acréscimo do novo *timer* no final da fila de *timers*. Observa-se ainda que o aumento progressivo no número de falhas injetadas implica em mínima variação no tempo de execução.

8 Conclusões

Diversas abordagens de implementação de injetores de falhas têm sido propostas na literatura. O diferencial do INFIMO concentra-se na característica tempo real do protocolo alvo, no escopo de falhas a ser validado e na idéia de *toolkit*, a qual possibilita a comparação entre duas abordagens de implementação.

Protocolos com característica tempo real constituem um desafio no contexto de injeção de falhas. Diferentemente de grande parte dos injetores de falhas ([CAR98], [CUN00], [KAN95], [KRI98]), os injetores do INFIMO enfocam falhas de comunicação, injetando falhas no envio de pacotes. RT_Xception [CUN00], apesar de tratar tempo real, se restringe a falhas de memória e processador, não se prestando à validação de protocolos de comunicação.

Além da condução de experimentos de injeção de falhas, o INFIMO caracteriza-se principalmente pela sua modularidade, possibilitando a expansão do *toolkit*. Por estar implementado sobre plataforma Linux e utilizar biblioteca não comercial, pode-se propor outros protocolos alvo e outras abordagens de implementação para o INFIMO. O INFIMO *toolkit* pode auxiliar ao desenvolvedor de protocolos de comunicação com característica tempo real na utilização dos injetores de falhas como técnica de validação.

Bibliografia

[CAR98] CARREIRA, J.; MADEIRA, H.; SILVA, J. G. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. IEEE TOSE, v. 24, n. 2, Fev. 1998.

[CHI89] CHILLAREGE, R.; BOWEN, N. S. Understanding Large-System Failures: A Fault-Injection Experiment. FTCS 19, Los Alamitos, California, p.356-363, 1989.

[CUN00] CUNHA, J. C.; RELA, M. Z.; SILVA, J. G. RT-Xception: Fault-Injection for Real-Time. Coimbra: Centro de Informatica e de Sistemas da Univ. de Coimbra, 2000. (TR-2000/002).

[HAN95] HAN, S. et. al. DOCTOR: an integrateD sOftware fault injeCTion enviRonment for Distributed RT systems. Int. Comp. Performance and Dependability Symp., Erlangen, Germany, 1995.

[HSU97] HSUEH, M. et. al. Fault Injection Techniques and Tools. IEEE Computer, v. 30, n. 4, Apr. 1997.

[JRT99] LIESENBORGS, J. Manual da JRTLIB. http://limumba.luc.ac.be/jori/jrtplib/jrtplib.html

[KAN95] KANAWATTI, G. A. et. al. FERRARI: A flexible software-based fault and error injection system. IEEE Transactions on Computers, v. 44, n. 2, Feb. 1995.

[KRI98] KRISHNAMURTHY, N. et. al. A design methodology for software fault injection in embedded systems. Int. Workshop Dependable Comp. and Applications, Johannesburg, South Africa, 1998.

[SCH97] SCHULZRINNE, H. G. Protocolo RTP. http://www.cs.columbia.edu/~hgs/RTP.

A Hierarchical Failure Detection Service with Perfect Semantics*

Francisco V. Brasileiro, Jorge C. A. de Figueiredo, and Lívia M. R. Sampaio

Universidade Federal da Paraíba
Departamento de Sistemas e Computação
Av. Aprígio Veloso, 882
58.109-970 Campina Grande, Paraíba, Brazil
Tel: (+55) 83 310 11 19 Fax: (+55) 83 310 11 24
{fubica,abrantes,livia}@dsc.ufpb.br

Abstract

A *failure detector* is an important abstraction to support the implementation of higher level fault-tolerant protocols on distributed asynchronous systems. In this paper we show, via a counter example, that using the best possible failure detector of a given class is not always the key to achieve the best performance for a specific higher level consensus protocol. We argument that this behaviour is due to structural limitations of the consensus protocol that are unlikely to be circumvented, unless stronger abstractions are provided. Thus, we advocate that the designer of a generic failure detection service should concentrate her efforts in implementing the strongest failure detector possible - even if it is not the best within its class, instead of trying to implement the best failure detector of a weaker class. Following this philosophy, we present the basis of the design of a hierarchical failure detection service with the strongest semantics known, namely that of a perfect failure detector.

1 Introduction

In the last few years, with the increasing popularity of distributed applications, the number of applications with dependability requirements has greatly increased. For these applications, failures in the components that form the system can cause undesirable consequences, such as loss of income, clients, information and confidentiality. To attain their dependability requirements, these applications must execute extra mechanisms that are able to tolerate faults.

*This work is supported by grants from CTPETRO and CNPq/Brazil (300.904/94-0 and 300.646/96).

Most off-the-shelf infrastructures for deploying distributed applications are characterised by the absence of upper bounds on both message transmission and scheduling delays, *i.e.* they are asynchronous systems. Unfortunately, a well known impossibility result presented in [FLP85] shows that it is impossible to reach consensus [Fis83] (a basic building block for many fault tolerance mechanisms) in a pure asynchronous distributed system subject to failures. The following observation is the core of the impossibility result presented in [FLP85]: due to the uncertainty on communication and scheduling delays, it is impossible to differentiate a processor that has failed from one that is simply slow.

In face of this result, a number of models that strengthen the pure asynchronous distributed system model, allowing solutions for the consensus problem have been defined. Among them, the asynchronous distributed system model augmented by an unreliable failure detector [CT96] has been widely studied. This system model assumes the existence of an "oracle", named a failure detector, that is able to give an idea of which processors have crashed. Although this oracle may make mistakes (*e.g.* by suspecting a correct process), the information it provides is precise enough to allow deterministic solutions for the consensus problem [CT96].

The semantics of a failure detector is characterized by two properties namely: *completeness* and *accuracy*. The first property defines how broad is the reach of the failure detector, while the other restricts the mistakes that the failure detector may make. The stronger the semantics of the failure detector, the less restrictive the assumptions required to guarantee the correctness of the higher level protocol [CT96] and, possibly, the more efficient the protocols. On the other hand, one can argue that the weaker the semantics of the failure detector, the simpler its implementation. In [CHT96] it is proved that the weakest class of failure detectors that allows a solution to the consensus problem is named $\diamond S$[1] and is defined by the following properties [CT96]:

- **strong completeness**: eventually every process that crashes is permanently suspected by every correct processes.
- **eventual weak accuracy**: there is a time after which some correct process is never suspected by any correct processes.

Following this result, several implementations of $\diamond S$ failure detectors have been reported in the literature. To allow the comparison of different implementations of failure detectors, three primary quality of service (QoS) metrics have been defined [CTA00] for the failure detector module of a process q (FD_q) that monitors a process p:

- **detection time (T_D)**: it is the time that elapses since p has failed until it is permanently suspected by FD_q.
- **mistake duration (T_M)**: it is the time that takes for FD_q to stop mistakenly suspecting a correct p.

[1] In fact, [CHT96] proves that $\diamond W$ is the weakest failure detector to solve consensus, however [CT96] shows that $\diamond W$ and $\diamond S$ are equivalent classes of failure detectors.

- **mistake recurrence time** (T_{MR}): it is the time elapsed between two consecutive mistakes of FD_q.

Intuitively, the best $\diamond S$ failure detector module would have: i) the smallest T_D; ii) the smallest T_M; and iii) the largest T_{MR}. Unfortunately, the use of better failure detectors does not necessarily guarantee better performance to the higher level protocols. In this paper we show, via a counter example, that the performance of a consensus protocol based on a $\diamond S$ failure detector may improve when the QoS of the failure detector it uses worsens. This is because the performance of the protocol is dependent on the performance of a process that plays the role of the round coordinator that first reaches a decision.

A careful design of a failure detection service can increase the performance of higher level protocols, however this can only be achieved by using *ad hoc* implementations tailored for a particular setting and protocol [SDS99]. Therefore, we advocate that the designer of a generic failure detection service should not strive to implement the best failure detector possible of a given class. Rather, she should concentrate her efforts in building *any* failure detector of the *strongest* class possible. Following this idea we present in this paper the design of a hierarchical failure detection service of the class P (perfect) [CT96]. The class of perfect failure detectors is the strongest class among those proposed in [CT96]. We believe that it is possible to build consensus protocols on top of a perfect failure detector that not only are able to tolerate more faults, but are also simpler and more efficient than those built on top of a $\diamond S$ one.

The rest of the paper is structured as follows. Section 2 discusses the basis of our argument on the inadequacy of trying to implement better failure detectors to provide generic failure detection services. Then, in Section 3 we present the basis of the design of a hierarchical failure detection service that provides the semantics of a perfect failure detector. Finally, Section 4 concludes the paper with our final remarks.

2 The Counter Example

The counter example uses one of the consensus protocols based on the rotating coordinator paradigm presented in [CT96]. We will use the protocol that is supported by a $\diamond S$ failure detector and whose functioning can be summarised as follows. n processes, from which at most $\lfloor (n-1)/2 \rfloor$ may fail, participate in the consensus. Each process has access to its local $\diamond S$ failure detector module that gives it hints about which processes might have failed.

The protocol is executed in asynchronous rounds and it is assumed that all processes have an a priori knowledge of the identity of the process that plays the role of the coordinator of each round. Within each round the protocol proceeds in the following four phases. In the first phase every process sends its estimate of the decision value to the current round coordinator. In the second phase the round coordinator gathers $\lceil (n+1)/2 \rceil$ estimates, chooses one of them[2] and send it to all processes as their new estimate value. In phase three processes wait for the new estimate from the round coordinator. To avoid the possibility

[2]This choice must respect a locking mechanism that is not important for the purpose of this paper; the interested reader should refer to [CT96].

of blocking due to a faulty coordinator, processes constantly query their failure detector module to assess the round coordinator status. If the round coordinator is suspected the process sends a *nack* message to the round coordinator (notice that a suspicion does not mean that the round coordinator has indeed failed). On the other hand, if it receives the new estimate value from the round coordinator it adopts the new estimate value and sends an *ack* message to the round coordinator. In phase 4 the round coordinator collects $\lceil (n+1)/2 \rceil$ replies (*acks* and *nacks*) and if all replies are *acks* it decides for the estimate value it has proposed. The processes are informed of the decision via the execution of a reliable broadcast protocol [CT96]. A process finishes the execution of the protocol when it reliably delivers the decision value.

As discussed in Section 1, based on the QoS metrics proposed in [CTA00], improvements on the performance of a $\diamond S$ failure detector (or any failure detector, for that matter) can be achieved by reducing T_D, reducing T_M, or increasing T_{MR}. Since in our example we will only consider the most frequent runs where processes are not faulty, we will disregard T_D.

We will first consider all runs of the consensus protocol described above where processes are non-faulty, and their $\diamond S$ failure detector modules behave in such a way that $T_M = 0$ and $T_{MR} = \infty$. This is to say that in the runs selected the failure detector modules do not make mistakes and behave as failure detector modules of class P. Considering only these runs, and the QoS metrics used, it is not possible to implement better $\diamond S$ failure detector modules. Let us assume that process p is the coordinator of the first round of all executions of the consensus protocol and that for some reason p (or its communication with the other processes) is much slower that the other processes (say k times slower). Since the failure detector modules do not make mistakes, p will never be suspected by any process and the performance of the protocol will be hugely influenced by the performance of p.

Now consider the same runs as above but supported by worse $\diamond S$ failure detector modules. Let us assume that these failure detector modules have a non-zero T_M and a much smaller T_{MR} and, because of that, causes all other processes to suspect p and advance to round two. Since all processes are non-faulty the faster processes that advanced to round two can reach a decision in that round without the aid of p. For that to happen T_{MR} must be such that all failure detector modules mistakenly suspect the slow p but do not suspect any of the other faster processes. Given the time required to execute the consensus protocol and the speed of the faster processes it is not difficult to find values for T_{MR} and k such that the second scenario will always outperform the first one.

We have modeled the consensus protocol based on the rotating coordinator paradigm presented in [CT96] by means of a Coloured Petri Net (CPN). Without loss of generality, we have modeled a 3-process instance of the referred consensus protocol that is able to tolerate one process failure. In order to make performance analysis of the protocol, we simulated the CPN model using the Design/CPN tool [CPN93]. Two scenarios were analysed, by tunning the failure detector QoS metrics previously discussed and using different communication delays between the processes. First, we have set the T_M and T_{MR} to 0 and ∞, respectively. Such a configuration corresponds to a failure detector that does not make mistakes (a failure detector that behaves as a perfect one). By increasing the T_M value and reducing the T_{MR} value we have worsen the QoS of the $\diamond S$ failure detector. We have simulated the two configurations within three situations, namely: i) communication delays between processes

is the same; ii) communication delays between the coordinator and processes is 5 times slower than between the other processes; and iii) communication delays between the coordinator and processes is 10 times slower than between the other processes. Table 1 shows the consensus time[3] obtained in the simulations (time is expressed in ms).

T_M	T_{MR}	$Delay_{coordinator->processes}$	$Delay_{processes->processes}$	Consensus Time
0	∞	1	1	4
0	∞	5	1	20
0	∞	10	1	40
1	50	1	1	4
1	50	5	1	8
1	50	10	1	13

Table 1: Results of the Consensus Simulation Using Different $\diamond S$ Failure Detectors

The results obtained through these simulations show that the performance of the protocol is hugely influenced by the performance of the round coordinator. They also demonstrate that in some situations, considering a worse $\diamond S$ failure detector yields better performance for the consensus protocol. Further, they suggests that it is not easy to assess the impact of a failure detector on the performance of an application based solely on the QoS metrics proposed by [CTA00].

It may well be possible that there exists a consensus protocol based on a $\diamond S$ failure detector that does not possess the unwanted property discussed above. However, it is more likely that one such protocol would require a stronger failure detector. In the next section we propose the basis of the design of a failure detection service with a perfect semantics, which is the strongest semantics known for failure detectors.

3 A Hierarchical Failure Detection Service

In this section we propose a hierarchical failure detection service (FD service) for wide area networks (WANs). Initially, we will consider that the service runs over a non-partitionable WAN.

The proposed FD service has a perfect semantics (*i.e.* it implements a failure detector of class P) and is structured in two levels: the local level (within a local area network - LAN, or a segment of a LAN) and the global level (within the WAN that connects all local level segments). Each LAN segment is referred to as a local domain of detection, which is supervisioned by a local FD service. A set of local domains of detection connected via a WAN is named a federation of detection, which is supervisioned by a global FD service. Figure 1 illustrates this structuring.

In the local domain of detection, a perfect failure detector can be implemented by adding an extra communication channel for the network nodes in the domain. Such a channel will be used exclusively to convey traffic of the local FD service. This implies that the communi-

[3]Consensus time was measured as the time required for two processes to decide.

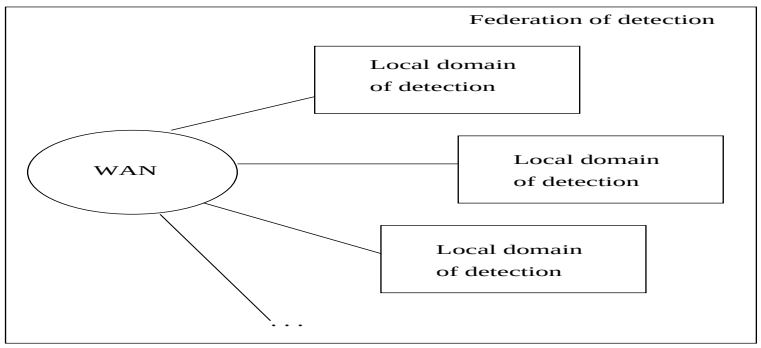


Figure 1: The Structuring of a Hierarchical Failure Detection Service

cation traffic generated through this channel is known and controlled [AV96], and therefore the maximum delay to transmit messages using this channel can be easily accounted.

To guarantee that the end-to-end maximal communication delay between any two functioning modules of the local FD service is bound, one needs to ensure that these modules are scheduled at the required time. One way to achieve that is using a real-time operating system [VCF00]. Another approach is to use a conventional operating system and implement the failure detection service within the system kernel (*e.g.* as a kernel thread), or as a user process that runs with maximal priority. In both approaches, by appropriately choosing the time interval between the transmission of two consecutive heartbeat messages, it is possible to guarantee that the maximal transmission delays in the redundant channel will never be violated [AV96, CB97]. This characterises a synchronous system within which implementing a perfect failure detector is a trivial task.

Our target system is composed of a number of Ethernet-based local area networks of PCs running Linux and connected by some routers. We have envisaged two strategies to implement the extra channel required at each local domain of detection. The first requires each machine in a local domain of detection to have an extra Ethernet card. These cards are connected to form a redundant communication channel (when recabling is not suitable, a wireless LAN can be used). Notice that the FD service does not impose any extra load in the asynchronous communication channel used by applications. The only cost it imposes is that related to the processing time required to execute each module of the FD service at the corresponding machine. The second approach further reduces this cost. It uses a failure detector device attached to each machine. This device has very simple processing and wireless communication capabilities. Whenever a machine is initialized the device starts detecting the other machines that are up (by listening to their heartbeats). It then chooses a slot of time for starting emitting heartbeats at some a priori agreed rate following a TDMA-based protocol. Finally, it starts monitoring the other machines. A watchdog mechanism implemented at kernel level (the only processing cost added) allows the failure detector device to detect the failure of its own host. Whenever the device detects the failure of its host it stops sending heartbeats, therefore allowing the other devices to also detect this failure.

The global FD service implemented within the federation is composed by a number of federation detection modules. Each local domain of detection in the federation executes one

of these modules. They exchange information about failures on the nodes belonging to their respective local domains in order to implement the global FD service. This information is provided by the underlying local FD service (see Figure 2).

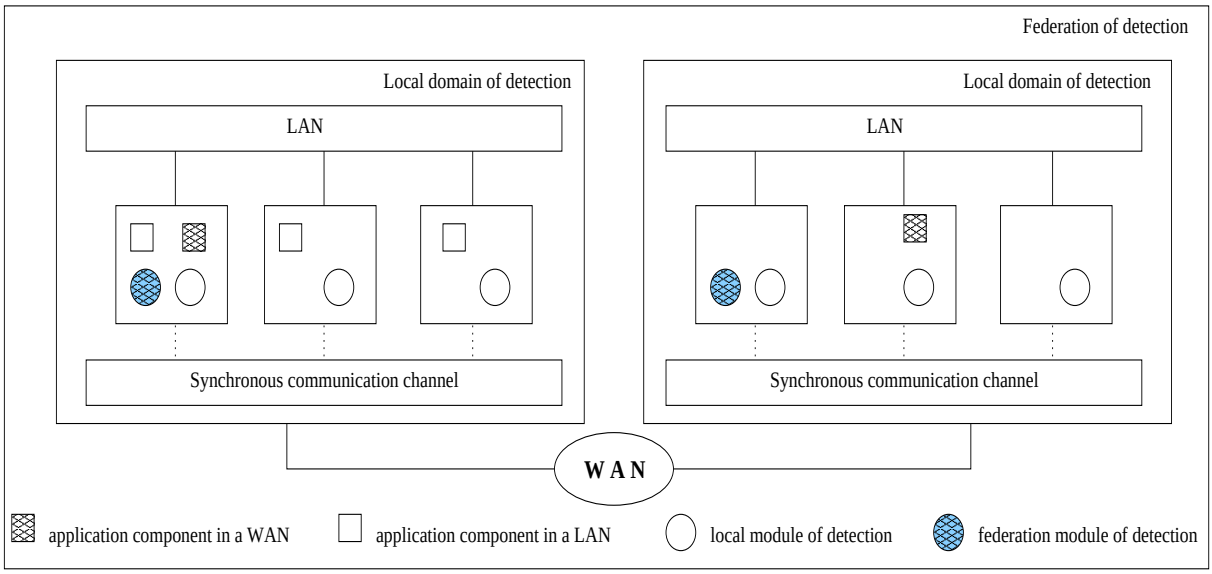


Figure 2: A Detailed View of the FD Service

It is important to notice that, although the transmission delay on the WAN is finite, (the WAN is non-partitionable) it is not bounded. Therefore, the federation detection modules executing on each local domain of detection must be fault-tolerant, otherwise it would be impossible to differentiate between a faulty module from one that is simply slow [FLP85]. The federation detection modules can be made fault-tolerant by having their implementation replicated within each local domain of detection. The required level of replication should be such that the probability of all replicated modules within a local domain fail is negligibly small.

Notice that when compared to traditional implementations of heartbeat based FD services, this two level hierarchy substantially reduces the number of messages required to be exchanged by failure detector modules, allowing for greater scalability. In fact, one can easily extend the two-level hierarchy with more levels to further increase scalability.

If it is not possible to assume a non-partitionable WAN, the global FD service cannot provide a perfect semantics. However, it can easily provide a weaker semantics, *e.g.* a $\diamond S$ failure detector [CT96]. In this case, applications spanning the federation would have to use protocols based on this weaker FD service. Nevertheless, those applications confined within a particular local domain of detection could still take advantage of a perfect FD service.

4 Concluding Remarks

We think that the result presented in [CHT96] has placed too much bias toward the implementation of $\diamond S$ failure detectors. Since these are the weaker failure detectors that allow

consensus to be solved, it is very much possible that they are the simplest and cheapest ones to implement, hence the interest they have raised recently. However, we are very confident that stronger failure detectors can indeed be built and at a very reasonable cost. Further, we believe that stronger failure detectors not only can reduce the assumptions that have to be made to guarantee the correctness of higher level protocols, but also yield substantially bigger performance gains if designers would be able to develop more efficient higher level protocols based on these stronger abstractions.

We are now modeling several consensus protocols based on a variety of failure detectors to gain insights on how to increase the performance of fault-tolerant applications. Our preliminary simulation analysis based on a Petri-net model supports our claims in favour of using stronger failure detectors. We are also implementing the FD service discussed in Section 3 to confront experimental results with the ones yield by our simulations.

References

[AV96] C. Almeida and P. Veríssimo. Timing failure detection and real-time group communication in quasi-synchronous systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time System*, L'Aquila, Italy, Jun 1996.

[CB97] V. S. Catão and F. V. Brasileiro. Serviço de comunicação síncrona para nodos replicados. In *Anais do VII Simpósio de Computadores Tolerantes a Falhas*, Paraíba, Brazil, Jul 1997.

[CHT96] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, Jul 1996.

[CPN93] Design/cpn: User's manual. CPN group, 1993.

[CT96] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar 1996.

[CTA00] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *DSN'2000*, Jun 2000.

[Fis83] M. J. Fischer. The consensus problem in unreliable distributed systems. Research Report 273, Yale University, Jun 1983.

[FLP85] M. J. Fischer, N. A. Lynch, and M. D. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, Apr 1985.

[SDS99] N. Sergent, X. Défago, and A. Schiper. Failure detectors: Implementation issues and impact on consensus performance. Technical Report SSC/1999/019, École Polytechnique Fédérale de Lausanne, Switzerland, May 1999.

[VCF00] P. Veríssimo, A. Casimiro, and C. Fetzer. The timely computing base: Timely actions in the presence of uncertain timeliness. In *DSN'2000*, Jun 2000.

Desenvolvimento de um Detector de Defeitos para Sistemas Distribuídos baseado em Redes Neurais Artificiais

Nivea Ferreira[1], Raimundo Macêdo
Laboratório de Sistemas Distribuídos - LaSiD
Universidade Federal da Bahia - UFBA
{niveacf, macedo}@ufba.br

Resumo

Detectores de defeitos são mecanismos importantes para a implementação de sistemas distribuídos tolerantes a falhas, i.e., sistemas que garantam serviços continuados mesmo na presença de falhas. Em sistemas distribuídos assíncronos, sem limites de tempo conhecidos para a transferência de mensagens e/ou computação de ações locais, detectores de defeito perfeitos (ou confiáveis) não podem ser implementados. Nesses ambientes, falhas podem apenas ser suspeitadas. Uma forma de diminuir a ocorrência de falsas suspeitas é a utilização de *timeouts* adaptativos que possam ser calculados em função, por exemplo, da carga da rede de comunicação. Em [2] discutimos o uso desses *timeouts* adaptativos para a implementação do módulo CTI (C*onnectivity Time Indicator*) que, por sua vez, foi utilizado para implementar detectores de defeitos não confiáveis do tipo <>S [1]. Em outro artigo [11], mostramos como o CTI pode ser usado para, além de detectar defeitos, controlar os níveis de qualidade de serviço (QoS) de comunicação entre processos distribuídos, no nível da aplicação. No presente artigo mostramos uma implementação do módulo CTI através de Redes Neurais Artificiais que interagem com agentes SNMP (*Simple Network Management Protocol*)[5] e MIB (*Management Information Base*)[6] para predizer tempos de conectividade baseados nas características operacionais dinâmicas de uma rede IP (*Internet Protocol*) como congestão, perda de pacotes, etc.

Abstract

Failure detectors are mechanisms used to implement fault tolerant distributed systems, i.e., systems that can provide continuous services even when failures may occur. In the so-called asynchronous distributed systems, without bounded and known time delay for message transfer and local processing, perfect - or reliable - failure detectors can not be implemented. In such systems, failures can only be suspected. In order to avoid false suspicions, adaptive timeouts can be calculated based on, for instance, the communication network load. In [2] we discussed the use of such adaptive timeouts to implement a mechanism called CTI (Connectivity Time Indicator) which in turn was used to implement a failure detector of the class <>S [1]. In another paper [11], we showed how the CTI could be used to control the levels of quality-of-service (QoS) of communication time between distributed processes at the application level. In this paper we show an implementation of the CTI mechanism through artificial neural network which interact with SNMP (Simple Network Management Protocol)[5] agents and MIB (Management Information Base)[6] in order to predict communication times based on the dynamic operational conditions of an IP (Internet Protocol) network.

[1]Bolsista CNPq/ProTeM-CC pelo projeto ARGO (processo número: 381520/2000-5).

1. Introdução

Prover tolerância a falhas é fundamental para que as aplicações em sistemas distribuídos, principalmente aquelas de segurança crítica, garantam os seus serviços mesmo na presença de falhas de alguns de seus componentes. Um módulo básico de sistemas tolerantes à falhas é o detector de defeitos, que informa sobre processos que falharam.

Em sistemas distribuídos assíncronos, onde não há limites no atraso de entrega de mensagens e tempos de execução, não é possível distinguir entre processos falhos daqueles muito lentos. Por outro lado, esse modelo de sistema é de especial interesse devido ao fato de ser mais realista para computações distribuídas atuais, como por exemplo, na Internet. Chandra e Toueg estenderam o modelo puramente assíncrono com a noção de detectores de defeitos [1]. Estes mecanismos de detecção de falhas podem cometer erros, e, por isso, são chamados de detectores não-confiáveis, e são como um oráculo do estado de funcionamento de processos. Nesses sistemas, *timeouts* não podem ser tomados como uma indicação precisa de falhas.

Valores de *timeout* muito altos diminuem a possibilidade de haver uma falsa suspeita de falha, caso um processo esteja muito lento, mas impede que falhas reais sejam detectadas mais rapidamente. Para resolver o impasse que recai sobre o usuário, que nos algoritmos de detecção de falhas tradicionais deve sugerir os valores de *timeouts* a serem utilizados, surgem os detectores de defeitos adaptativos. Com valores adaptáveis de *timeout* é possível detectar-se falhas de maneira mais precisa, já que teremos valores mais adequados às características atuais do canal de comunicação. *Timeouts* adaptativos sugerem a determinação de valores mais coerentes com a sobrecarga da rede.

Em [2] discutimos o uso de timeouts adaptativos para a implementação do mecanismo chamado CTI (*Connectivity Time Indicator* ou Indicador de Tempo de Conectividade), que visa indicar dinamicamente os valores de conectividade entre os processos monitorados. O CTI foi utilizado para implementar detectores de defeitos não confiáveis do tipo <>S. Em outro artigo [11], mostramos como o CTI pode ser usado para, além de detectar defeitos, controlar os níveis de qualidade de serviço (QoS) de comunicação entre processos distribuídos, no nível da aplicação. Nos trabalhos citados, o CTI foi implementado usando-se o último tempo de *round-trip* das mensagens. No presente artigo mostramos uma implementação do modulo CTI através de Redes Neurais Artificiais que interagem com agentes SNMP (*Simple Network Management Protocol*)[5] e MIB (*Management Information Base*)[6] para predizer tempos de conectividade baseados nas características operacionais dinâmicas de uma rede IP (*Internet Protocol*) como congestão, perda de pacotes, etc.

Redes neurais artificiais são poderosas ferramentas quando considera-se algumas classes de problemas complexos [3, 4, 13]. Em nossa proposta, os valores de variáveis da MIB são utilizados como entradas e os valores de tempo de conectividade associados caracterizam a saída da rede neural.

Este trabalho integra o projeto ARGO, que objetiva a concepção e desenvolvimento de serviços de comunicação em grupo, a partir de uma abordagem que permita limitar e controlar as disfunções típicas de sistemas distribuídos assíncronos. O ARGO tem, portanto, na detecção de falhas nestes sistemas uma das principais motivações[2].

[2] Para maiores informações, acesse: http://www.lasid.ufba.br/projetos/argo.

Na próxima seção apresentaremos o CTI. A seção 3 fala sobre redes neurais artificiais e na seção 4 será explicada a implementação do nosso detector de defeitos. A seção 5 é reservada à apresentação dos resultados obtidos. Os trabalhos relacionados serão apresentados na seção 6. A conclusão e os próximos passos desta pesquisa serão apresentados na seção 7.

2. Indicador de Tempo de Conectividade (CTI)

O tempo de conectividade, *ct*, entre dois processos P_i e P_j, é definido como o tempo que uma mensagem leva para trafegar de P_i a P_j (ou vice-versa) num dado momento. Em sistemas como a Internet, o tempo de conectividade pode assumir valores distintos, podendo variar de 0 (se i = j) até infinito (se P_i e P_j estão desconectados).

Como é impossível predizer precisamente o futuro, o CTI deve sugerir o tempo de conectividade atual. Assim, existirá um módulo CTI sendo executado em cada nodo do sistema distribuído e ele estará atualizando constantemente as informações sobre conectividade dos processos locais e remotos.

Os *timeouts* utilizados na detecção de defeitos são determinados pelos tempos de conectividade estabelecidos entre os processos.

Na próxima seção serão explicados os principais conceitos de redes neurais artificiais, que facilitarão o entendimento da implementação do módulo CTI, apresentada na seção 4.

3. Redes Neurais Artificiais

Genericamente, uma rede neural é composta por um número de unidades conectadas. Cada ligação tem um peso, um valor numérico, associado. Este pesos representam a capacidade de armazenamento/codificação de informação da rede neural, e o processo de aprendizado é associado à alteração desses pesos.

O modelo escolhido para a tarefa de implementação do CTI foi uma rede MLP (*Multilayer Perceptron*). Redes MLP são redes associativas formadas basicamente por:

- Uma camada que recebe os estímulos de entrada, sendo constituída, normalmente, por um número de neurônios correspondente à quantidade de atributos dos dados;
- Uma ou mais camadas intermediárias, cujas unidades recebem como dado de entrada os sinais provenientes de camada de entrada ou da camada intermediária anterior;
- Uma camada de saída que, recebendo as entradas da camada intermediária, chega a uma predição ou classificação.

Os pesos são modificados com base no erro verificado na saída da rede neural (aprendizado supervisionado). Mais detalhadamente: o estímulo de entrada é apresentado à rede e, então, propagado para as diversas camadas. Em cada unidade, as entradas são ponderadas pelos valores dos pesos, associados a cada uma das conexões, para determinar a ativação desta unidade. Com base nesta ativação, cada unidade transmite às unidades da camada seguinte, às quais está conectada, a sua saída. A saída produzida pela última camada é então comparada ao resultado esperado e o erro, a diferença entre o valor produzido e o esperado, é calculado. Esse erro é propagado de volta até alcançar as conexões que ligam a camada de entrada à primeira camada intermediária. Este aprendizado recebe o nome de *backpropagation*.

4. Implementação

Como dito anteriormente, a idéia principal do nosso processo de detecção de falhas em sistemas assíncronos considera que existirá um módulo CTI sendo executado em cada nodo, atualizando as informações sobre a conectividade dos processos distribuídos. Assim, a proposta deste trabalho é a utilização de redes neurais para implementar o módulo CTI, e realizar a tarefa de determinar *timeouts* que sejam mais adequados, de acordo com as características atuais do canal de comunicação.

O SNMP é o protocolo utilizado para fazer a interface entre os objetos gerenciáveis da rede de comunicação (descritos na MIB) e a rede neural. Utilizando o SNMP, os objetos são lidos a partir de:

- Grupo de Interface, que representa a interface de rede
- Grupo IP, que representa o *Internet Protocol* , e
- Grupo UDP, que representa o *User Datagram Protocol*.

Estes diferentes grupos da MIB representam diferentes tipos de funcionalidade.

A rede neural MLP com aprendizado *backpropagation* foi implementada utilizando-se a linguagem Java. Como padrões de entrada desta rede neural temos os valores de variáveis da MIB, através dos quais podemos caracterizar o tráfego da rede de comunicação. Desta forma, o tempo de conectividade associado a cada padrão de entrada é a saída dada pela rede a cada instante.

Os dados utilizados para treinamento e teste da rede neural implementada foram coletados nos módulos do Laboratório de Sistemas Distribuídos, onde esta pesquisa está sendo desenvolvida. Para isso, foram utilizadas 4 máquinas.

5. Resultados

A nossa base de dados inicial é composta por 4697 padrões de entrada da rede[3], divididos em 6 arquivos de tamanhos variáveis (i.e., possuindo número diferente de padrões cada um). Foram feitas várias etapas de treinamento e teste, utilizando-se os dados destes arquivos.

O treinamento da rede neural é feito utilizando-se um dos arquivos, sendo o número de ciclos de treinamento determinado por um erro mínimo especificado. Tendo encerrado o treinamento, o desempenho da rede é verificado utilizando-se um outro arquivo de dados, diferente daquele utilizado para o treinamento. Ou seja, na fase de testes são apresentados à rede neural padrões desconhecidos. A avaliação da precisão dos valores sugeridos como saída da rede neural são comparados com os valores reais, registrados durante a coleta dos dados.

A Figura 1 mostra o tempo de conectividade associado a cada padrão registrado. Estes padrões são referentes a um arquivo de teste. Na Figura 2 são mostradas as saídas da rede neural usando os padrões registrados no arquivo de teste como padrões de entrada. Comparando-se os gráficos, os valores sugeridos pela rede se aproximam e muito dos valores reais observados.

[3]Que são os valores coletados nos objetos da MIB.

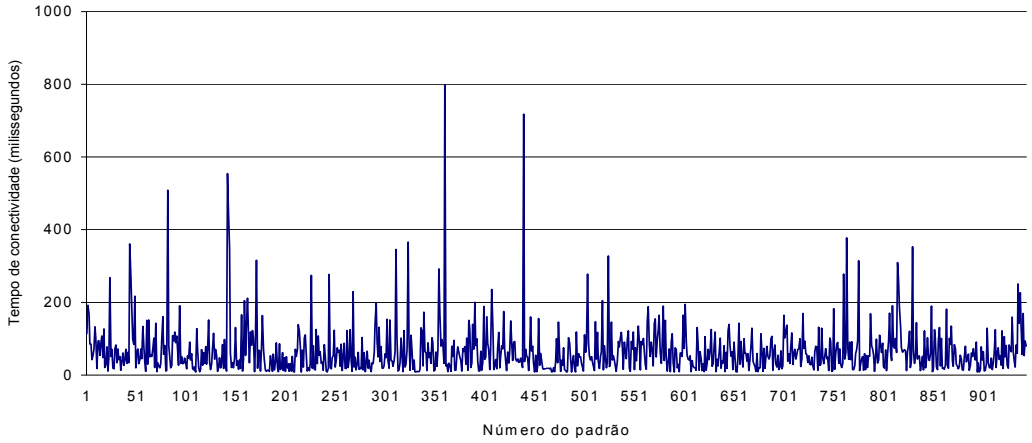


Figura 1: Tempo de conectividade real relacionado a cada padrão.

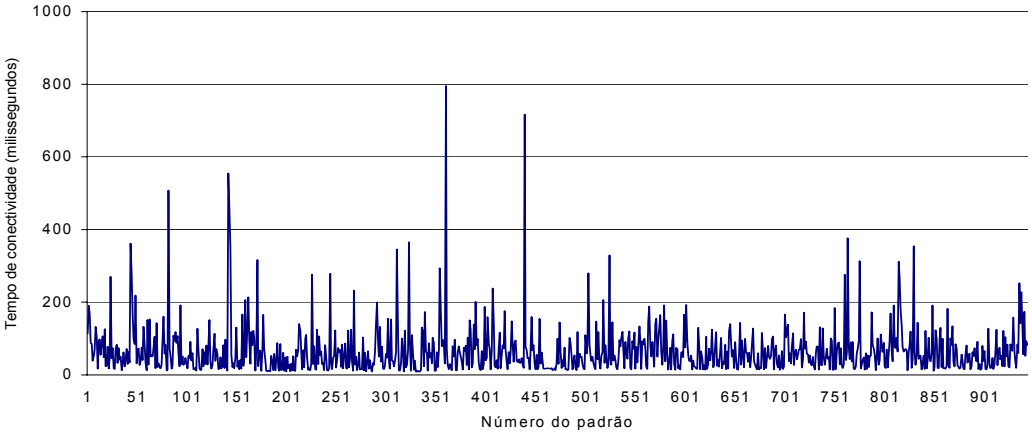


Figura 2: Tempo de conectividade sugerido pela rede neural relacionado a cada padrão.

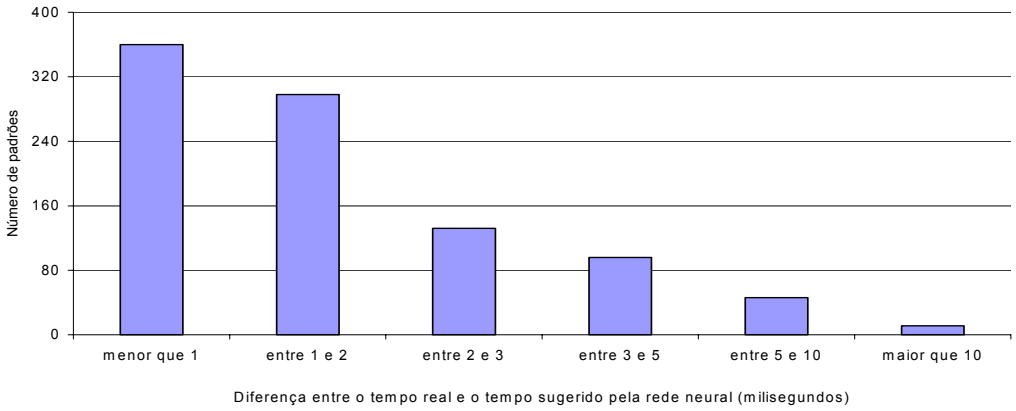


Figura 3: Classificação dos padrões quanto à diferença entre o tempo real e o tempo sugerido pela rede neural.

A Figura 3 apresenta a diferença dos valores de tempo de conectividade sugeridos pela rede neural e os valores reais, apresentados nos gráficos anteriores. Considerando os dados

descritos neste gráfico, 99% desta diferença está entre 0 e 10 milissegundos, o que consideramos como um desempenho satisfatório.

6. Trabalhos Relacionados

O trabalho de Hood e Ji [10] utiliza a abordagem de redes bayesianas para monitoração de falhas de sistemas. Essa abordagem está baseada na implementação de agentes SNMP inteligentes, que aprendem o comportamento anormal da rede e combina isto com a modelo probabilístico da rede bayesiana. No nosso trabalho, visamos detectar defeitos de processos, definidas com base no uso de *timeouts*, trazendo uma nova concepção de cálculo destes valores de modo a torná-los adaptativos. O uso de variáveis da MIB, neste caso, auxilia-nos na identificação de mudanças no sistema que caracterizariam as suspeitas de defeitos.

Uma outra abordagem é apresentado em [9], que descreve um algoritmo, chamado DPCP, para cálculo de *timeouts* e intervalos de monitoração adaptativos. Este algoritmo tem como idéia principal o uso do tempo atual, descartando-se os anteriores, para calcular estes valores. Isto difere da nossa abordagem uma vez que não considera as características de sobrecarga da rede para este cálculo e pelo fato de não considerar valores anteriores de tempo de conectividade.

Em [12] foi descrito um conjunto de métricas para especificar detectores de defeitos para sistemas com comportamentos probabilísticos e apresentado o algoritmo que será analisado de acordo com estas métricas. O algoritmo adaptativo deve ser capaz de se reconfigurar de acordo com as mudanças nas características probabilísticas de rede. Os detectores de defeito, neste caso, não são definidos como nos sistemas assíncronos, pois as aplicações com restrições de tempo requerem detectores que possam prover qualidade de serviço com garantias quantitativas de tempo. Ou seja, neste artigo são estudados os sistemas onde a perda e o atrasos das mensagens seguem distribuições probabilísticas.

O trabalho apresentado em [13] é uma proposta de uso de redes neurais para um controle adaptativo de qualidade de serviço (QoS) para redes ATM. Para garantir parâmetros de qualidade de serviço como tempo de atraso de células ou probabilidade de perda de células, é necessário gerenciamento de tráfego e controle de congestão. A idéia neste artigo é utilizar redes neurais para estimar QoS para o controle de admissão de chamadas. Para tal, a rede neural usa a combinação do número de conexões usando áudio, vídeo e serviços de comunicação de dados, sendo o número de entradas da rede determinada principalmente pelo número de parâmetros de tráfego de cada umas destas categorias. Neste artigo são mostrados os resultados obtidos por uma rede MLP.

Nos sistemas de tempo-real com altas restrições de recurso, o mecanismo de tolerância deve se adequar aos recursos disponíveis e às condições do ambiente. O objetivo do trabalho apresentado em [14] é a operação autônoma de funções críticas de espaçonaves, isto é, executá-las de maneira autônoma sem que haja perda significativa de performance ou de segurança, mesmo na presença de eventos indesejáveis. Mais detalhadamente, o objetivo do mecanismo de adaptação é minimizar a utilização de recursos, mantendo os requisitos de confiabilidade; deve ser genérico o suficiente para lidar com vários tipos de falhas (de sensor, de processador, de aplicativos); responder às mudanças de ambiente, missão, fases, estados do sistema e perfis de usuário, que provê os parâmetros da aplicação que afetam a tomada de decisão sobre a adaptação. Para isso, o mecanismo usa informações sobre os recursos

disponíveis, a demanda do ambiente e um histórico de falhas recentes, como indicativo da probabilidade de falhas futuras.

7. Conclusões

O uso t*imeouts* é comum em detecção de falhas. Valores de *timeouts* constantes comprometem a eficiência dos detectores de defeitos, uma vez que não considera as variações na sobrecarga da rede, que interferem diretamente nos tempos de resposta dos processos ativos (isto é, não-falhos). Valores baixos são responsáveis por suspeitas incorretas de processos corretos como sendo falhos, enquanto valores altos adiam a suspeita de ocorrência real de falhas de processos.

A partir da necessidade de definição de valores mais coerentes com as características correntes da rede e da comunicação entre processos, surgem os *timeouts* adaptativos. Tendo valores que estão de acordo com essas características, busca-se minimizar a ocorrência de erros na detecção de falhas em sistemas distribuídos assíncronos, dado que nestes sistemas, devido às suas características, valores de *timeouts* são aproximativos.

O uso de redes neurais para a implementação do mecanismo do CTI com características adaptativas apresentou ótimos resultados. O nível de acerto dos tempos de conectividade sugeridos pela rede neural se aproximaram, com grande precisão, dos tempos de conectividade registrados.

Com base nestes resultados, estamos iniciando uma nova fase que se caracteriza pela integração deste módulo de detecção de falhas aos outros módulos que compõem a plataforma ARGO de tolerância a falhas.

Além dessa integração, considerando a característica estática da rede neural explicada anteriormente, decidimos verificar o comportamento de uma rede que possuísse características dinâmicas e fosse adequada, portanto, para predição de séries aleatórias [8]. Desta forma, seria possível realizar a tarefa de predição de valores de conectividade, ou seja, mais do que valores atuais, seríamos capazes de determinar valores em instantes futuros. Neste caso, é necessário estender a arquitetura de redes neurais descrita anteriormente.

Novos testes estão sendo realizados para atestar a capacidade preditiva deste tipo de topologia de rede neural. Os resultados já obtidos mostraram-se bastante interessantes, indicando ser esta um linha de pesquisa com um chance concreta na obtenção de importantes resultados.

8. Agradecimentos

Ao CNPq, que através do ProTeM-CC, viabilizou o desenvolvimento desta pesquisa.
À Fábio Lima, aluno da Graduação do Curso de Ciência da Computação, que realizou a tarefa de implementação dos agentes SNMP.

Referências

[1] Chandra T., Hadzilacos V. and Toueg S., *The weakest Failure Detector for Solving Consensus*. Journal of the ACM, 43(4): 685-722, July 1996.

[2] Macêdo R., *Failure Detection in Asynchronous Distributed Systems*. Proc. of II Workshop on Tests and Fault-Tolerance, pp. 76-81, July 2000, Curitiba, Brazil.

[3] Rietman, E.A. and Frye, R.C. Neural Control of a Nonlinear System with Inherent Time Delays. Conference on Analysis of Neural Network Applications, pp.140-145, 1991.

[4] Chow, M. and Yee, S.O. Real Time Application of Artificial Neural Networks for Incipient Fault Detection of Induction Machines. Proceedings of the 3rd International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 90), pp. 1030-36, Charleston, USA, July 1990.

[5] Case, J., Fedor, M., Schoffstall, M., Davin, J. Connected: An Internet Encyclopedia – A Simple Network Management Protocol at www address http://deese.univ-lemans.fr:8003/connected/RFC/1157/index.html

[6] McCloghrie, K., Rose, M. Connected: An Internet Encyclopedia – Management Information Base for Network Management of TCP/IP-based internets: MIB-II at www address http://deese.univ-lemans.fr:8003/connected/RFC/1213/index.html

[7] Russell, S. and Norvig, P. *Artificial Intelligence- A Modern Approach*. 1st ed. New Jersey, Prentice-Hall, 1995.

[8] Haykin, S. *Neural Networks – A Comprehensive Foundation*. 1st ed. New York, Macmillan, 1994.

[9] Sotoma, I. and Madeira, E.R.M. DPCP (Discard Past Consider Present) – A Novel Approach to Adaptive Fault Detection in Distributed Systems. 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'2001), vol.1, pp.76-82, Bologna, Italy, October 2001 .

[10] Hood, C. S. and Ji, C. Intelligent Agents for Proactive Fault Detection. Internet Computing, v.2, n.2, pp.65-72, March/April, 1998.

[11] Batalha, M. and Macêdo R.J.A.Um Serviço Tolerante a Falhas para o Gerenciamento de Sistemas Distribuídos Sobre CORBA. Proceedings of the Latin-American Conference on Informatics (CLEI'2001). Mérida, Venezuela. September/2001.

[12] Chen W., Toueg, S. and Aguilera, M.K. On the Quality of Service of Failure Detectors. Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000), New York, June 2000.

[13] Hiramatsu, A. Training Techinques for Neural Network Applications in ATM. IEEE Communications Magazine, pp.58-67, October 1995.

[14] Shokri, E. and Beltas, P. An Experiment with Adaptive Fault Tolerance in Highly-Constraint Systems. Proceedings of the Fifth International Workshop on Object-Oriented Real-Time Dependable Systems, California, USA, November 1999.

Uma Implementação do Detector de Falhas do FT-CORBA

Luelson Marlos Nunes
Elias Procópio Duarte Jr.

Universidade Federal do Paraná
Departamento De Informática
Caixa Postal 19081 Curitiba 81531-990 PR Brasil
E-Mail: {lnunes,elias}@inf.ufpr.br

Resumo

Neste trabalho apresentamos uma abordagem de implementação do detector de falhas do FT-CORBA (*Fault Tolerant-Common Object Request Broker Architecture*), utilizando a linguagem de programação Java integrada e o serviço COSNaming de uma implementação aberta do padrão CORBA. A abordagem utilizada neste trabalho para a implementação do detector de falhas propõe a monitoração periódica exclusivamente da réplica primária dos servidores. A monitoração das demais réplicas é executada apenas em caso de falha da réplica primária. Neste caso um novo servidor primário é eleito entre as réplicas sem falhas. Também é apresentado um exemplo de construção de uma aplicação tolerante a falhas utilizando o detector implementado.

Palavras-Chave: Sistemas Distribuídos, Tolerância a Falhas, Replicação, CORBA, Detecção de Falhas.

1 O Detector de Falhas do FT-CORBA

O padrão CORBA define um dos modelos de computação para objetos distribuídos mais amplamente utilizados [1]. O CORBA é a especificação de uma arquitetura e interface que permite que aplicações efetuem requisições a objetos de forma transparente e independente da linguagem de programação, sistema operacional ou mesmo da localização dos objetos[4].

O padrão para tolerância a falhas FT-CORBA [2] tem por objetivo fornecer um suporte para aplicações que necessitam de garantias de disponibilidade. O foco deste trabalho é a implementação de um dos componentes deste padrão: o Detector de Falhas. Também foi implementada uma versão simplificada do Gerenciador de Replicação.

O Detector de Falhas é responsável pela percepção da presença de falhas no sistema e geração de um relatório de falhas para o Notificador de Falhas. O Notificador de Falhas efetua uma seleção sobre este relatório visando eliminar informações desnecessárias ou duplicadas enviando então um evento de notificação para os consumidores deste serviço, como o próprio Gerenciador de Replicação [2].

Um exemplo de interação cliente/servidor tolerante a falhas baseada no FT-CORBA é ilustrada na figura 1. Um servidor de hora é utilizado no exemplo. O componente *Client* representa um cliente que deseja utilizar o servidor. O componente *COS-Naming* representa o serviço de nomes do CORBA [3] onde os grupos e os objetos replicados são registrados. O componente *ReplicationManager*, neste caso, representa o objeto responsável por iniciar o processo de recuperação da referência da réplica do servidor primário do grupo de objetos solicitado pelo cliente. O componente *PrimaryTimeServer* representa o servidor primário do grupo de objetos. O componente *FaultDetectorImpl* representa a classe responsável pela monitoração dos servidores primários dos grupos de objetos. O componente monitor representa a *thread* iniciada para monitoração de cada servidor primário dos grupos de objetos existentes.

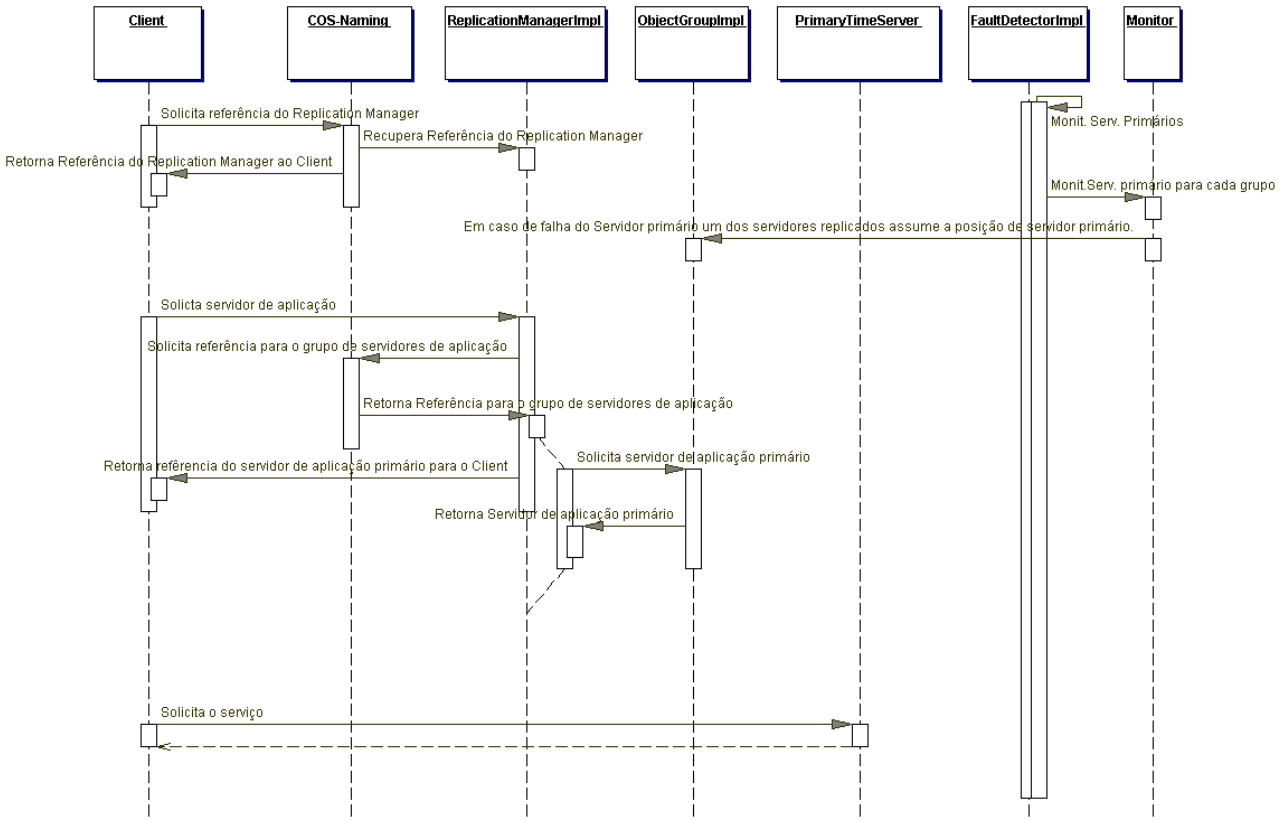


Figura 1: Interação cliente/servidor tolerante a falhas.

O processo apresentado na figura 1 ilustra a requisição do *Client* ao *COS-Naming* para obtenção da referência ao *ReplicationManager*. A partir desta recuperação o cliente solicita ao *ReplicationManager* a referência do servidor de hora pelo nome do grupo de servidores. O *ReplicationManager* por sua vez efetua uma consulta ao *COS-Naming* buscando a referência para o grupo de objetos desejado, que é representado na figura 1 pelo componente *ObjectGroupImpl*. Ao obter a referência para o grupo de objetos, este retorna a referência para o servidor primário do grupo desejado que é então retornado para o *ReplicationManager* e, por sua vez, retornado para o *Client* que pode então efetuar a solicitação desejada.

O servidor primário de cada grupo é periodicamente monitorado através do componente representado na figura 1 por *FaultDetectorImpl*. O intervalo de monitoração é configurável. O componente *FaultDetectorImpl* cria uma *thread* para cada grupo de objetos, estas *threads* são representadas na figura 1 pelo componente *Monitor*.

O processo de monitoração consiste na obtenção da referência apenas do servidor primário de cada grupo, checando se o mesmo está funcional. A monitoração não é feita para todas as réplicas, o que oferece um ganho potencial de performance se considerarmos um grande número de réplicas. Por outro lado, caso múltiplas réplicas falhem, o intervalo de tempo até que um novo servidor seja eleito pode ser consideravelmente elevado. No processo de monitoração implementado, somente quando o servidor primário não está ativo, a monitoração das réplicas é efetuada elegendo-se então um novo servidor primário para o grupo de objetos em questão e configurando esta nova réplica como servidor primário deste grupo conforme representado na figura 1 pela interação entre o componente *Monitor* e o componente *ObjectGroupImpl*.

2 Integração de uma Aplicação com o Detector de Falhas

A integração ao Detector de Falhas de uma aplicação exemplo que consiste em um sistema cliente-servidor de hora é descrita a seguir. Para que o servidor de hora *TimeServerImpl*, representado na figura 2 no canto superior direito, possa ser monitorado é necessário que ele implemente duas interfaces. A primeira interface é chamada de *OrbInterface* e define o método *initOrb*. A partir desta implementação é possível disponibilizar o servidor via ORB. A segunda interface é chamada de *PullMonitorableOperations* e define a implementação do método *isAlive()*. Este método é utilizado teste do objeto replicado e é invocado periodicamente pelo monitor do detector de falhas para monitorar se o servidor primário do grupo de objetos está falho ou sem-falhas.

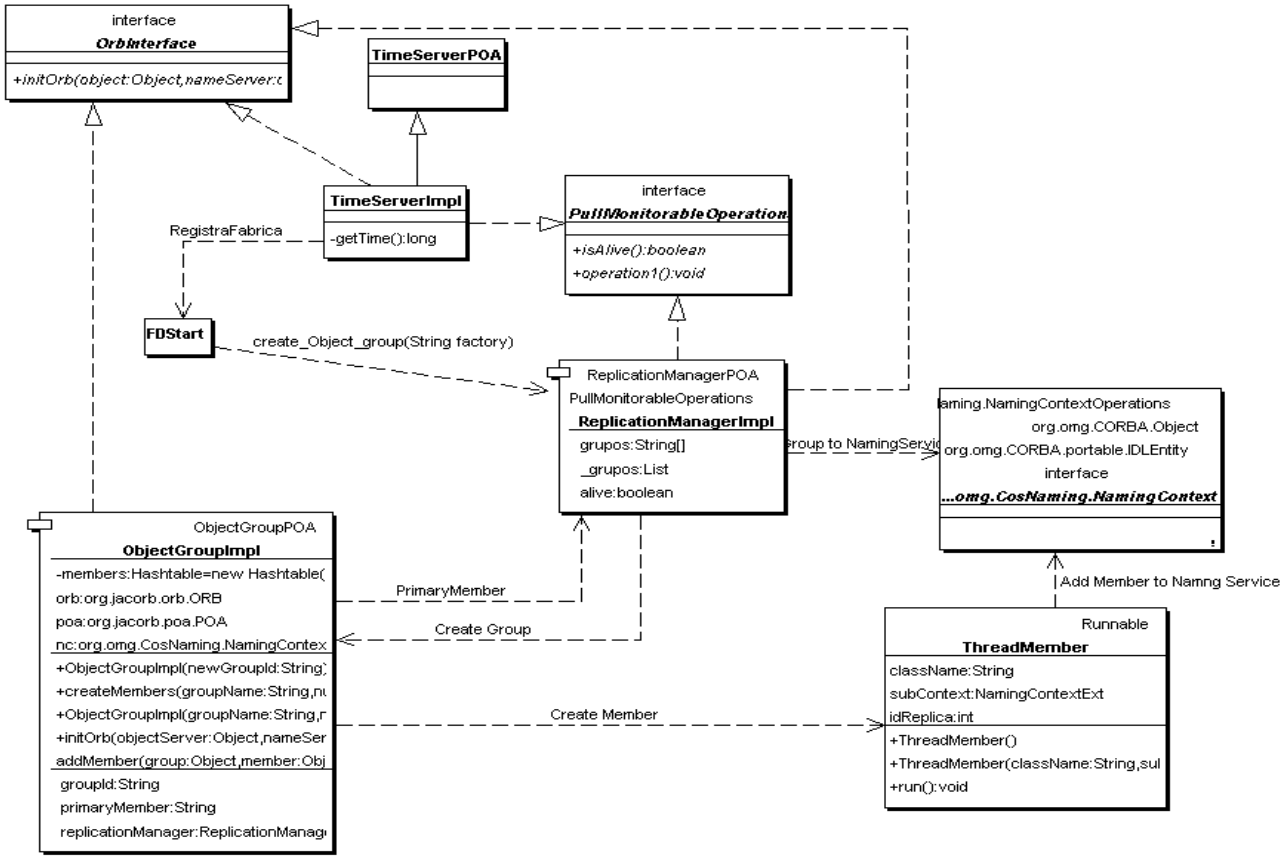


Figura 2: Integração de uma aplicação ao Detector de Falhas.

Qualquer aplicação servidora monitorada pelo detector de falhas irá utilizar os componentes *FDStart* e *ObjectGroupImpl*. A partir do componente *FDStart* a aplicação servidora faz o registro do seu nome. A partir deste nome é possível que seja efetuada a instanciação dinâmica das réplicas do servidor pelo componente *ObjectGroupImpl* que é o responsável pela criação e disponibilização das réplicas dos grupos de objetos a serem monitorados.

A figura 2 apresenta a integração de uma aplicação com a arquitetura de detecção de falhas. O inicio do processo de criação de grupos de objetos replicados é feito pelo *FDStart* que aciona o *ReplicationManagerImpl*, o qual registra o novo grupo no serviço de nomes CORBA e invoca o *ObjectGroupImpl* que por sua vez inicia o processo de disponibilização dos servidores replicados.

O componente responsável pela criação das réplicas dos grupos a serem monitorados é representado na figura 2 pelo componente *ObjectGroupImpl*. Este componente realiza a intanciação dinâmica das réplicas do servidor e disponibiliza os servidores em diferentes *threads* representadas na figura 2 pelo componente *ThreadMember*. O número de réplicas é configurável por host e a existência de réplicas em diferentes hosts é possível.

3 Considerações Finais

A abordagem utilizada neste trabalho para a implementação do Detector de Falhas propõe a monitoração periódica de atividade apenas do servidor primário, iniciando o processo de monitoração para as demais réplicas somente em caso de falha deste servidor. Neste caso o servidor primário é substituído por um novo servidor replicado ativo, isto aumenta o desempenho do sistema pois elimina parte do *overhead* de monitoração de múltiplas réplicas. Por outro lado, caso múltiplas réplicas falhem, o intervalo de tempo até que um novo servidor seja eleito pode ser consideravelmente elevado.

Uma avaliação quantitativa da utilização da estratégia proposta está prevista como trabalho futuro. Trabalhos futuros incluem também, além da implementação do Notificador de Falhas, a implementação da Fábrica Genérica de objetos replicados.

Referências Bibliográficas

[1] A. Pope, *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture*, Addison-Wesley, 1997.

[2] Object Management Group, “Fault Tolerant CORBA Specification V1.0,” *OMG Document ptc/2000-04-04 ed.*, 2000. Disponível em URL: ftp://ftp.omg.org/pub/docs/ptc/00-04-04.pdf

[3] Object Management Group - OMG Home. Disponível em URL: http://www.omg.org/

[4] Object Management Group, “A discussion of the Object Management Achitecture, ” *OMG Document 00-06-41 ed.*, 1997. Disponível em URL: ftp://ftp.omg.org/pub/docs/formal/00-06-41.pdf

Preenchendo o Vazio entre Comunicação em Grupo e Multicast Escalável

Maglan Cristiano Diemer Marinho Pilla Barcellos

PIPCA- Programa de Pós-Graduação em Computação Aplicada
C6 - Centro de Ciências Exatas e Tecnológicas
UNISINOS - Universidade do Vale do Rio dos Sinos
Av. Unisinos, 950 - São Leopoldo, RS - CEP 93.022-000 - BRASIL
{maglan,marinho}@exatas.unisinos.br

Resumo

Certos sistemas de comunicação em grupo necessitam de um protocolo de transporte multicast subjacente para operarem de forma eficiente sobre a Internet. Por outro lado, protocolos multicast não oferecem a confiablidade e as garantias desejadas para aplicações distribuídas. Em geral, os sistemas de comunicação em grupo oferecem confiabilidade mas não são escaláveis. Este trabalho tem o objetivo de aproximar os protocolos multicast escaláveis dos sistemas de comunicação em grupo propondo uma arquitetura para trabalharem juntos. Será utilizado o sistema de comunicação em grupo NewTOP ([11]) e o protocolo multicast PRMP ([4]).
Palavras-chaves: comunicação em grupo, protocolo multicast, tolerância a falhas.

1 Introdução

Desde a década de 80 diversos grupos de pesquisa estudam protocolos e *sistemas de comunicação em grupo* e seu emprego em aplicações distribuídas tolerantes a falhas. Muitas contribuições resultaram (e continuam a resultar) de tal esforço, como por exemplo [1], [23], [15], [13] e [2], apenas para citar alguns. Os sistemas de comunicação em grupo fornecem o suporte e a confiabilidade necessários para que aplicações troquem informações de maneira consistente entre membros de grupos, apesar da ocorrência de falhas. Para que esses protocolos e sistemas operem de maneira eficiente na Internet, muitos são os desafios a serem vencidos. Exemplos de esforços nesse sentido são [6], [22] e [14].

Dada a tradicional filosofia de projeto de separar a complexidade em camadas, é desejável a utilização de um *protocolo de transporte multicast* subjacente, que ofereça serviços básicos como controle de erro (detecção de perdas e retransmissão) e congestionamento (convivência com outros fluxos na Internet), entre outros ([3]). Mas estes protocolos de transporte multicast enfrentam problemas de escalabilidade, em particular se confirmações de recebimento (ACKs) são necessárias (múltiplos fluxos TCP, a maneira mais simples e popular, é pior ainda). Antes que comunicação em grupo com tolerância a falhas possa ser empregada com sucesso em larga escala na Internet, sistemas de comunicação em

grupo devem ser combinados com protocolos de transporte multicast e cuidadosamente avaliados em configurações representativas da Internet.

Neste trabalho, descreve-se uma iniciativa cujo objetivo é combinar em uma arquitetura um protocolo escalável para transporte multicast (PRMP [3, 4]) com um sistema de comunicação em grupo que oferece ordenamento total e um sistema de controle de composição de grupo (NEWTOP [16, 18, 17, 11]). Com essa arquitetura, pretende-se implementar uma versão do NEWTOP com PRMP, baseando-se na versão atual do protótipo do NEWTOP ([16]). Tal implementação será utilizada para executar simulações em configurações de rede de larga escala (em número de nós e latências), avaliando-se seu comportamento e escalabilidade. Adicionalmente, a arquitetura será avaliada em cenários com falha de colapso, valendo-se de uma extensão de um conhecido simulador de redes.

O restante deste artigo está organizado da seguinte maneira. A Seção 2 aprofunda e diferencia os conceitos de *comunicação em grupo* e *transporte multicast escalável*. A Seção 3 descreve os princípios básicos e a implementação do NEWTOP, enquanto a Seção 4 discute os controles e o funcionamento do protocolo PRMP. A Seção 5 define uma proposta de arquitetura para a utilização do PRMP como suporte, na camada de transporte, para o sistema de comunicação em grupo NEWTOP. Por fim, a Seção 6 fecha o artigo com as considerações finais e trabalhos futuros.

2 Comunicação em Grupo x Transporte Multicast Escalável

Como acima citado, os sistemas de comunicação em grupo necessitam de uma camada de transporte multicast com garantias mínimas de confiabilidade, para que possam operar eficientemente em aplicações na escala da Internet. O termo *sistema de comunicação em grupo* é usado para representar os protocolos multicast *confiáveis* que gerenciam a comunicação entre os seus membros, possibilitando a troca de mensagens segundo o modelo vários-para-vários. O termo *protocolo de transporte multicast escalável* corresponde ao protocolo de transporte que oferece garantias mínimas de confiabilidade, como por exemplo confirmação positiva de recebimento (ACK). O protocolo de transporte será responsável pelo envio eficiente e escalável de mensagens segundo o modelo um-para-vários, operando sobre a arquitetura de IP multicast. Baixando ainda mais um nível (rede), IP multicast é formado por duas partes bem distintas: protocolos de roteamento multicast inter-redes (como MOSPF ou PIM-SM), e o protocolo de "gerência de grupo" intra-redes (IGMP). Ambos componentes não oferecem qualquer garantia de confiabilidade: operações podem falhar silenciosamente ou atrasar por períodos arbitrários. Os próximos parágrafos esclarecem os serviços prestados pelos *sistemas de comunicação em grupo* e dos *protocolos de transporte multicast*, respectivamente.

Em geral, *sistemas de comunicação em grupo* podem oferecer garantias de entrega de mensagens tais como atomicidade e ordenamento, além de controle consistente de composição de grupo. A atomicidade garante que uma mensagem, uma vez entregue a um processo (*membro*) de um determinado grupo, deve também ser entregue a todos os outros processos em funcionamento do mesmo grupo, mesmo que o processo que originou a mensagem falhe antes de finalizar a transmissão ([8, 19]). O ordenamento das mensagens é necessário para se obter o comportamento correto dos membros dos grupos, sincronizando

a ordem em que as ações são executadas. Há dois tipos comuns de ordenamento utilizados pelos sistemas de comunicação em grupo: ordenamento causal e ordenamento total ([12, 9]). O controle de grupo corresponde ao gerenciamento dos membros de um grupo: criação e destruição de grupos, inserção e remoção de membros. O controle de grupo também é responsável por manter atualizada a informação referente aos membros que compõem um grupo ([7, 10, 17]), atualizando-a consistentemente em caso de falhas. Estas facilidades são importantes para auxiliar a construção de sistemas distribuídos complexos, em particular com requisitos de *dependabilidade.* Entretanto, protocolos e algoritmos para ordenamento e controle de membros existentes nos *sistemas de comunicação em grupo* foram projetados para redes locais, e não são escaláveis ([24]).

Por outro lado, os *protocolos de transporte multicast* não possuem os serviços básicos de comunicação em grupo ([3]). Os *protocolos de transporte multicast* são baseados em um serviço UDP de "melhor esforço", ou seja, tentam entregar a mensagem ao seu destino sem oferecer absolutamente nenhuma garantia. Existem muitos exemplos de protocolos de transporte multicast, estando os mesmos divididos nas classes *orientado-a-remetente* e *orientado-a-receptor* ([25]). Protocolos orientado-a-remetente são baseados em ACKs (confirmações positivas), o qual leva ao problema da implosão (*feedback implosion*). Protocolos orientado-a-receptor, em contraste, visam escalabilidade em detrimento da confiabilidade ou desempenho. Baseados em NACKs (confirmações negativas), estes protocolos passam a **responsabilidade de detectar e recuperar perdas para o receptor**, de forma que o remetente **não mantenha quaisquer informações sobre os receptores**. Por essa razão, esta classe de protocolos não oferece garantias de confiabilidade "fim-a-fim" desejáveis a uma camada subjacente de suporte à comunicação em grupo.

O *protocolo de transporte multicast* baseado em ACKs pode oferecer as garantias desejadas pelo *sistema de comunicação em grupo*, mas, como já mencionado, possui o problema da implosão. Uma alternativa intermediária são os protocolos baseados em *polling* (veja Seção 4).

3 NewTOP

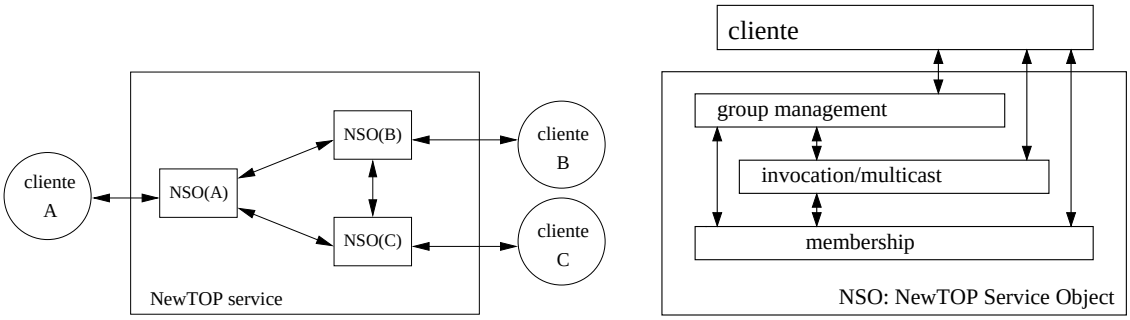
O NewTOP é um sistema de comunicação em grupo que possui os protocolos de ordenamento e controle de grupo garantindo a atomicidade na troca de mensagens. Ele assume que os membros podem estar presente em vários grupos simultaneamente e o tamanho do grupo não é limitado. Define-se que o ambiente de execução é assíncrono, onde o tempo de transmissão da mensagem não pode ser estimado com precisão. A camada de rede pode sofrer um particionamento sendo que a funcionalidade da comunicação entre os membros é preservada ([16, 11]).

O NewTOP implementa ordenamento causal e duas versões de ordenamento total: assimétrico e simétrico. No assimétrico, um único membro do grupo é responsável por determinar a ordem de entrega, enquanto no simétrico todos os membros do grupo compartilham a responsabilidade por determinar a ordem.

O controle de membros é implementado através de *groups-views.* Cada membro possui uma visão do grupo que é atualizada sempre que se detecte/suspeite a sua alteração, normalmente originada pela falha de um membro. Além disso, há garantia que todos os membros possuirão uma visão consistente do grupo, que é regida pelas propriedades

identificadas por VC - *view consistency*. Assim como as VCs, existem as propriedades chamadas de MD - *message delivery*, que garantem a atomicidade na troca de mensagens entre os membros ([11]).

Atualmente, o NEWTOP está sendo implementado via serviço CORBA ([17, 18]). Os membros dos grupos são objetos CORBA. Os clientes do serviço podem gerenciar a criação e a exclusão dos membros. Os objetos podem participar em mais de um grupo simultaneamente, permitindo também que os membros dos grupos se sobreponham. O NEWTOP é um serviço distribuído e auxiliado pelos NSOs (*NewTOP Service Object*) (ver Figura 1). Para cada cliente existe a alocação de um NSO que controla a comunicação com o grupo. A comunicação entre os NSOs é realizada pela camada ORB. A implementação dos controles oferecidos pelo NEWTOP é realizada pelos NSOs.



(a) Clientes NewTOP e seus NSOs

(b) Serviços implementados pelo NSO

Figura 1: Implementação do NEWTOP.

4 PRMP

O PRMP é um protocolo de transporte multicast baseado em *polling* que possui um mecanismo eficiente para o controle das implosões. O protocolo estende o modelo de comunicação TCP, conhecido pela maioria dos desenvolvedores de aplicações de rede, para um esquema com múltiplos destinatários. Notadamente, um protocolo de transporte multicast como o PRMP implementa controle de congestionamento multicast, de forma a permitir o compartilhamento justo dos fluxos TCP e IP multicast ([5]). A seguir, será fornecida uma breve explicação sobre o funcionamento do PRMP e os aspectos relevantes de sua implementação ([4, 5]).

Os dados são colocados em pacotes e transmitidos através de IP multicast para os receptores. O transmissor e os receptores mantém uma "janela deslizante"; o transmissor marca na sua janela quais os pacotes que foram recebidos (ACK) pelos receptores, de acordo com as respostas. Para evitar a implosão de respostas, o transmissor utiliza uma política de *polling* para controlar a quantidade de respostas geradas pelos receptores: somente quando solicitado, por uma requisição de *polling*, o receptor pode enviar, por unicast, uma resposta contendo a informação sobre ACKs e NACKs dos pacotes recebidos até o momento. Quando a resposta chega no transmissor, a sua janela é atualizada, e a detecção de perda

e recuperação podem serem executadas. O transmissor detecta a perda de pacotes através dos NACKs contidos na resposta enviada pelos receptores, após a requisição de *poll*. A recuperação é realizada através de retransmissão, que pode ser realizada por múltiplos *unicast* ou por uma única operação *multicast,* dependendo o número de cópias (do pacote) que deve ser retransmitido. As requisições de *poll* e as respostas também podem serem perdidas, e ambas as perdas são detectadas pelo transmissor através de *timeouts*. Se, o transmissor re-envia a requisição de *poll* um novo *timer* é definido para esperar por respostas; o processo é repetido até que uma resposta seja recebida ou que o transmissor retire o receptor desta sessão.

A janela do receptor é "deslizada" de acordo com os pacotes recebidos, que são consumidos pela "camada superior" (sistema de comunicação em grupo). A janela do transmissor é "deslizada" de acordo com as respostas recebidas pelos receptores, permitindo a transmissão de novos dados. Este mecanismo somente transmite novos pacotes de dados se o transmissor pode garantir que o pacote pode ser recebido e armazenado nos *buffers* dos receptores.

5 Arquitetura NewTOP + PRMP

Esta seção tece considerações sobre uma arquitetura que combinará PRMP e NewTOP em um *middleware* de grupo. O objetivo deste middleware, tal como [14], é propiciar a desenvolvedores facilidades para criar aplicações na Internet com requisitos de dependabilidade através do paradigma de comunicação em grupo.

Conforme ilustrado na Figura 2, a arquitetura é dividida em quatro camadas. A camada mais acima é a aplicação do usuário, que interage com o serviço de comunicação em grupo através de um conjunto de chamadas de método do NewTOP (conforme descrito na Seção 3). Para a aplicação, o NewTOP tem a responsabilidade de realizar a entrega atômica e ordenada de mensagens. O PRMP, logo abaixo, fornece o transporte multicast necessário para que os algoritmos do NewTOP sejam eficientemente aplicados. Por fim, a camada mais inferior corresponde aos níveis inferiores da pilha de protocolos TCP/IP, em particular UDP/IP multicast. IP multicast é necessário para a transmissão e roteamento multicast.

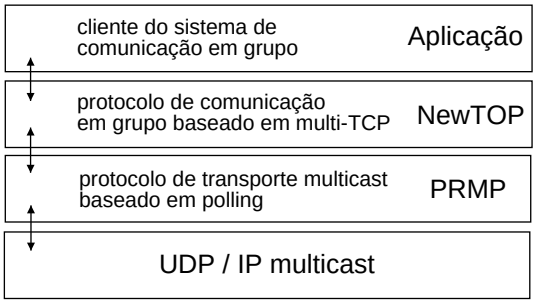


Figura 2: Arquitetura NewTOP + PRMP

A arquitetura exige a integração dos protocolos PRMP e NewTOP. NewTOP e PRMP foram projetados de forma independente. Desta forma, faz-se necessário a adaptação de

ambos os protocolos para que possam interagir e compor uma arquitetura integrada e eficiente.

O PRMP preenche os requisitos necessários para um serviço de transporte multicast como camada subjacente a um protocolo de comunicação em grupo ([3]). Estes requisitos são:

- o protocolo deve ter o controle sobre a quantidade de respostas enviadas pelos receptores para evitar a implosão;
- transmissão confiável "fim-a-fim" dos datagramas (ou bytes de um *stream*) para múltiplos receptores, detectando a perda de pacotes e recuperando-os;
- configuração e controle do grupo multicast, detectando falhas de *hosts* e particionamentos da rede;
- gerenciar os *buffers* do transmissor e receptor prevenindo a perda de pacotes desnecessária pela sobrecarga dos receptores, e
- auxiliar a prevenção de congestionamento nos gargalos da rede de uma maneira *TCP-friendly*.

Além disso, o PRMP deve:

- conhecer os membros controlados pelo NewTOP, bem como as informações geradas pelo *group-view*;detectar e informar ao NewTOP a falha de comunicação com os membros;
- repassar mensagens ao NewTOP, para que este as mantenha em uma fila até que as mesmas se tornem estáveis, e possa entregá-las à aplicação;
- lidar eficientemente com diferentes padrões de comunicação empregados pela aplicação: tráfego caracterizado pela troca de mensagens (potencialmente esporádicas) ou transmissão massiva de dados (grandes volumes).

Por fim, o NewTOP deve utilizar o PRMP para o envio das mensagens aos membros, de maneira eficiente, confirmada (ACKed) e sujeita a controle de congestionamento.

6 Considerações Finais

Os sistemas de comunicação em grupo necessitam de um protocolo de transporte multicast escalável para operarem de forma eficiente sobre a Internet. Neste artigo, considera-se uma arquitetura que combina o sistemas de comunicação em grupo NewTOP com o protocolo de transporte multicast escalável PRMP. O NewTOP oferece atomicidade, ordenamento de mensagens mais controle e gerenciamento dos membros de cada grupo. Em operações sobre a Internet, controle de congestionamento multicast deve estar presente de forma a haver um compartilhamento justo entre os fluxos TCP existentes e os fluxos gerados pelo IP multicast. O PRMP oferece a escalabilidade com controle de fluxo e congestionamento para o troca de mensagens entre os membros.

A próxima etapa é avaliar a arquitetura através de simulações utilizando o NS ([20]). Será implementado uma versão do PRMP e do NewTOP para o NS. Simulações do NewTOP utilizando multiplas conexões TCP e do NewTOP utilizando o protocolo multicast PRMP serão comparadas e avaliadas.

Agradecimentos

Agradecemos aos autores do NewTOP, em particular Graham Morgan e Dan Owen em Newcastle, pelo auxilio e discussões técnicas relativas à implementação do NewTOP.

Referências

[1] Y. Amir, et. al, "Transis: A Communication Subsystem for High Availability", FTCS-22, Boston, pp. 76-84, July 1992.

[2] Y. Amir, Danilov C., Stanton J., "A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication", FCTS-30, New York, June 2000.

[3] M. Barcellos, A. Detsch, H. Muhammad, G. Bedin, "Efficient TCP-like Multicast Support for Group Communication Systems", Brazilian Symposium on Fault-Tolerant Computing, SCTF 2001, Florianópolis, Brasil, pp. 192-206, 5-7 March 2001.

[4] M. Barcellos, P. D. Ezhilchelvan, "An End-to-End Reliable Multicast Protocol Using Polling for Scalability", In IEEE INFOCOM' 98, San Franciso, pp. 1180-1187, April 98.

[5] M. Barcellos, P. D. Ezhilchelvan, "PRMP: Poll-based Scaleable Reliable Multicast Protocol", Ph.D. Thesis, University of Newcastle, Newcastle upon Tyne, 200p., Oct. 1998.

[6] K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, Y. Minsky, "Bimodal multicast", ACT Transactions of Computer System, 17(2), pp. 41-88, May 1999.

[7] K. Birman, "Building Secure and Reliable Network Applications", Prentice Hall, 500p., 1996.

[8] K. Birman, A. Schiper, "Lightweight causal and atomic group multicast", ACM Transactions on Computer Systems, 9(3), pp. 272-314, August 1991.

[9] K. Birman, T. Joseph, "Reliable Communication in the Presence of Failures", Communications of ACM, 5(1), pp. 47-76, 1987.

[10] K. Chandy, L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", ACM Transactions on Computer Systems, 3(1), pp. 63-75, 1985.

[11] P. Ezhilchelvan, R. Macedo, S. Shrivastava, "Newtop: A Fault-Tolerant Group Communication Protocol", In IEEE 15th Intl. Conf. Distributed Computing Systems, Vancouver, pp. 296-306, May 1995.

[12] L. Lamport, "Time, clocks, and ordering of events in a distributed system", Communications of ACM, 21(7), pp. 558-565, July 1978.

[13] Lau C. L., J. Fraga, L. M. Souza, R. S. Padilha, "GroupPac: Um Framework para Implementação de Aplicações Tolerantes a Falhas", Conferência Latino Americanda de Informática, CLEI 2000, Cidade do México, México, 18-22 Setembro 2000.

[14] L. E. Moser, P. M. Melliar-Smith, "The InterGroup Protocols: Scalable Group Communication for the Internet", Globecom, Sydney, Australia, 14-16 December 1998.

[15] L. E. Moser, P. M. Melliar-Smith, "Totem: a Fault-tolerant Multicast Group Communication System", In Communications of ACM, 39(4), pp. 54-63, April 1996.

[16] G. Morgan, P. D. Ezhilchelvan, "Policies for using Replica Groups and their effectiveness over the Internet", Proceedings of the Internation Workshop on Networked Group Communication, NGC 2000, Palo Alto, California, USA, 8-10 November 2000.

[17] G. Morgan, "A Middleware Service for Fault-tolerant Group Communications", PhD. Thesis, Dept. of Computing Science, University of Newcastle upon Tyne, September 1999.

[18] G. Morgan, S. K. Shrivastava, P. D. Ezhilchelvan, M. C. Little, "Design and Implementation of a CORBA Fault-Tolerant Group Service", In 2nd IFIP WG 6.1 Internation Working Conference on Distributed Applications and Interoperable Services, Helsinki, June 99.

[19] S. Mullender, "Distributed Systems", ACM Press Frontier Series, Addison-Wesley, 580p., 1993.

[20] The network simulator - ns 2 - web site, http://www.isi.edu/nsnam/ns.

[21] K. Obraczka, "Multicast transport protocols: a survey and taxonomy", IEEE Communications Magazine, 36(1), pp. 94-102, Jan 1998.

[22] Qixiang Sun, D. Sturman, "A Gossip-based Reliable Multicast for Large-Scale High-Throughput Applications", Proceedings of the International Conference on Dependable Systems and Networks, DSN 2000, New York, 25-28 June 2000.

[23] R. Renesse, K. Birman, S. Maffeis, "Horus: A Flexible Group Communication System", Communications of ACM, 39(4), pp. 76-83, April 1996.

[24] S. Paul, K. Sabnani, J. Lin, and S. Bhattacharryya, "Reliable Multicast Transport Protocol (RMTP)", IEEE Journal on Selected Areas in Communications, 15(3), pp 407-421, April 1997.

[25] D. Towsley, J. Kurose, S. Pingali, "A Comparison of Sender-Initiated and Receiver-Initiated Reliable Multicast Protocols", IEEE Journal of Selected Areas in Communications, 15(3), pp. 398-406, 1997.

A multi–layer architecture for high available *Enterprise JavaBeans**

Marcia Pasin[1] Taisy Silva Weber[1] Michel Riveill[2]

[1] Universidade Federal do Rio Grande do Sul
Instituto de Informática
Av. Bento Gonçalves, 9500 – Campus do Vale
CEP 91501–970 Caixa Postal 15064
Porto Alegre – RS – Brazil

[2] Université de Nice – Sophia Antipolis
Ecole Supérieure en Sciences Informatiques/
930 route des Colles – BP 145
06903 Sophia Antipolis Cedex — France

Abstract

EJB (Enterprise JavaBeans) *spec* does not describe high availability as one of its properties. If the application server fails, the service remains unavailable while it recovers. Some EJB server vendors claim to provide this desirable property implementing server replicas through centralized protocols. Unfortunately, these protocols could lead to an unavailable service if the coordinator server crashes. We are presenting a new architecture aiming high available EJB servers based on distributed concepts. Our replicas are modeled as state machines synchronized by group communication primitives. We achieve high availability to EJB application servers running *stateful* and *stateless session beans*.

Keywords – high availability, replication, group communication, Enterprise JavaBeans.

1 Introduction

The EJB (Enterprise JavaBeans) *spec* [9] describes a component system architecture based on a multi–tier framework. This architecture mainly comprises the client, the EJB application server and the database server. Programmers build EJB applications in a transparent and adaptable way. They *develop* their applications (the *enterprise service* or functional requirements) without worry with non–functional properties (as persistency, security, transaction management and scalability), and *deploy* their applications using a *deployment tool*. The deployment tool automatically includes non–functional properties.

EJB applications can be *transaction–aware*. A transaction could aggregate operations on multiple objects. It is a sequence of methods encapsulated by a *begin operation* and a *commit operation*. If at least one method inside a given transaction cannot be completed due to a system failure, all updates generated by this transaction will be undone. The transaction execution will be aborted and an exception will be thrown.

Objects from the EJB architecture are called *bean*s. EJB application servers host and manage *beans* through a component called *container*. Clients could request operations (read or update a *state*) to any *bean* just running an EJB application. The main kinds of *beans* are *session bean* and *entity beans*.

A *session bean* just retains state during a client-server session. When the session terminates, its state is lost. The *session bean state* is volatile and unique to each client (one individual thread is used to each client). *Session beans* are not sharable. Otherwise, *entity beans* maintain their persistent states through a database connection and could be sharable between different clients. This work provides high availability to *session beans*. High available persistent states will be treated in a future work.

* This work was supported by the French Ministry of Research through project RNTL Arcad and by the Brazilian Ministry of Education through CNPq contracts 142808/98–9 and 200594/00–1.

The EJB *spec* assures a safe state to *beans* despite failures. However it does not guarantee high availability. If the EJB server goes down, its service will be unavailable: the state of a *session bean* is volatile and will be lost. High availability requires replication and synchronization protocols. Group communication has proven to be a convenient abstraction for implementing distributed systems requirements, particularly for synchronizing replicas [4]. Implementing group communication concepts in transactional systems, as the EJB architecture, is quite different [11].

We are extending the EJB *spec* to allow building replicated *bean*s without changing the way that users develop and deploy their EJB applications. Users do not have to worry with replica management. They can build their EJB applications in the usual (non–replicated) way and, optionally, could specify which service will be replicated using the *deployment tool*. Our replicated service exploits group communication to minimize communication costs.

High availability is archived through a multi–layer architecture that comprises a group communication layer, a replication layer and a conventional EJB layer. An open EJB application server implementation [5] provides the EJB layer. The replication layer provides consistency to the replicated EJB application servers. Consistency is achieved through a synchronization protocol following the *state machine approach* [7]. A group communication system [2] provides suitable services to the replication layer. These services assure group membership, failure detection and reliable multicast primitives even in presence of failures.

The paper follows presenting the section 2 with high available *session beans* requirements. The section 3 describes the distributed system model with the state machine approach. The section 4 describes our system design. The section 5 describes the replicated system implementation. The section 6 describes our preliminary results. The section 7 presents some related works. The paper ends with concluding remarks.

2 High available *session beans*

There are two kinds of *session beans*: *stateful* and *stateless*. *Stateful beans* retain state on behalf of an individual client. *Stateless bean*s are not aware of any client history. Recovery a server with *stateless session bean*s is straightforward because there is no information about the *bean* state stored in the server side. It requires the client reissue the request to another EJB server – it means *failover*.

Achieving high availability to *stateful session beans* requires replicate the *bean state* held during a *bean* method execution. Here we multicast this state to a replication group using group communication concepts. Implementing *failover* and maintaining the *exactly–once semantic* despite failures are also required. In the exactly–once semantic, the client makes a request and is guaranteed by the reply that the request has been executed.

3 Distributed system model

Our distributed system model is composed by clients and servers (EJB application servers). Clients could request read and update operations to objects (*beans*) placed in servers. Servers compose a *replication group,* and follow the *state machine approach* [7]. Here the state machine approach uses group communication primitives to achieve consistency to all group members. Initially the client request is executed locally at one server (the primary) and then the new state is multicast to all group members.

The *state machine approach* defines a consistent behavior to a collection of distributed objects. These objects run identical state machines (here, the EJB application servers) and perform the same sequence of operations, producing the same sequence of outputs and transitioning through the same sequence of states.

The behavior of the replicated objects is indistinguishable from that of a single high available object. Each client knows only the primary address and issues the request directly to it. The primary executes the service locally and then forwards its new state to the backups using a multicast primitive [4]. This primitive assures that all objects in the system (the *object group* – or, in our case the replication group) receive messages (state updates) in the same order, so that all objects perform the same sequence of state updates**.**

When a client wants to use high available service, it first contacts a *name service* to receive a unique *primary server address*. The other servers in the replication group work as *backups*. Then the client issues requests to this unique primary server. A request could be a read or an update operation to an object state (or *bean* state).

A server could work as *primary server* to a client and as *backup* to another because the *name service* could provide different primary addresses to different clients. This approach allows having multiple primaries simultaneously and avoids bottlenecks, a typical drawback of primary–backup approach.

Faulty backups are transparent to clients. Faulty primaries are not transparent to its clients and require the clients selecting a new primary and reissue the last request. The new primary discards the already done operations related to the issued request. Clients detect faulty primaries using *timeouts*.

We assume an asynchronous distributed system where neither message delays nor computing speeds can be perfectly bounded. Messages between different servers cannot be lost because an underneath group communication system provides reliable messages as well group membership. Group members can be assumed as *fault–suspected*, because there is no way to distingue between overloaded and faulty members. The group communication system also provides a *failure detector* to remove fault–suspected state machines from the object group. A fault–suspected state machine could be (repaired and) restarted and rejoins the object group by means of *state transfer* from surviving members.

4 System design through a multi–layer architecture

The high available service is provided by a multi–layer architecture (figure 1). Each server (primary or backup) has a group communication layer, a replication layer and an application layer. These layers are previously used by *Amir et al.* [1]. Here the application layer is played by the JOnAS EJB server [5]. The JavaGroups communication system [2] provides support to the group communication layer. These systems are open source and implemented using Java language. As both systems are implemented as component abstractions, its integration was straightforward.

4.1 Application server layer

The application server layer follows the EJB *spec* and manages the *beans* through a component called *container*. The container provides all non–functional properties (as persistency, security, transaction management and scalability). The *bean components* [9] are the *remote interface*, the *home interface*, the *bean class* and the *deployment descriptor*. These components are developed by the programmer. The *remote interface* is the client view of the *bean*. It contains the signatures of all *bean methods* (functional properties). The *home interface* contains the signatures of all methods for the bean life cycle (creation, suppression) and for instances retrieval (finding one or several *beans*) used by the client. The *bean class* implements the functional properties, and all methods allowing the *bean* to be managed in the EJB application server. The *deployment descriptor* contains the *bean properties* that may be edited at *configuration time*. The *beans* properties could identify, for example, if a bean is *stateful* or *stateless*.

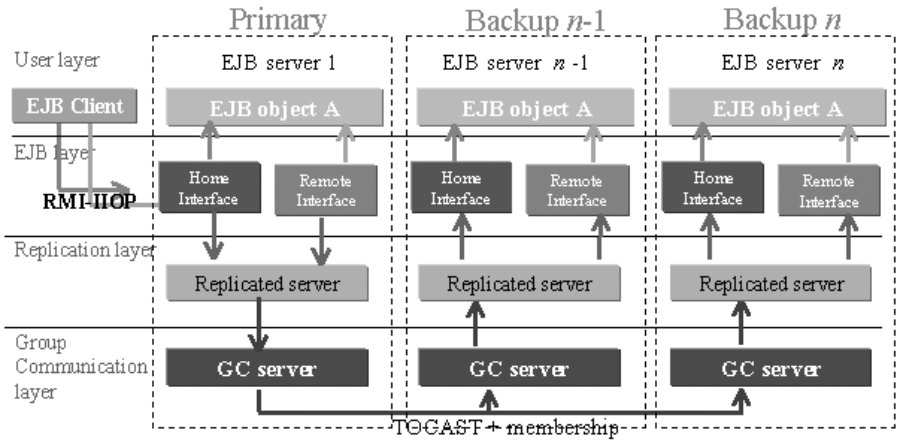


Figure 1 – The multi–layer architecture

4.2 Replication layer

The replication layer implements an interface between the EJB application server and the group communication layers. Whenever a local server receives a request from a client to execute an update method in a *stateful bean*, and after executing this method, it signals the replication layer. The replication layer generates a message containing the new bean state and forwards it to the group communication layer, which multicast this message to the group members using the TOCAST (*total–order multicast*) primitive [4]. Each group member receives the message in its local group communication layer and delivers it to the replication layer. By delivering the same set of messages using the TOCAST primitive, the algorithm guarantees that all servers hold the same bean state to a given client. If one application server in the group fails, clients are guaranteed access to the same state through the backups.

To minimize communication costs, only new states are multicast to backups. The replication layer could be able to distinguish updates from simple reads to *bean*s, either identifying the executed method or comparing the new signaling state with the stored previous one. Examining the *bean code* in pre–compiling time, it is possible to identify the methods that could potentially chance the *bean state*.

Update propagation and transactional attributes

Update propagation is an important drawback of distributed replicated systems because it can decide the system performance. There are two different strategies to propagate updates: *deferred update* and *immediate update*. The deferred update strategy processes all transactions locally at one server (the primary one) and forwards the last final result to the others at commit time. Immediate update synchronizes every update across all servers.

Although deferred update has advantages over immediate update, as reducing the number of distributed states, it requires executing the service from the last committed transaction when a failure happens. Both strategies could be implemented to analyze their efficiency in a comparative study.

The EJB *spec* supports both *transaction–aware* and *non–transaction–aware bean*s. *Non–transaction–aware beans* requires using immediate update whenever. *Transaction–aware beans* require analyzing the transactional attribute available in the *bean deployment descriptor*. The `NotSupported` and `Never` transactional attributes ever require immediate update. `Required`, `RequiresNew`, `Mandatory` and `Supports` attributes support both immediate update and deferred update. The `RequiresNew` attribute supports both

immediate and deferred update. Both execution effects are the same because it requires a new transaction per each remote method invocation.

4.3 Group communication layer

The group communication layer is implemented by the JavaGroups group communication system [2]. The group communication system uses available protocol layers responsible for achieving group membership, total ordering of messages, and other properties as building blocks with which the high–level state machine replication semantics are obtained.

5 Implementation

This section describes the implementation of the replication layer. The implementation addresses the server–side and client–side, and both kinds of *session beans* (*stateless* and *stateful*). At the client–side, *failover* enables selecting a new server to provide access to *stateless* and *stateful session beans*. At the server–side, the replication layer overloads a non–replicated JOnAS server to multicast distributed state from *stateful session beans*.

5.1 Failover

In the EJB *spec*, before a new *bean instance* is created, the client executes a *lookup operation* using a JNDI (Java Naming and Directory Interface) server to contacts the name service. The JNDI server provides an application server address that can be used by the client to create a *bean instance* in the EJB server using the *home interface*. Then the client could access the *bean methods* using the *remote interface*. To finalize, the client destroys the *bean instance*.

Implementing a high available service requires locating the *bean* in a backup when a failure occurs – it means *failover*. The *lookup operation* is reexecuted by the client to locate another *bean copy* in the backup server. Then a *new bean instance* is created in this backup. *Stateful beans* require state update from the last state received by the multicast message.

We implement failover redirecting faulty requests in the client–side. *Hooks* could be transparently included in the client–side by a pre–compiler. A *hook* is a concept borrowed from the EMACS editor, which allows executing arbitrary commands before performing some operation. We change Java exceptions to both *lookup* and *create bean* operations by new operations to allow failover. This service is transparent to the programmer.

5.2 Replicated servers

As we saw, our replicated server implements a new replication layer between JavaGroups and JOnAS. This layer is responsible to manage the replication group. This is done by overloading classes of the non–replicated JOnAS server to include the TOCAST primitive and introducing *hooks* in the *bean class*. Replicated servers join the group and use this primitive to setting the distributed state to *stateful session beans*. *Stateful* and *stateless session beans* are distinguished by means of code inspection using the *bean deployment descriptor*.

Code inspection is also required to distinguish read from update methods. It could be done observing by introspection the method signature in the *bean remote interface*. A non–void class could be interpreted as an update operation. Classes with void as return parameter could be assumed as a read operation. Optionally, the user could specify which methods will be replicated using the deployment tool.

To the programmer, high available *beans* are built in a transparent way, by using a pre–compiler. This pre–compiler changes the *bean* code to include *hooks* that enable the high

available service takes place. This approach is also used by the GenIC compiler from JOnAS [5], which includes non–functional properties using the bean interfaces.

Distributed state

A *distributed state* contains information about active *stateful bean*s in the replication group. Each distributed state holds information about the last *bean state*, the client who is using this *bean* and the primary server who is proving access to the *bean methods*.

The EJB *spec* enables one or more *create methods* to a *bean*. These create methods can differ by the parameters sent to each method. To implement the replicated service, we could overload all create methods with a new parameter called *failover*. Having the parameter failover as true value means that the new *bean instance* should be update with the value stored in the last distributed state, because the *bean* execution was failover. Once the service is done and the client disconnects, a particular *bean instance* is destroyed. Then all group members remove all distributed states from this client–server session.

Remote invocation in the presence of failures

The current EJB *spec* supports transactional services over non–reliable infrastructures. It implements the *best–effort execution semantic*. In the *best–effort semantic*, the client sends the message, and the client and infrastructure do not attempt retransmissions. High available services require a more sophisticated approach despite failures. *Five different classes of failures* [10] can occur in *remote procedure call* systems. These same failures could also occur in *remote method invocation* systems. They are required be treated by our replicated system because the EJB architecture is based on *remote method invocation.*

The first one, *the client is unable to locate the server*, here is treated providing failover. *Stateless session beans* require the *at–least–once semantic*: the client makes a request and retries the request until it receives the response. If a failure occurs, the client is enable try all group member candidates, which are specified in a list by the system administrator. If none is available, in the worst case, the client finally throws an exception. Duplicate message processing by the client is not a problem, because the *bean state* is not retained in the server–side.

Stateful session beans require the *at–most–once* or the *exactly-once semantics*. In the *exactly–once semantic*, the client makes a request and is guaranteed by the reply that the request has been executed. An approach to assure *exactly-once semantic* in replicated transactional systems is showed in *Frolund et al* [3]. The *at–most–once* builds on the *at–least–once* scenario. The client retries the request until it gets a response. A mechanism like *message identifiers* allows the server to suppress any duplicate requests, insuring the request is not executed multiple times. We follow this approach.

The second one, *the request message to the client to the server is lost*, is treated using *timeout* in the client–side, following the EJB *spec*. The third one, *the response from the server to the client is lost*, is treated using *timeout* in the client–side. In this case the client reissues the request to the same server. The message identifiers allow the server to suppress any duplicate requests. The forth one, *the server crashes after receiving a request*, we also treat using message identifiers to assure the *at-most-once semantic*. Finally, the fifth one, *the client crashes after sending a request*, could potentially generate *orphan bean instances*. The EJB *spec* treats *orphan bean instances* through the container. It periodically removes all *bean instances* from the server memory if they are not current used. We need extend this approach to remove all correspondent distributed state established to that client.

6 Preliminary results

Initially our implementing effort was focused in changing the GenIC compiler provided by the JOnAS EJB server. The programmer builds its *beans* and uses the GenIC compiler to mainly generate the container classes. However, the GenIC acts over the *home* and the *remote interface*. We need acts over the *bean class* that actually holds the *bean state*. So we need include *hooks* in the *bean class* not in the *home* and *remote interface* as GenIC does.

Presently, we are testing our implementation for automatic re–routing of clients' requests (client side) and the approach to establishing the distributed state to *stateful session beans* (server side). A performance study will take into account different replica number and failure scenarios. We expect that replication does not considerably disturbs the application response time, when compared with non–replicated application servers, by allowing requests to be handled by several nodes rather than one besides eliminating a single point–of–failure.

7 Related works

Some EJB application server providers implement high availability to *session bean*s. They mainly use the *in–memory replication technique*. In–memory replication has two different variations. The first approach writes information to a centralized server (all servers in the cluster use the same centralized server). In the second approach, each server chooses an arbitrary backup. The BEA WebLogic 6.0 uses this last approach and the HP Bluestone Total–e–server uses the first one. However, these and other solutions [6] are implemented using *cluster concepts*. We are using a more non–restrictive model. Our system runs over a local network and could support asynchronous communication.

There is no accurate information available about how state propagation is applied in other high available EJB servers. The majority of the implementations are proprietary solutions. However, these implementations seem forwarding distributed states using centralized approaches as *n–phase commit protocols.* Two–phase commit protocols are mainly used to assure consistent states in distributed transactional systems. In these protocols, a process called *coordinator* applies distributed state to all replicas. They could accept human intervention to solve abnormal behavior because they may block if the coordinator crashes. Three–phase commit protocols provide higher availability, but they assume a restricted fail–stop model. Group communication abstractions allow non–blocking protocols and support a wider failure model. It has proven to be a convenient abstraction to improve distributed application availability.

Some related work is showed in *de Sousa* [8]. That work also differs from our work mainly because we use a group communication system to provide reliable communication through EJB servers. It assures that our *multicast protocols* are already extensively validate avoiding implementing our own protocols and minimizing programming errors. The solution described in *de Sousa* [8] does not mention several aspects as enabling multiple primaries simultaneously to avoid bottlenecks. Replication (and high availability – our aim) is not the main aim of that work: its main aim is explore the reflexive features of the EJB architecture.

Our work also differs from CORBA solutions implemented around both the `CosNaming` service and a centralized JNDI tree. The Sybase Enterprise Application Server uses this approach in a cluster. Name servers house the centralized JNDI tree for the cluster and keep track of which servers are on–line. If there is a failure in between EJB method invocation, the CORBA stub retrieves another home or remote interface from an alternate server returned from the name server. The name server is the drawback of this solution because it needs to remain active in one node at least. If all nodes that hold JNDI servers instances are down, the system will be unavailable. Finally, centralized JNDI tree clusters suffer from an increased time to convergence (the time the cluster takes to know all its server active instances) as the

cluster grows in size. That is, scaling requires adding more name servers. Our solution does not require time to convergence because we are using the independent JNDI tree approach.

8 Concluding remarks

Transactional systems could benefit from group communication to achieve availability [11]. Nevertheless group communication introduces the membership management overhead that can be reduced using suitable design decisions. Here, these decisions include excluding clients from the replication group (to avoid frequent membership changing) and propagating to backups only updates or committed results. It allows eliminating additional communication rounds and improves performance. We also expecting reducing the overhead generated by the replication protocol enabling several servers working as primary servers.

References

[1] Amir, Y.; Dolev, D.; Melliar–Smith, P. M.; Moser, L. E. *Robust and efficient replication using group communication.* Technical Report CS9420, Institute of Computer Science, Hebrew University of Jerusalem, Nov. 1994.

[2] Ban, B. *JavaGroups user's guide*. Department of Computer Science, Cornell University. August 1999. 73p.

[3] Frolund, S. and Guerraoui, R.; *Implementing e–transactions with asynchronous replication*. Proceedings of the International Conference on Dependable Systems and Networks 2000, New York, IEEE, June 2000.

[4] Guerraoui, R.; Schiper, A. *Fault tolerance by replication in distributed systems*. In Proc. Conference on Reliable Software Technologies (invited paper), p. 38–57. Springer Verlag, LNCS 1088, June 1996.

[5] *JOnAS – Java Open Application Server* – http://www.objectweb.org/~jonas

[6] Kang, Abraham. *J2EE clustering, Part 1. Clustering technology is crucial to good Website design; do you know the basics?* JavaWorld. Feb. 2001. http://www.javaworld.com/javaworld/jw–02–2001/jw–0223–extremescale.html

[7] Schneider, F. B. *Replication management using the state machine approach*. In: Mullender, Sape (Ed.). Distributed Systems. 2. ed., New York: ACM Press, 1993. p. 169–198.

[8] de Sousa, C.V.P.B.; Maziero, Carlos Alberto. *Uma abordagem reflexiva para replicação de componentes servidores da plataforma Java para corporações*. WTF 2000. Curitiba – PR, pp.106–111.

[9] Sun Microsystems, Inc. *EJB specification* 2.0.

[10] Tanembaum. A.S. *Communication in distributed systems*. In: Modern Operating Systems. Prentice Hall, New Jersey 1992. pp.395–462.

[11] Wiesmann, M.; Pedone, F.; Schiper, A.; Kemme, B. and Alonso, G. *Understanding replication in databases and distributed system*s. Proc. ICDCS 2000, pp.264–274, Taipei, Taiwan, R.O.C., April 2000.

Implementing FT-CORBA with Portable Interceptors: Lessons Learned

Fabíola Greve[⋆†] Jean-Pierre Le Narzul[• †]
{fgreve|jlenarzu}@irisa.fr

⋆ LaSiD-DCC-UFBA
Campus de Ondina
40170-110 Bahia, Brasil

† IRISA
Campus de Beaulieu
35042 Rennes, France

• ENST-Bretagne
Campus de Rennes
35512 Cesson-Sévigné, France

Abstract

In this paper, we present the design of Open EDEN, an implementation of the FT-CORBA specification based on the use of a group communication framework, called EDEN. The design of Open EDEN was driven by the desire to use only portable techniques (mainly portable interceptors) to integrate the EDEN framework within a CORBA platform. We discuss the main difficulties we encountered and we draw some conclusions about the adequacy of this choice.

1 Introduction

Fault tolerance in CORBA (Common Object Request Broker Architecture) is provided through object replication. A critical CORBA service can be made reliable by replicating a CORBA object implementing it on different sites of the distributed system. The set of replicas forms an object group. If sites fail independently, an invocation to the service will succeed even if some replicas of the group have crashed.

Apart redundancy, fault tolerance in CORBA relies also on fault detection and notification, logging and recovery. All the components of a FT-CORBA architecture are described in an OMG specification [5] which has been recently adopted by the OMG board. The specification describes the interfaces of the services, different fault tolerance strategies and fault tolerance properties associated to an object group.

The designer of a FT-CORBA architecture is left with a lot of freedom in choosing the way the components of this architecture are implemented and the way they interact. Consequently, there have been several efforts to add fault tolerance in CORBA systems. Some of these efforts adhere very closely to the FT specification ([4, 7, 1]); others are voluntarily less FT-CORBA compliant and favor innovative solutions ([2]). Among these efforts, we can distinguish three major approaches [4]: (i) the service approach in which clients and servers must explicitly interact with a specific service to get reliability; (ii) the integration approach in which a group communication framework providing replication functionnality is integrated inside the ORB; (iii) the interception approach in which, requests or messages (depending on the location of the interception layer) are intercepted and redirected to a group communication framework.

Of course, each of these approaches presents advantages and drawbacks. The service approach is probably the simplest to implement but has a major drawback that is the lack of transparency. The integration approach is the most efficient but cannot be easily re-used between ORBs as it is very dependent on ORB internals. The interception approach offers transparency but, depending on how it is implemented, can be non-portable (from a system point of view) if the interception layer is placed between the ORB and the system interface.

We are currently prototyping an architecture exclusively based on the interception approach for supporting active replication of CORBA objects. For this development, we decided to use only portable techniques. By portable, we mean (1) portable with respect to the system support; (2) portable with respect to the group communication system; (3) portable with respect to the ORB. The first condition precludes the use of system-level interceptors like in Eternal [4]. The second condition precludes a tightly coupling between some components of our FT architecture and the group communication system. The third condition leads us to use a specific feature of CORBA which are portable interceptors [6]. Portable interceptors are the only portable mechanism specified by the OMG which allows to transparently add service code to the ORB. The objective of this paper is to describe this experience in using portable interceptors for prototyping a fault-tolerant CORBA system and to draw some conclusions about the adequacy of this technique. Our prototype is based on a group communication system called EDEN.

The remainder of this paper is structured as follows. Section 2 briefly describes the EDEN group communication framework. Section 3 summarizes the FT-CORBA specification and introduces CORBA portable interceptors. Section 4 presents our prototype which implements a FT-CORBA service based on the use of the EDEN framework. Section 5 lists some of the problems we encountered. Section 6 concludes this paper.

2 The EDEN Group Communication Framework

A group service allows a collection of related objects to be considered externally as a single logical entity. This paradigm is used to simplify replication management and construct fault-tolerant applications. Because of its importance, many group communication frameworks have been constructed [13]. The group service includes different primitives used by group members to coordinate their activities. One of those primitives is called *atomic broadcast*, which assures that all requests received by the group entity are delivered to its members in the same order. Another fundamental abstraction is the *group membership*. This service tracks changes in the group composition, due to the desire of members to join or leave the group, or the occurrence of crashes. The *view synchrony communication* allows the delivery of requests "in synchrony" with the delivery of changes in the group composition. All these abstractions are considered as agreement problems that can be solved as a reduction to the *consensus* problem [8, 9].

EDEN is a configurable group-based toolkit, which aims at developing reliable, object-oriented, distributed applications. It is formed of two components: a library of protocols (named ADAM), and an event-based architecture for structuring services (named EVA [12]).

ADAM is a library of autonomous agreement protocols which can be pieced together to implement reliable services. At the heart of this library is a General Agreement Framework (GAF) [9] suited to generate ad hoc agreement protocols. GAF is a Chandra-Toueg [8] consensus-based service which implements some versatility parameters that encompasses many interesting features (early decision, multiple propositions of initial values, decision on a vector of values, deferring of initial values proposals) suitable to generate efficient protocols. In order to solve a particular agreement protocol one must instantiate the parameters with appropriate values.

Clever instantiations of the GAF functions have been investigated in order to implement the group membership [11], atomic broadcast and view synchrony communication protocols.

The framework EVA [12] implements a publish-subscriber communication environment to structure entities composing high level protocols. In this architecture, protocols are regarded as a number of cooperating objects (entities) that communicate via an event channel.

3 CORBA

3.1 FT-CORBA Standard

The FT CORBA specification defines a set of standard interfaces for replication management, fault management and logging/recovery management. One of the main components of the FT-CORBA architecture is the replication manager which interfaces allow to specify the fault tolerance properties of an object group. The replication style is one of this property for which three values are possible: `COLD_PASSIVE`, `WARM_PASSIVE` (passive replication is based on the use of a primary replica) and `ACTIVE` (all replicas play the same role in the group).

One important point in the specification is the structure of an object group reference. Such reference must be built according a standard schema in order to allow inter-operability between client and server hosted by heterogeneous infrastructures. The specification defines the use of multiple `TAG_INTERNET_IOP` profiles to identify replicas (or a set of gateways) and introduces a new component `TAG_FT_GROUP` to encapsulate an object group identifier. An object group is identified by a Group IDentifier (GID) which does not depend of the group membership. At group creation, the replication manager allocates the GID for the group and then constructs an object group reference populated with this GID.

The specification does not define a standard multicast group communication protocol to be used for maintaining consistency between replicas. Consequently, there is no inter-operability between ORBs hosting server objects ; all the replicas of a server object group must be hosted by the same FT infrastructure. This particularity allows FT CORBA designers to choose the most appropriate protocol for their infrastructure.

3.2 Portable Interceptors

Portable interceptors are hooks into the ORB through which ORB services can intercept the normal flow of execution of the ORB. The specification [6] defines IOR (Interoperable Object Reference) interceptors and request interceptors. The former allows to add specific components inside a IOR during its creation. The latter allows to intercept the flow of a client invocation to a server at different points; some of these points are located at the client side and others are located at the server side. A client request interceptor allows to execute service code before the request is sent to the server and before the reply is returned to the client code. Using a specific CORBA exception mechanism, the interceptor code can change the target of an invocation. A server request interceptor allows to execute service before delivering the request to the server object and after the server object has completed its job.

4 Prototyping an EDEN Based FT-CORBA Service

4.1 Overall Architecture

As mentioned in the introduction of this paper, our goal is (1) to design a portable and flexible fault-tolerant service for CORBA and (2) to experiment the usability of EDEN as a group communication support for such a service. For the sake of simplicity and because we did not focus on a full FT-CORBA prototype, we decided the following restrictions:

- We consider only active replication (`ReplicationStyle` property set to `ACTIVE`). Consequently, logging and recovery mechanisms (only needed for passive replication) are not implemented.
- Membership and consistency are controlled by the infrastructure (`MemberShipStyle` and `ConsistencyStyle` properties are respectively set to `MEMB_INF_CTRL` and `CONS_INF_CTRL`).
- Fault detection is not accessible via CORBA interfaces.

Figure 1 shows the main components of our architecture (named Open EDEN) : (1) a gateway [1] which acts as an intermediary object on the request path between a client and a server object group; (2) a CRPI component, which is a Client Request Portable Interceptor attached to a client and used to redirect client requests to a gateway; (3) a SRPI component, which is a Server Request Portable Interceptor attached to each server hosting a replica and used to implement the interface with the EDEN framework.

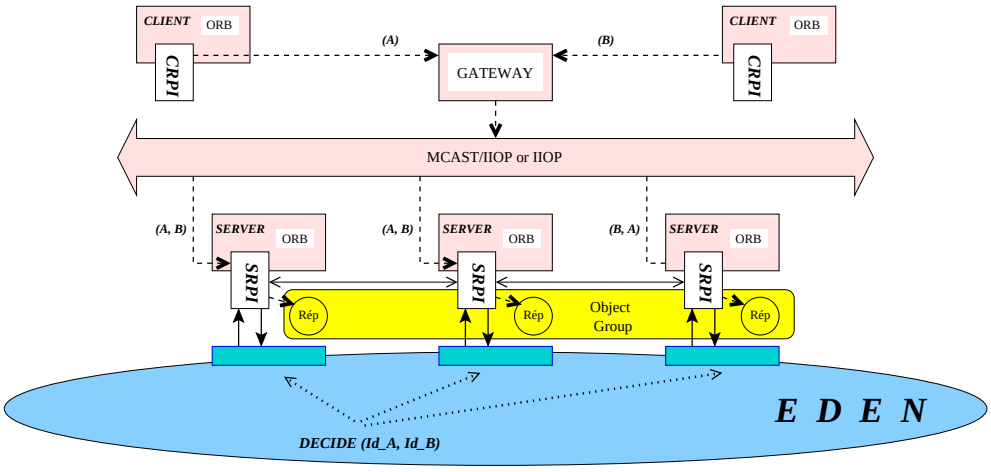


Figure 1: Overview of the Open EDEN Architecture

4.2 Client Redirection

The client of a server object group obtains the object group reference through usual ways, i.e. either by using a call to the CORBA Naming Service or by reading it from a file. Then, the

[1] We only represent one gateway in the figure but for fault-tolerance reasons, our architecture supports multiple gateways.

client transparently (i.e. without knowing that the target object is an object group) invokes the method of the server object group. At this point, a client interceptor intercepts the call, extracts identification of gateways and the group identifier from a tagged component and finally redirects the request towards one of the available gateways.

As represented in Figure 1, the protocol used by the gateway to address an object group can be either IIOP or MCAST/IIOP. In the former case, the group invocation is accomplished by using multiple unicasts. In the latter, the invocation is multicasted to the group. We will not give details in this paper on how the gateway obtains the multicast reference (or list of references); in our prototype, the mapping between the object group identifier and references is implemented by the replication manager.

4.3 Replica Consistency

Infrastructure-controlled consistency is one of the main requirements of the FT-CORBA specification. As stated in the specification, it means that for the `ACTIVE` replication style, "at the end of each method invocation on the object group, even in the presence of faults, the members should have the same state". To maintain group consistency, it is necessary that (1) each member behaves deterministically and that (2) the same sequence of requests be delivered in the same order to each member of the group. Regarding the first condition, like Narasimhan in [3], we consider that the unique source of undeterminism is due to potential concurrent executions of threads; we provide a specific scheduler to solve it. For the second condition, thanks to the EDEN framework, we built an adequate delivery protocol. In the following paragraphs, we discuss strategies we used to (1) re-order the requests, (2) to ensure their reliable delivery and (3) to finalize their execution. All these strategies are implemented in a Server Request Portable Interceptor (SRPI) which interfaces CORBA with the EDEN framework.

To minimize the quantity of information exchanged between the two worlds (CORBA and EDEN) and to optimize the performance of the protocols inside the EDEN framework, we decided that the SRPI components will feed EDEN with requests identifiers (instead of whole requests). The consequences of this choice are also examined in the following paragraphs.

Re-ordering requests Each time a client request is intercepted, its thread is blocked and the SRPI component proposes its identifier (request identifier) to the EDEN framework. At each site, EDEN collects the proposed identifiers and runs a consensus protocol with its peers to agree on a set of identifiers and a single delivery order for them. When a decision (or a set of decisions) has been taken, the EDEN framework calls the SRPI components and gives them an ordered list of identifiers; each SRPI component can then mark the corresponding requests as deliverable. To protect against concurrent threads executions, a scheduling algorithm is implemented by each SRPI component to serialize the execution of the deliverable requests by their associated threads.

Reliable delivery Due to the unreliability of the MCAST protocol or the intermediate view of the replication manager, an SRPI component could be asked by the EDEN framework to deliver a request that it has never received. To solve this problem, we added a specific operation to the CORBA-EDEN interface. When the EDEN framework detects that a particular SRPI has not proposed an identifier for which delivery has been decided, it calls the SRPI to give it a list of SRPI components able to forward the missing request. Another problem arises when EDEN is unable to decide a single delivery order for a request (or a set of requests) because there is no

majority of propositions for the request (or for the set). In such a case, the SRPI components which have proposed will be asked to forward the request (or the set) to the SRPI components which have not proposed. Therefore, the SRPI components must put each new incoming request in a log before proposing its identifier to the EDEN framework.

Stabilizing When all SRPI components have proposed a same set of request identifiers, it is no more necessary for the SPRI components to keep a log of the corresponding requests. This situation is detected by the EDEN framework which calls a specific operation of each SRPI component to ask them to remove requests from the log.

5 Lessons Learned

We were confrontated with some difficulties in implementing our prototype of a FT-CORBA service solely based on the use of portable interceptors to interact with the EDEN framework. In the following paragraphs, we enumerate some of the most important problems and we discuss about the advantages and inconvenients of alternative solutions.

1. Redirecting client invocation. Due to known limitations of client portable interceptors [1] (impossibility to generate a reply or to filter among multiple replies), it's not possible to build a FT-CORBA client without the help of an intermediary object.

   That is the reason why we decided to redirect to a gateway a client invocation destinated to an object group. The gateway, which uses the CORBA dynamic interfaces, DSI and DII, behave like a server for the client and like a client for the server object group.

2. Generating object group reference. Creating a group reference is an operation driven by the replication manager. When it is being asked to create a server object group by an application, the replication manager needs (1) to create the replicas by calling the factory operation `create_object` and (2) to generate a reference identifying the group. Unfortunately, since the replication manager is a regular CORBA object without any access to the internal ORB interfaces and functionalities, it has very few means for manipulating references. The only way to act, at a service level, on the creation of a reference is to use IOR interceptors (one of the three categories of portable interceptors). IORInterceptors only enable a server to add components. Thus, using IORInterceptors is insufficient for replication manager because: (1) IORInterceptors do not provide for adding profiles, only components; (2) IORInterceptors are not invoked for already existing references, only for newly created ones; and (3) even for newly created references, it is underspecified when IORInterceptors are invoked.

3. Language Mapping. We implemented the server part of our architecture in C++. In the C++ mapping, the interceptor is allowed to access all the attributes of the `RequestInfo` structure. This way, we could get the arguments attribute and log a request such that the SRPI component be able to forward it to others when necessary. Unfortunately, with the Java portable bindings, the arguments attribute is not available. Consequently, we could not port the server part of our solution using the Java language.

4. Thread Model Dependency. The strategy we used to re-order CORBA requests is strongly related to the concurrency model selected at the server-side and to the way portable interceptors are implemented. Using a "thread-per-request" model at the server-side and

considering that, in ORBacus, interceptors are implemented in a manner such that the thread executing the interceptor code also executes the user server code, we managed to implement a scheduling policy that re-order requests (based on the EDEN decisions). Any modification to the implementation of portable interceptors in the ORB (still compliant to the specification) could lead us to reconsider our implementation. Consequently, even if we have used only portable mechanims (like portable interceptors) to implement requests re-ordering, we need to carefully examine how interceptors are implemented before to contemplate migrating to a new ORB.

6 Conclusion

Our goal was to prototype a FT-CORBA service by using the most portable features of CORBA to interact with the EDEN group communication framework. We can summarize the results of this exercice by saying that client portable interceptors are useful tools to redirect client invocation and that server portable interceptors are also useful but not sufficient to provide all required functionnality at the server side.

We are now investigating the design of a new architecture in which the EDEN framework is still cleanly interfaced with an ORB but in which interfaces could be exposed to application programmer. Of course, in that way, we loose portability but one of our goals is now to bring to developpers of fault-tolerant CORBA application the benefits of the modularity of the EDEN framework. For instance, in our prototype, we considered only infrastructure controlled consistency. Application controlled consistency is another possibility allowed by the FT-CORBA specification. In that case, it could be interesting to provide to the application developper high-level services like consensus to help it to customize specific consistency protocols.

Acknowledgements

We would like to thank Michel Hurfin and Roman Vitenberg for helping with the design of the interface between CORBA and EDEN.

References

[1] R. Baldoni, C. Marchetti, and L. Verde. ”CORBA Request Portable Interceptors: Analysis and Applications”. In *Proceedings of the 3nd International Symposium on Distributed Objects and Applications (DOA 2001)*, pages 208–217, sep 2001.

[2] R. Friedman and E. Hadad. A Group Object Adaptor-Based Approach to CORBA Fault-Tolerance. In *IEEE Distributed Systems Online*, volume 2. Special Issue on Middleware 2001, 2001.

[3] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith. Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 263–273, sep 1999.

[4] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith. The Interception Approach to Reliable Distributed CORBA Objects. In *Proceedings of the third USENIX Conference on Object-Oriented Technologies and Systems* , jun 1999.

[5] OMG. Fault Tolerant CORBA Specification. V1.0. ptc/00-04-04.

[6] OMG. Portable Interceptors. ptc/01-03-04.

[7] N. Wang, K. Parameswaran, and D.C. Schmidt. The Design and Performance of Meta-Programming Mechanisms for Object-Request Broker Middleware. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)*, feb 2001.

[8] T. Chandra, S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(1):225–267, 1996.

[9] M. Hurfin, R. Macêdo, M. Raynal, F. Tronel, A General Framework to Solve Agreement Problems *Proc. 18th IEEE Symp. on Reliable Distributed Systems - SRDS*, pp. 56-65, 1999.

[10] R. Guerraoui, A. Schiper, The Generic Consensus Service. *IEEE Transactions on Software Engineering*, Vol. 27, No.1, pp. 29-41, January/2001.

[11] F. Greve, M. Hurfin, M. Raynal, F. Tronel, Primary Component Asynchronous Group Membership as an Instance of a Generic Agreement Framework. *ISADS'2001: 5th International Symposium on Autonomous Decentralized Systems*, pp 93-100, March 2001.

[12] F. Brasileiro, F. Greve, M. Hurfin, J-P Le-Narzul, F. Tronel, EVA: an Event Based Framework for Developing Specialised Communication Protocols. In *NCA'2001 (IEEE International Symposium on Network Computing and Application)*, october 2001.

[13] D. Powell, Group Communication, *Communications of the ACM*, Guest Editor, vol. 39, num. 4, pp. 50–53, april 1996.

Recuperação com Base em *Checkpointing*: uma Abordagem Orientada a Objetos

Francisco Assis da Silva[1,2], Ingrid Jansch-Pôrto[2], Maria Lúcia Lisbôa[2]

chico@apec.unoeste.br, ingrid@inf.ufrgs.br, llisboa@inf.ufrgs.br

[1] Faculdade de Informática de Pres. Prudente
Universidade do Oeste Paulista – UNOESTE

[2] Pós-Graduação em Ciência da Computação
Instituto de Informática – UFRGS

Resumo

A recuperação de processos visa propiciar maior disponibilidade para aplicações computacionais, minimizando o tempo total de execução quando da ocorrência de falhas. Com o advento do paradigma da orientação a objetos, novos problemas foram introduzidos na atividade de recuperação quanto aos mecanismos de salvamento de estados e retomada da execução. Tendo em vista prover tolerância a falhas para aplicações computacionais na presença de falhas do sistema, e contribuir na execução do processo de recuperação de aplicações orientadas a objetos escritas em Java, objetiva-se neste trabalho o desenvolvimento de uma biblioteca que implementa os mecanismos de checkpointing e recuperação. Para a concepção do trabalho, são considerados ambos os cenários no paradigma orientado a objetos: objetos centralizados e distribuídos. São utilizados os recursos da API de serialização Java e a tecnologia Java RMI para objetos distribuídos.
Neste artigo, é descrita a biblioteca proposta; são comentados alguns aspectos de implementação e resultados obtidos a partir da avaliação funcional e de desempenho temporal dos mecanismos. A apresentação de conceitos básicos restringe-se aos relacionados à área de orientação a objetos empregados no artigo.

1. Introdução

Na área de Tolerância a Falhas, a busca de parâmetros adequados na propriedade de disponibilidade dos serviços computacionais, mesmo em presença de falhas do sistema, traz consigo o interesse na pesquisa das atividades de recuperação em sistemas computacionais. Do ponto de vista prático, isto se traduz na necessidade de minimizar o tempo total de execução de aplicações computacionais de longa duração, ao mesmo tempo em que as prepara para não sofrerem perdas significativas de desempenho, em caso de falhas. A técnica tradicional de recuperação de processos em um sistema computacional baseia-se na restauração de um processo ou conjunto de processos para um estado operacional normal (estado livre de erros).

Os estados salvos durante a execução de um processo, para os quais ele possa mais tarde ser restaurado, são conhecidos como pontos de recuperação ou *checkpoints* [1]. O termo *checkpointing* é usado para a ação de estabelecer o ponto de recuperação. O estado de um programa em execução deve ser periodicamente salvo para um meio estável (protegido contra falhas do meio de armazenamento), do qual pode ser recuperado a partir da detecção do erro [2]. Este enfoque orientado a processos é bastante tradicional; tendo sido explorado em vários trabalhos, incluindo o desenvolvimento de bibliotecas, tais como a *libckpt* [2].

Em uma abordagem orientada a objetos, salvar dados de uma aplicação em execução é uma operação requisitada com freqüência. Essa representação de dados feita no armazenamento estável (representação externa) é tipicamente muito diferente da mesma representação de

dados de um programa em execução (representação interna). Ao buscar os dados para reutilização, a partir do armazenamento estável, eles necessitam ser convertidos da representação externa para a representação interna. Em um programa orientado a objetos, os dados são representados em forma de objetos, e produzir objetos persistentes é uma maneira eficiente de salvar os dados da aplicação no armazenamento estável [3].

Segundo Lawall e Muller [4], programas escritos em uma linguagem orientada a objetos, tal como Java, introduzem novas demandas de *checkpointing*: (i) o estilo de programação orientado a objetos propicia a criação de muitos objetos pequenos. Cada objeto pode ter alguns campos que são somente de leitura, e outros que são freqüentemente modificados; (ii) o programador de Java não tem nenhum controle sobre a alocação de objetos. Assim, é impossível assegurar que objetos modificados freqüentemente estejam todos armazenados na mesma página de memória. Além disso, uma única página pode conter objetos vivos e objetos que aguardam o coletor de lixo para serem desalocados da memória; (iii) programas Java são executados em uma máquina virtual que suporta processos simultâneos. Assim, a linguagem Java encoraja paralelismo como um método de engenharia de software; bibliotecas de programação, como para tratar a interface gráfica do usuário, criam muitos processos cujos estados nem sempre são proveitosos em um *checkpoint*. Também, a alocação de objetos não é usualmente gerenciada da mesma forma empregada com processos numa linguagem imperativa; neste caso adiciona memória desnecessária para o *checkpoint*.

Estes argumentos sugerem que um enfoque em nível de linguagem dirigido pelo usuário pode ser apropriado para programas Java. Mas o *checkpointing* realizado em nível de linguagem aumenta o tamanho do programa fonte com um código para registrar os estados dos objetos do programa. Para promover segurança de funcionamento, esse código de *checkpointing* deveria ser introduzido sistematicamente, e interferir o mínimo possível no comportamento padrão do programa. *Checkpointing* incremental pode ser implementado associando uma variável de instância (*flag*) para cada objeto, indicando se o objeto foi modificado desde o último *checkpoint* prévio [3,5].

A atividade de *checkpointing* numa abordagem orientada a objetos visa salvar o estado dos objetos do programa em execução. Essa atividade pode ser implementada utilizando o mecanismo de serialização de objetos em Java. A recuperação baseia-se nos objetos serializados e o reinício da execução da aplicação. A partir da seqüência de *bytes* (objetos serializados no *checkpoint* prévio livres de erros), os objetos podem depois ser recriados por meio da de-serialização dos objetos. Em Java, serialização é implementada usando reflexão em tempo de execução. Reflexão é usada para determinar a estrutura estática de cada objeto (seu tipo, nome do campo, etc.), e para ter acesso aos valores de campos registrados [4].

Com base nas considerações expostas, o objetivo central do artigo é descrever as características e aspectos de projeto de uma biblioteca que oferece métodos de *checkpointing* e recuperação e que foi denominada *Libcjp – Library of checkpoints in Java programs*.

Dos trabalhos investigados, o que despertou maior interesse, e pode ser considerado como referência fundamental à biblioteca desenvolvida neste trabalho, é a *libckpt* de Plank *et al*. [2]. Porém, a presente implementação é bastante diferente, visto que explora o modelo de objetos. Os trabalhos de Lawall e Muller [4] e de Killijian *et al* [5] serviram de fonte de conhecimento sobre *checkpointing* orientado a objetos em Java, embora apresente soluções distintas para a atividade de *checkpointing* e recuperação. Assim, neste artigo são apresentados os conceitos básicos de persistência e serialização, as características da biblioteca, alguns aspectos de projeto e implementação, exemplo de uso e resultados preliminares obtidos nos testes realizados.

2. Persistência e serialização

A persistência, em linguagens de programação orientadas a objetos, é a habilidade dos objetos existirem além do tempo de vida do programa, no qual eles foram criados. O tempo de vida de um objeto inicia quando ele é criado pelo operador *new*, e subsiste até ser destruído pelo *garbage collector* da JVM - *Java Virtual Machine* [6]. Em Java, essa habilidade de persistência pode ser conseguida através da serialização de objetos.

Geralmente, a persistência é implementada para preservar o estado de um objeto. Neste contexto, preservar o estado significa converter o objeto para uma seqüência de *bytes* e armazená-los em um meio, que prolongue sua vida útil. Um objeto persistente pode ser armazenado em um arquivo para posterior uso ou ser transmitido pela rede para outra máquina [7]. O modelo de memória estável escolhido depende das hipóteses de falhas consideradas e deve ser adequadamente implementado.

A persistência consiste em armazenar o estado de um objeto, ou conjunto de objetos. Por exemplo, o programa P1 armazena em disco os objetos *obj1* e *obj2*; assim que houver necessidade, em uma futura execução, é possível restaurar o estado de execução de P1, utilizando os dados persistentes de *obj1* e *obj2* [7].

O mecanismo de serialização de objeto em Java permite capturar o estado de um ou mais objetos e representá-los como uma seqüência de *bytes* em um formato independente de plataforma [8, 9]. A partir desta seqüência de *bytes*, os objetos podem depois ser recriados em tempo de execução no mesmo ou em qualquer outro sistema Java. O processo de capturar o estado de um objeto é denominado de serialização de objeto (*serializing*), e o processo de recriar o objeto a partir do estado capturado é denominado de-serialização do objeto (*deserializing*). Quando um objeto é de-serializado, um novo objeto é criado e seu estado é estabelecido pelo estado representado pela seqüência de *bytes*; efetivamente é criada uma cópia do objeto serializado, ao invés de sobrescrever o estado de um objeto existente [10].

3. A biblioteca proposta: características e funcionalidades

A biblioteca *Libcjp – Library of checkpoints in Java programs* foi projetada com o propósito de trabalhar com aplicações computacionais escritas na linguagem de programação Java. *Libcjp* é uma biblioteca desenvolvida para ser utilizada em nível de usuário, projetada para uso em situações onde se deseja minimizar o tempo de execução total de uma aplicação, na presença de falhas. A biblioteca proposta facilita o estabelecimento de *checkpoints* e a retomada do processamento pós-falhas (ou recuperação) de aplicações orientadas a objetos. Foi desenvolvida com funcionalidades para capturar o estado de um ou mais objetos de uma aplicação em execução, e representá-los no meio de armazenamento (memória secundária) para posterior recuperação, após a ocorrência de falhas. Esta representação dos estados dos objetos da aplicação no disco (usado como opção de armazenamento, por simplicidade, apesar de formalmente não atender aos requisitos do meio estável) foi alcançada utilizando os mecanismos de persistência e serialização presentes na linguagem Java, através da API de serialização Java [8]. A retomada da execução da aplicação ocorre restaurando os estados dos objetos, a partir dos arquivos de *checkpoint* salvos.

Sabe-se que em um ambiente livre de falhas, o uso de mecanismos de salvamento intermediário de estados da computação acarretará queda de desempenho - se comparado à execução sem esta operação adicional. Entretanto, no momento em que ocorre falha, o reinício integral da aplicação pode ser bastante oneroso. A adoção cuidadosa de procedimentos - tais como o uso de técnicas otimizadas e a escolha adequada da freqüência na

execução de salvamentos, planejada de acordo com a probabilidade de falhas - deve trazer um resultado proveitoso no cômputo do desempenho. O projeto da biblioteca levou em conta estas questões, permitindo ao programador estabelecer adequadamente estes parâmetros.

Portanto, alguns benefícios fornecidos pelo enfoque proposto para a biblioteca *Libcjp* podem ser observados: (i) fazendo o uso da biblioteca, consegue-se prover maior disponibilidade para muitas aplicações computacionais orientadas a objetos escritas em Java, aprimorando a retomada da execução, diante da ocorrência de falhas; (ii) diferentes formas de utilização da biblioteca poderão ser feitas pelo programador. Em alguns casos a biblioteca adiciona pequena sobrecarga de processamento para salvamento dos estados dos objetos sobre o tempo de execução da aplicação; (iii) uma das principais vantagens da linguagem Java é a independência de plataforma: um programa Java pode ser executado em qualquer plataforma que possua uma JVM e deverá apresentar o mesmo comportamento em cada uma destas plataformas. A *Libcjp* foi preparada para trabalhar não apenas na plataforma a qual foi desenvolvida (sistema operacional *Windows*), mas poderá ser utilizada também no sistema operacional *Linux*; (iv) os *checkpoints* são independentes de plataforma, ou seja, uma dada aplicação Java poderá ser reiniciada em uma outra JVM de outra plataforma, sem levar em consideração o ambiente de criação dos *checkpoints*.

Para prover os benefícios recém mencionados, a biblioteca agrega alguns custos para sua utilização: (i) *checkpointing* com *Libcjp* não é completamente transparente ao programador da aplicação, é necessário algum esforço de programação para o seu uso. Diz-se que *checkpointing* é transparente quando nenhuma modificação precisa ser feita no código-fonte da aplicação. Tal esforço, pequeno em princípio, dependerá das funcionalidades escolhidas e do perfil de aplicação desenvolvida; (ii) o tamanho do código-fonte da aplicação não deverá aumentar significativamente, sendo acrescidas somente algumas linhas de código para especificar à biblioteca os objetos que deverão estar presentes no arquivo de *checkpoint* e para efetuar a recuperação.

As principais funcionalidades propostas e implementadas são descritas a seguir.

Mecanismo de *checkpointing* tradicional: é o método mais direto para estabelecer um arquivo de *checkpoint*. A execução da aplicação é suspensa, enquanto os estados dos objetos da aplicação são serializados e salvos no arquivo de *checkpoint*. Este mecanismo também é denominado de *checkpointing* seqüencial, porque transferências da memória (*stream* contendo os estados dos objetos serializados) para o meio de armazenamento (arquivo de *checkpoint*) são intercaladas com a execução da aplicação.

Mecanismo de *checkpointing* incremental: embora a idéia seja semelhante ao mecanismo tradicional, através deste mecanismo, somente os estados dos objetos modificados desde o *checkpoint* prévio são salvos no arquivo de *checkpoint*. A base do *checkpointing* incremental é não salvar repetidamente os objetos que não foram modificados.

Em geral, o tamanho de um *checkpoint* **não incremental** não varia significantemente ou cresce lentamente com o passar do tempo. Além disso, apenas o arquivo mais recente de *checkpoint* necessita ser mantido para recuperação; os arquivos mais antigos podem ser apagados. Em contraste, quando se emprega o mecanismo de *checkpointing* incremental, arquivos de *checkpoint* antigos não podem ser apagados, porque os estados dos objetos da aplicação estão espalhados em muitos arquivos de *checkpoint*. Os estados inalterados dos objetos são restabelecidos a partir de *checkpoints* prévios. Salvando apenas os objetos cujos estados foram modificados, reduz o tamanho de cada arquivo de *checkpoint*.

***Checkpointing* programado**: esta técnica pode ser empregada em conjunto com ambos

mecanismos de *checkpointing*: tradicional e incremental. O programador especifica pontos no código da aplicação onde é vantajoso ou necessário o *checkpointing*. O arquivo de *checkpoint* pode ser estabelecido de duas formas: (i) **executado sempre**: toda vez que a linha de execução da aplicação passar por uma chamada ao método que ativa o mecanismo, o *checkpoint* será estabelecido; (ii) **inibido por controle de tempo**: o *checkpointing* apenas ocorrerá depois de decorrido um período mínimo de tempo, desde o último estabelecimento.

Mecanismo de recuperação: a técnica tradicional para recuperação baseia-se na recomposição de um objeto ou vários objetos a partir dos estados de-serializados de um arquivo de *checkpoint* para um estado operacional normal. Após a ocorrência desta atividade, a execução da aplicação é retomada.

Na atividade de recuperação, quando os arquivos de *checkpoint* forem estabelecidos através do mecanismo de *checkpointing* incremental, haverá a necessidade de ser realizada a fusão dos arquivos velhos de *checkpoint*. A atividade de fusão consiste em buscar nos arquivos de *checkpoint* os estados mais atuais dos objetos da aplicação. Isto é feito com a leitura de cada arquivo de *checkpoint* estabelecido previamente durante a execução da aplicação sem a ocorrência de falhas, até que se obtenha os estados mais atuais dos objetos.

Controle do número de arquivos de *checkpoint*: esta funcionalidade cuida para manter armazenada apenas a quantidade de arquivos estipulada.

Registros de informações (tempos): esta funcionalidade tem por objetivo propiciar a obtenção de informações de tempos referentes ao estabelecimento dos arquivos de *checkpoint* e a execução da aplicação.

4. Projeto da biblioteca e exemplo de uso

Inicialmente as classes da biblioteca *Libcjp* foram modeladas usando UML – *Unified Modeling Language*; após, foram implementadas utilizando a linguagem orientada a objetos Java (pacote *jdk1.3.1_01*). A Figura 1 mostra um diagrama contendo um modelo simplificado de classes da biblioteca *Libcjp*.

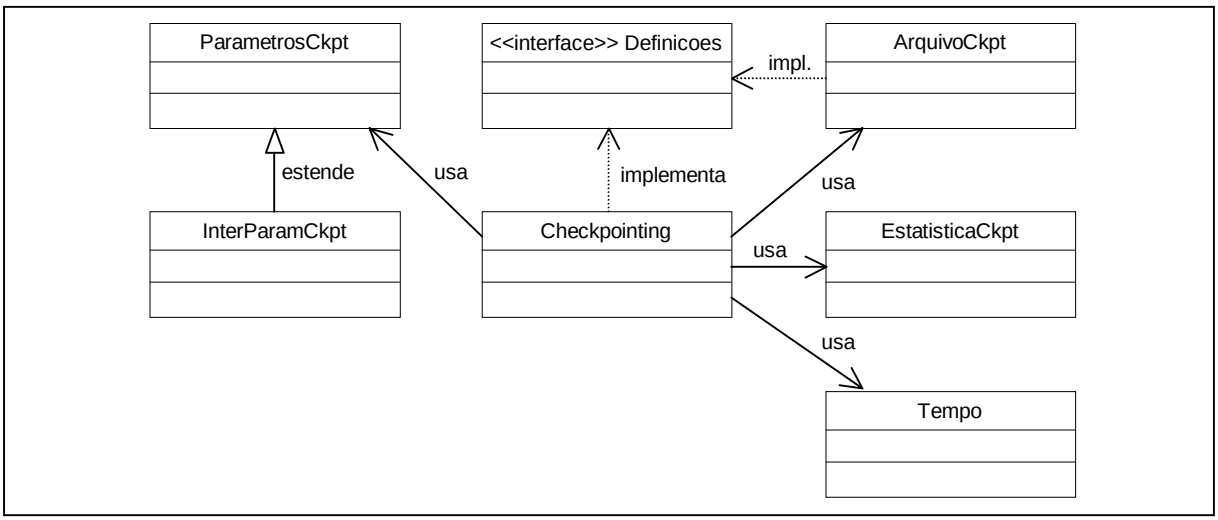


Figura 1 – Modelo simplificado de classes da biblioteca *Libcjp*.

Nesse diagrama, a classe **`Checkpointing`** é a principal, pois nela estão implementados os mecanismos para prover estabelecimento de *checkpoints* e recuperação, e também todo o controle operacional da biblioteca. As classes **`ArquivoCkpt`** e **`Tempo`** dão apoio funcional a

classe **Checkpointing**, no que se refere à manipulação das características dos arquivos de *checkpoint* e controle de tempo para o estabelecimento dos *checkpoints*. A classe **EstatisticaCkpt** faz o registro dos tempos gastos para o estabelecimento de cada *checkpoint* e o tempo total de execução da aplicação. A interface **Definicoes** contém as constantes utilizadas pelas classes **Checkpointing** e **ArquivoCkpt**. As classes restantes **ParametrosCkpt** e **InterParamCkpts** são utilizadas como apoio à biblioteca, provendo um arquivo de parâmetros denominado de **paramckpt.dat**. Estes parâmetros têm por finalidade configurar a biblioteca para alguns tipos diferentes de utilização. Os possíveis parâmetros do arquivo **paramckpt.dat**, seus possíveis valores e seus valores padrão são listados na Figura 2, e explicados a seguir.

Valores padrão	**possíveis valores**
intervalo=1000	<**1**... (número correspondente ao tempo)>
tipo_intervalo=L	<**H** (**H**ora), **M** (**M**inuto), **S** (**S**egundo), **L** (mi**L**issegundo), **V** (sem inter**V**alo)>
num_max_arquivos=0	<**0**,**1** (sem efeito), **2**... (número máximo de arquivos de *checkpoint* incremental)>
incremental=N	<**S**/**N**>
diretorio=.	<. (diretório desejado)>

Figura 2 – Valores possíveis dos parâmetros do arquivo "**paramckpt.dat**".

O parâmetro "**intervalo**=<*n*>" estabelece um valor correspondente a um período mínimo de tempo que deve decorrer entre um *checkpointing* e outro. A definição de unidade empregada é fornecida através do parâmetro "**tipo_intervalo**=<*H/M/S/L/V*>". O número máximo de arquivos de *checkpoint* para *n* é dado por "**num_max_arquivos**=<*n*>"; este parâmetro pode ser usado tanto para o *checkpointing* incremental ou *checkpointing* tradicional. Valores de *n*>1 permitem ao usuário impor um equilíbrio entre o número de arquivos de *checkpoint* e a quantidade de espaço utilizado no meio de armazenamento. O parâmetro <**incremental**=*S/N*> ativa ou desativa o mecanismo de *checkpointing* incremental. O parâmetro "**diretorio**=<**.**>" especifica o diretório no qual os arquivos de *checkpoint* serão gravados.

5. Avaliação e testes preliminares

Foram desenvolvidas algumas aplicações para testar e avaliar o desempenho funcional da *Libcjp*. As empregadas nos testes preliminares são programas científicos típicos escritos em Java: (i) MAT efetua a multiplicação de duas matrizes 615 x 615 de elementos inteiros; (ii) TDC calcula a transformada discreta do cosseno. Esta transformada é utilizada no processo de compressão do JPEG. Foi utilizada uma matriz 512 x 512, representando as informações de cada *pixel* de uma imagem, em tons de cinza; (iii) SHELL é o método de ordenação – *ShellSort*. Para ordenação, foi utilizado um vetor de 100.000 elementos inteiros, em ordem inversa; (iv) HEAP corresponde ao método de ordenação *HeapSort*. Também foi utilizado como dado inicial um vetor de 100.000 elementos inteiros em ordem inversa; (v) GAUSPIV efetua o método da eliminação gaussiana com pivoteamento. Foi submetido à aplicação para resolução um sistema linear 30 x 30.

Como ambiente de teste para avaliação da *Libcjp*, foi utilizado um microcomputador isolado, com processador *Duron* 850 Mhz, 128 *Mbytes* de RAM e sistema operacional *Linux Mandrake release* 8.0 com *kernel* 2.4.3-20mdk. Cada aplicação foi executada 100 vezes.

A Figura 3 demonstra a utilização da biblioteca *Libcjp* em uma aplicação que efetua a multiplicação de duas matrizes. As modificações e inserções de novas linhas de código para

ativar os mecanismos de *checkpointing* e recuperação (em negrito), são observadas na segunda coluna da figura. As linhas 8, 9 e 10 são responsáveis por ativar o mecanismo de recuperação, as linhas 22, 23 e 24 são responsáveis por ativar o mecanismo de *checkpointing*.

```
public Integer[][] multMatriz(
          Integer m1[][], Integer m2[][])
{
  Integer m3[][] = new Integer[615][615];

  for (int i=0  ; i<615 ; i++)
  {
    for (int j=0 ; j<615 ; j++)
    {
      int s=0;
      for (int k=0 ; k<615 ; k++)
        s+=(m1[i][k].intValue()*
        m2[k][j].intValue());
      m3[i][j] = new Integer(s);
    }
  }
  return m3;
}
```

(a) programa sem usar a biblioteca

```
 1 public Integer[][] multMatriz(
 2 Integer m1[][], Integer m2[][], String args[])
 3 {
 4   Integer m3[][] = new Integer[615][615];
 5   int ini=0;
 6   if (c.verificaRecuperacao(args))
 7   {
 8     c.recover_oockpt();
 9     m3=(Integer[][])c.recover();
10     ini=(((Integer)c.recover()).intValue());
11   }
12   for (int i=ini  ; i<615 ; i++)
13   {
14     for (int j=0 ; j<615 ; j++)
15     {
16       int s=0;
17       for (int k=0 ; k<0615 ; k++)
18         s+=(m1[i][k].intValue()*
19         m2[k][j].intValue());
20       m3[i][j] = new Integer(s);
21     }
22     c.save(m3);
23     c.save(new Integer(i+1));
24     c.checkpoint_here();
25   }
26   return m3;
27 }
```

(b) programa fazendo o uso da biblioteca

Figura 3 – Exemplo de utilização da biblioteca *Libcjp*.

A Tabela 1 e a Tabela 2 mostram os resultados parciais obtidos nos testes efetuados com o uso da biblioteca, através de experimentos de *checkpointing*. Estes resultados são preliminares; novas análises serão realizadas sobre os dados para se obter resultados mais completos e um estudo estatístico mais elaborado. Na Tabela 1, estão expressos: os tempos médios de execução de cada aplicação sem e com *checkpointing* (em segundos); o desvio padrão (σx); o intervalo de confiança de 95% (IC); a porcentagem de *overhead* (sobrecarga) de processamento sobre a aplicação causada pelo *checkpointing*; o intervalo de tempo entre cada dois *checkpoints* (Δ); e o número de arquivos de *checkpoint* (Nº arq).

Tabela 1 – Resultado dos experimentos de *checkpointing* para cada aplicação.

Aplicação	Exec_s/*ckpt*	Exec_c/*ckpt*	σx	IC 95%	% *overhead*	Δ (seg)	Nº arq.
MAT	106,5350	126,083	12,7712	5,0063	18,3489	10	10
TDC	33,7787	57,6452	1,2347	0,4840	70,6552	0,5	5
SHELL	368,9478	390,6071	10,1003	3,9593	5,8706	30	12
HEAP	598,5755	636,1563	1,7499	0,6860	6,2784	30	20
GAUSPIV	21,0798	21,811	0,3100	0,1215	3,4687	2	7

A Tabela 2 mostra em cada uma das linhas: o tempo médio do estabelecimento de um *checkpoint* para cada aplicação, utilizando o mecanismo de *checkpointing* tradicional; o desvio padrão (σx) para estes dados; o intervalo de confiança de 95% (IC); e o tamanho médio dos arquivos de *checkpoint* expressos em *Kbytes*. Os valores dos tempos apresentam variações, de uma aplicação para outra, em função do volume de informações salvas em cada *checkpoint*.

Tabela 2 – Resultado das medidas da atividade de cada *checkpointing*.

Aplicação	Tempo *ckpt*. (seg)	σx	IC 95%	Tam. *ckpt*. (*Kbytes*)
MAT	1,9900	1,2416	0,1539	2034,8994
TDC	4,8398	0,9546	0,1673	5352,1743
SHELL	1,6980	0,1480	0,01675	976,9092
HEAP	1,4548	0,06119	0,005363	976,8701
GAUSPIV	0,03504	0,04068	0,005638	13,8711

6. Conclusões

Neste artigo, foram abordados a motivação e os princípios básicos relacionados ao estabelecimento de *checkpoints* em aplicações orientadas a objetos. Em seguida, foram apresentadas as características e funcionalidades da biblioteca *Libcjp*, proposta para atividades de recuperação de objetos. Esta biblioteca não isenta o programador de conhecer os mecanismos básicos envolvidos na execução das atividades de recuperação - ele precisa configurar alguns parâmetros e introduzir algumas chamadas. A biblioteca simplifica o restante da programação por disponibilizar os mecanismos necessários. Os números mostrados através da avaliação preliminar têm por objetivo dar uma idéia da repercussão da biblioteca em aplicações com diferentes perfis. Os valores, entretanto, podem ser bem diferentes se os parâmetros de uso (freqüência de chamada, volume de dados, etc...) variarem através da parametrização adequada ou de alteração nas características da aplicação.

Serão realizados, em uma próxima fase, novos experimentos de *checkpointing* utilizando a biblioteca, e uma análise detalhada dos resultados de desempenho obtidos, visando observar tendências e obter conclusões adicionais.

Referências

[1] N. Singhal; N. Shivaratri. Advanced Concepts in Operation Systems Distributed, Database and Multiprocessor Operation Systems. New York: McGraw-Hill, 1994.

[2] J. S. Plank et al. Libckpt: Transparent Checkpointing under Unix. In: Usenix Technical Conference, New Orleans, 1995. Proceedings. [s.n], January 1995.

[3] M. Kasbekar; et al. Issues in the Design of a Reflective Library for Checkpointing C++ Objects. In: 18th IEEE Symposium on Reliable Distributed Systems, Lausanne, Oct. 1999.

[4] J. L. Lawall; G. Muller. Efficient Incremental Checkpointing of Java Programs. In: Intl. Conference on Dependable Systems and Networks (DSN 2000), New York, June 2000. pp. 61-70.

[5] M. O. Killijian; J. C. Fabre; J. C. Ruiz-Garcia. Using compile-time reflection for object checkpointing. LAAS, February 1999. (Technical Report-99049).

[6] J. Mathis. Persistence and Java Serialization. In: Java 1.1: Unleashed. Indianapolis: Sams.Net, 3ed., 1997.

[7] S. C. Bertagnolli. Integrando Aspectos de Distribuição e de Tolerância a Falhas no Ambiente Java Reflexivo (Jreflex). Porto Alegre: PPGC da UFRGS, Julho 2000 (TI-913).

[8] Sun Microsystems. Java Object Serialization Specification – JDK 1.2 Beta 3. Disponível em URL: ftp://ftp.javasoft.com/docs/jdk1.2/serial-spec-JDK1.2.ps. Sun Microsystems, February, 1998.

[9] K. Arnold; G. James. The Java Programming Language. [S.l]:Addison-Wesley, 2. ed.,1997.

[10]E. K. Hansen. Checkpointing Java Programs on Standard Java Implementations using Program Transformation. Master's Thesis. Department of Computer Science, University of Copenhagen. Nov. 1999. Disponível por www em http://hjem.get2net.dk/esben_krag_hansen/jcp/index.html.

Checkpointing e Recuperação de Falhas em Sistemas Distribuídos Particionáveis

Tiemi C. Sakata Islene C. Garcia Luiz E. Buzato

Universidade Estadual de Campinas
Caixa Postal 6176
13083-970 Campinas, SP, Brasil
Tel: +55 19 3788 5876 Fax: +55 19 3788 5847
{tiemi, islene, buzato}@ic.unicamp.br

Resumo

Os protocolos existentes para *checkpointing* e recuperação por retrocesso normalmente consideram que não haverá partições na rede de comunicação e que a recuperação por retrocesso só será iniciada quando todos os processos que falharam voltarem à operação normal. Estas restrições podem tornar inviável a utilização de *checkpointing* e recuperação por retrocesso em aplicações que operam sobre ambientes móveis. Neste resumo comentaremos como essas restrições podem ser contornadas para permitir o avanço seguro da aplicação mesmo na presença de falhas e/ou partições da rede.

1 Introdução

Um sistema distribuído assíncrono é formado por um conjunto de componentes que se comunicam exclusivamente através da troca de mensagens. O meio pelo qual estas mensagens serão transmitidas pode variar de um sistema para outro. Em redes fixas, a ligação entre os componentes é feita através de cabos de comunicação. Em redes ad hoc os componentes móveis se comunicam diretamente através de conexão sem fio.

Um sistema distribuído está dividido em partições se pelo menos um par de componentes não consegue se comunicar. Cada partição é formada pelo subconjunto de componentes que ainda conseguem estabelecer comunicação. Nas redes fixas, partições ocorrem apenas quando há falhas de componentes ou de canais de comunicação. Nas redes ad hoc, as partições podem ocorrer por falhas nos componentes ou por incapacidade de comunicação devido ao distanciamento. Desta forma, podemos considerar que partições são relativamente raras e transientes em redes fixas, mas podem ser freqüentes na presença de componentes móveis.

Após a ocorrência de uma falha, é necessário que o sistema (r)estabeleça um estado global consistente, ou seja, um estado global que poderia ter sido analisado por um observador onisciente externo [3]. A recuperação pode ser feita por avanço (dependente da aplicação) ou por retrocesso de estado (independe da aplicação) [5, cap. 7].

Um *checkpoint* é um estado de interesse de um componente de uma computação distribuída escolhido para ser armazenado em memória estável e a atividade de *checkpointing* corresponde

à seleção de *checkpoints* feita por todos os componentes. Vários protocolos foram propostos na literatura para *checkpointing* e recuperação por retrocesso em redes fixas [3, 4, 6]. Mais recentemente surgiram trabalhos para redes móveis estruturadas [1, 2], mas não conhecemos nenhum trabalho sobre recuperação por retrocesso em redes ad hoc.

Em um sistema distribuído assíncrono, devido ao atraso arbitrário das mensagens, é impossível distinguir uma partição real de uma virtual [7]. Além disso, permitir que o sistema progrida na presença de partições pode levar a inconsistências. Provavelmente esta complexidade e a baixa ocorrência de falhas que resultam em partição em redes fixas induziram os protocolos existentes para recuperação por retrocesso a ignorar a possibilidade de partições.

Propomos a extensão dos protocolos existentes para *checkpointing* e recuperação de falhas de maneira a incorporar a possibilidade de (i) divisão do sistema em uma ou mais partições, (ii) avanço independente de cada partição, (iii) recuperação parcial em caso de falha em um dos componentes de uma partição e (iv) agrupamento ou reconfiguração das partições. No momento da reunificação das partições, pode ser detectada uma inconsistência que deverá ser corrigida através de recuperação por retrocesso ou por avanço.

2 Redes Ad Hoc

Uma aplicação distribuída em uma rede ad hoc é composta por um conjunto de nós móveis ($MH_1, \ldots, MH_n$), onde os nós móveis podem trocar informações entre si (desde que cada nó esteja dentro da área de alcance do outro) como ilustra a Figura 1 (a). Os nós das redes ad hoc podem se mover de forma arbitrária de modo que a topologia da rede muda freqüentemente, dificultando o roteamento e a localização dos nós móveis. O distanciamento pode inclusive ocasionar a partição da rede, como ilustram as Figuras 1 (b, c, d, e, f).

Graças ao avanço tecnológico, esses computadores possuem memória cada vez mais rápida e com maior capacidade. Utilizaremos esta memória para o armazenamento de *checkpoints* e replicação destes *checkpoints* em nós vizinhos com o objetivo de tolerar falhas de perda, roubo ou danos físicos.

3 Checkpointing na Presença de Partições

Propomos a extensão dos protocolos existentes para *checkpointing* e recuperação de falhas de maneira a incorporar a possibilidade de:

(i) divisão do sistema em uma ou mais partições—devida à falha em componentes, canais de comunicação ou ao distanciamento dos computadores em redes móveis.

(ii) avanço independente de cada partição— pode não ser desejável esperar pela reunificação completa dos componentes para continuar a computação.

(iii) recuperação parcial restrita a uma partição—a recuperação deve ser feita com a colaboração de todos os componentes desta partição.

(iv) agrupamento ou reconfiguração das partições— pode ser detectada uma inconsistência que deverá ser corrigida através de recuperação por retrocesso ou por avanço.

Ilustraremos estes quatro passos com o auxílio do diagrama espaço-tempo apresentado na Figura 2, cujas partições são mostradas através do cenário da Figura 1.

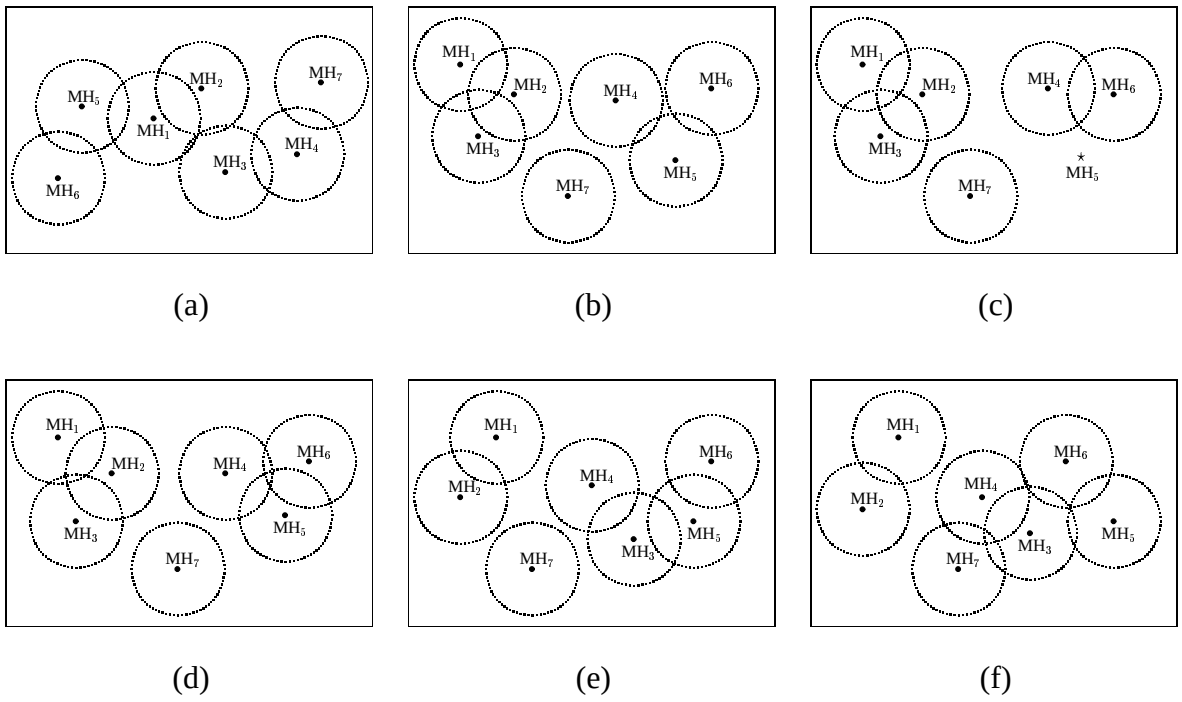


Figura 1: Rede ad hoc

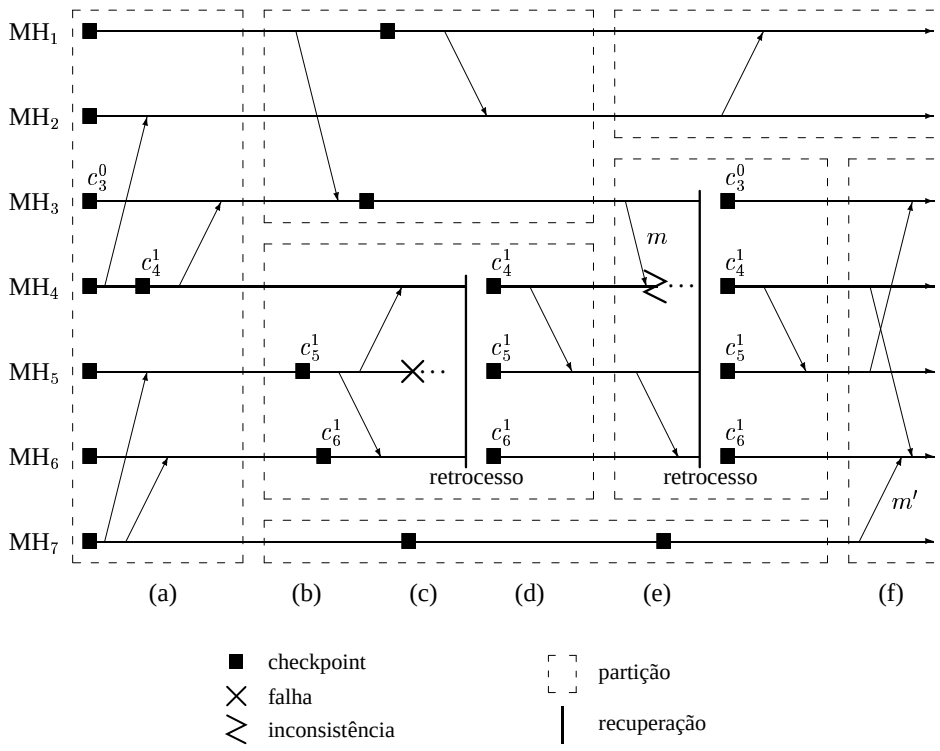


Figura 2: Reunificação de partições com recuperação por retrocesso

Imediatamente antes de começar a computação, todos os componentes (MH_1,...,MH_7) tiram um *checkpoint* inicial. A execução será descrita a seguir:

(a) todos conseguem se comunicar.

(b) divisão do sistema em três partições: (MH_1, MH_2, MH_3), (MH_4, MH_5, MH_6) e (MH_7).

(c) ocorre uma falha no componente MH_5.

(d) MH_5 volta a processar mas terá de retroceder a um *checkpoint* armazenado previamente. Todos os componentes da partição devem retroceder para um estado consistente e portanto MH_4, MH_5 e MH_6 retornam respectivamente para os *checkpoints*: c_4^1, c_5^1 e c_6^1.

(e) o sistema sofre uma reconfiguração formando três novas partições: (MH_1, MH_2), (MH_3, MH_4, MH_5, MH_6) e (MH_7). Na recepção da mensagem m, uma inconsistência é detectada. Na Figura 2, MH_3, MH_4, MH_5 e MH_6 devem retroceder para um estado consistente, ou seja, retornam respectivamente para os *checkpoints*: c_3^0, c_4^1, c_5^1 e c_6^1.

(f) o sistema sofre uma reconfiguração formando duas partições: (MH_1, MH_2) e (MH_3, MH_4, MH_5, MH_6, MH_7). Neste momento, MH_6 recebe uma mensagem m' de MH_7 e neste caso não ocorre inconsistência e a computação segue normalmente.

4 Conclusão

Um sistema distribuído pode ser particionado pela ocorrência de falhas ou por desconexão de um de seus componentes. Em uma rede ad hoc, o particionamento e a reunificação do sistema ocorre com freqüência pela mobilidade dos componentes. Este trabalho analisa as características desta rede e os pontos importantes que devemos considerar na extensão dos protocolos para *checkpointing* de rede fixa para as rede ad hoc.

Referências

[1] A. Acharya and B. R. Badrinath. Checkpointing Distributed Applications on Mobile Computers. In *International Conference on Parallel and Distributed Information Systems*, Sept. 1994.

[2] G. Cao and M. Singhal. Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems. *IEEE Trans. on Parallel and Distributed Systems*, 12(2):157–172, Feb. 2001.

[3] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computing Systems*, 3(1):63–75, Feb. 1985.

[4] E. N. Elnozahy, D. Johnson, and Y.M.Yang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-96-181, Carnegie Mellon University, 1996.

[5] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice.* Second, Revised Edition, Springer-Verlag, New York, 1990.

[6] D. Manivannan and M. Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. Technical Report OH 43210, Department of Computer and Information Science, The Ohio State University, 1997.

[7] A. Ricciardi, A. Schiper, and K. Birman. Understanding Partitions and the "No Partition" Assumption. In *Proceedings of the 4th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems*, pages 354–360, Lisboa, Portugal, 1993.

Uso de Redes de Autômatos Estocásticos para Modelar Mecanismos Tolerantes a Falhas*

Luciano A. Cassol, Avelino F. Zorzo, Paulo Fernandes
{lcassol,zorzo,paulof}@inf.pucrs.br

[1]FACIN - PUCRS - Av. Ipiranga 6681 - 90619-900 - Porto Alegre - RS

Resumo. O uso de mecanismos de tolerância a falhas tem sido cada vez mais freqüente no desenvolvimento de sistemas críticos. Com este aumento, a precisão da descrição de tais mecanismos é fundamental para que seu uso não gere situações inesperadas aos desenvolvedores dos sistemas, devido a má interpretação destes mecanismos. Desta forma, este artigo apresenta a descrição de interações multi-participantes confiáveis (DMI) usando redes de autômatos estocásticos (SAN), um formalismo para descrição de sistemas distribuídos onde interações entre participantes pode ser modelado como um autômato estocástico.

1. Introdução

Programas paralelos são compostos de diferentes atividades concorrentes, padrões de comunicação e sincronização entre essas atividades. Dessa forma, a programação paralela é considerada difícil, pois somando-se as preocupações normais de programação, o programador tem que trabalhar com uma complexidade ainda maior gerada pelas diversas *threads* do sistema, controlando sua criação e destruição, e controlando suas interações através de sincronização e comunicação [1].

Somado a este contexto, a proliferação de sistemas distribuídos tem aumentado consideravelmente nos últimos anos. Apesar da distribuição tornar o sistema mais robusto, através da replicação de seus componentes, algumas vezes o aumento na distribuição pode introduzir falhas indesejáveis. Desta forma, é vital que a programação distribuída seja feita de maneira o rganizada.

Uma forma de organizar aplicações distribuídas é através do uso de operações que envolvam mais de um participante, como ações separadas que coordenam as interações entre esses participantes.

Um mecanismo que inclui diversos participantes (objetos ou processos) executando um conjunto de atividades em conjunto é chamado de Interações Multi-participantes (*Multiparty Interaction* - MI). Em uma MI, diversos participantes "reúnem-se" para produzir um estado combinado, intermediário e temporário. Usam este estado para executar algumas atividades e assim deixam a interação e continuam sua execução normal [2]. Neste artigo utilizamos um mecanismo para interações entre diversos participantes que provê recursos para tratamento de

*Este projeto tem apoio da CAPES, CNPq e FAPERGS.

exceções concorrentes e consistência de estado no término da interação. Este mecanismo é denominado Interações Multi-participantes Confiáveis (DMI - *Dependable Multiparty Interaction*) [3]. DMIs tem sido aplicadas na implementação de diversos sistemas críticos, porém a verificação de tais sistemas é na maioria das vezes realizada de maneira empírica.

Uma forma de verificar se as interações entre diversos participantes possuem as propriedades e comportamentos esperados é através da formalização das características e comportamentos que as interações entre diversos participantes têm. Para formalizar uma especificação de um sistema, é necessário uma linguagem de especificação formal.

Linguagens de especificação formal são utilizadas para fornecer uma descrição clara e precisa de sistemas, para testar e provar matematicamente que tais sistemas possuem as propriedades e comportamentos esperados, permitindo assim uma única interpretação de seu funcionamento. Esta interpretação única nem sempre é fornecida por descrições feitas em linguagem natural, onde diferentes interpretações podem gerar diferentes resultados. Para prover esta interpretação, as linguagens de especificação formal fornecem um modelo matemático para descrição de sistemas, a partir do qual podem ser realizadas verificações, a fim de provar que o sistema possui determinadas propriedades e comportamentos.

A descrição sintática e semântica das linguagens é essencial para auxiliar no desenvolvimento e programação de sistemas concorrentes com qualidade. A descrição sintática descreve a forma correta que os programas devem ser escritos, enquanto uma descrição semântica descreve o significado que é anexado as várias construções sintáticas.

A definição de sistemas complexos dificilmente pode ser abordada cientificamente sem que uma descrição formal seja empregada. No entanto, as qualidades de uma formalização de um sistema não se limitam a sua modelagem de forma inequívoca. A utilização de um formalismo torna possível a obtenção de conclusões sobre a viabilidade do sistema descrito através de índices de confiabilidade e desempenho [4, 5]. As observações sobre a viabilidade do sistema podem ser divididas em:

- informações que estimam confiabilidade; e
- informações que estimam o desempenho.

O formalismo de Redes de Autômatos Estocásticos (*Stochastic Automata Networks*) [6] é de grande interesse na modelagem de sistemas onde o paralelismo e sincronização são primitivas básicas de relacionamento. Esse formalismo baseia-se na descrição de sistemas complexos como um conjunto de entidades autônomas que interagem ocasionalmente.

Este artigo tem como objetivo descrever a utilização das DMIs através de Redes de Autômatos Estocásticos (SAN).

2. Redes de Autômatos Estocásticos (SAN)

Redes de autômatos estocásticos baseiam-se no formalismo matemático chamado cadeias de *Markov* [7]. Esse formalismo descreve o funcionamento do sistema como um conjunto de estados que se alteram segundo taxas definidas por leis exponenciais (autômatos estocásticos). Uma

vez modelado o sistema, a resolução do sistema consiste na solução de sistemas de equações lineares. O tamanho deste sistema é igual ao número de estados da Cadeia de Markov [7].

Redes de autômatos estocásticos são um formalismo para a modelagem de sistemas com grandes espaços de estados. Ao contrário das Cadeias de Markov que descrevem o sistema como um único autômato, as Redes de Autômatos Estocásticos baseiam-se na descrição de um sistema como um conjunto de subsistemas, em que cada subsistema é modelado como um autômato estocástico. A interação entre os subsistemas é descrita através das regras estabelecidas entre os estados internos de cada autômato. Dessa forma, o formalismo de redes de autômatos estocásticos é importante para a modelagem de sistemas distribuídos, nos quais as unidades de processamento não interagem entre si a todo instante [8].

Um autômato pode ser descrito como um conjunto de estados e um conjunto de transições entre esses estados. Essas transições são divididas em transições locais e transições sincronizadas. Um evento local é associado a uma única transição local e um evento sincronizante é associado a um conjunto de duas ou mais transições sincronizadas cada uma delas em um autômato.

O estado global de um sistema é definido como a combinação de todos os estados locais de cada autômato do sistema. Alterando o estado local de um autômato, o estado global da rede altera-se também. A mudança no estado global numa rede de autômatos estocásticos pode ser conseqüência da ocorrência de um evento local, ou de um evento sincronizante.

Um estudo mais aprofundado sobre SAN foi realizado em [9] e uma referência mais completa pode ser encontrada em [6, 8].

3. Estudo de Caso: formalização de DMI em SAN

De forma a modelar uma DMI em SAN, primeiramente devemos estabelecer um conjunto de regras a serem descritas de maneira precisa e inequívoca. Este conjunto de regras já foi previamente especificado em outro formalismo [10]. Além disto, um framework foi implementado a partir deste conjunto de regras e usado para o desenvolvimento de diversas aplicações críticas [2, 11, 12]. O conjunto de regras que estaremos mostrando neste artigo é o seguinte:

- a DMI é composta por um conjunto de papéis (*roles*) que são executados por participantes (*players*);
- os participantes ativam um papel para executar as instruções que compõem o papel;
- a DMI é iniciada somente quando todos os papéis da DMI tiverem sido ativados, e uma pré-condição (*guard*) estiver habilitada;
- a DMI somente é finalizada quando todos os participantes tiverem terminado de executar seus papéis, a pós-condição (*assertion*) for validada, e nenhuma exceção tiver sido levantada;
- exceções podem ser levantadas durante a execução da DMI. Se a exceção é levantada, então todos os papéis que não levantarem uma exceção são interrompidos, e um algoritmo de resolução é executado;
- se existe um tratador para a exceção que foi escolhida pelo algoritmo de resolução, então esse tratador é ativado por todos os papéis;

- se não existe um tratador para a exceção que foi escolhida pelo algoritmo de resolução, então a exceção é sinalizada[1] para quem invocou os papéis.

Para demonstrar como modelar sistemas que são implementados utilizando DMIs, adotamos, neste artigo, uma DMI utilizada na implementação de um sistema de controle para uma célula de produção [13]. Esta DMI será descrita em SAN.

A DMI escolhida foi a `CarregaGuindaste1` (veja Figura 1). Esta DMI é responsável por executar a leitura do código de barras existente em um bloco de metal que se encontra em uma esteira alimentadora, e posteriormente fazer com que um guindaste retire este bloco de metal da esteira. Para executar estas tarefas, a DMI `CarregaGuindaste1` é modelada com três papéis: `EsteiraAlimentadora`, `LeitorCodigoBarras` e `Guindaste1`. Estes papéis são ativados (executados) pelos controladores de cada um dos dispositivos correspondentes. Veja na figura as atividades executadas em cada um dos papéis. As setas pontilhadas entrando ou saindo da DMI representam os acessos aos objetos externos à DMI (estes objetos representam os dispositivos físicos na célula de produção).

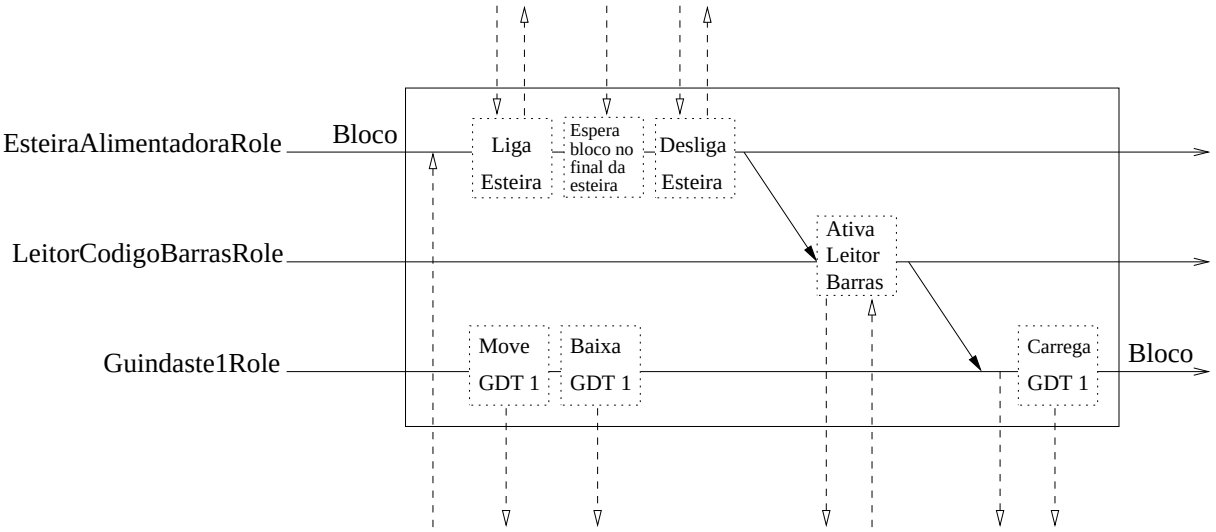


Figura 1: Exemplo da DMI CarregaGuindaste [13]

A Figura 2 representa parte do modelo gráfico da especificação da DMI `CarregaGuindaste1` em SAN. Nesta figura, um conjunto de autômatos é utilizado para representar uma `DMI`: cada um dos participantes (`PLAYER`$_i$), a pré-condição (`GUARD`), a pós-condição (`ASSERT`), o tratador de exceção (`HANDLER`), e o resultado e estado de cada um dos papéis (`ROLE`$_i$). A representação textual, para este modelo gráfico, que usa a ferramenta PEPS[2] [14] para avaliação de desempenho e confiabilidade, é apresentado na Figura 3.

O funcionamento da DMI, ou seja a troca de estados nos autômatos, durante a execução normal de uma DMI é feita conforme descrito a seguir. Um participante antes de querer executar

[1] O termo levantar exceção refere-se a uma exceção no contexto onde ela aconteceu, enquanto o termo sinalizar exceção refere-se a propagação da exceção para o nível superior do serviço que está sendo executado.

[2] PEPS -*Performance Evaluation of Parallel Systems, software* que resolve numericamente Cadeias de Markov com grande espaço de estados. O modelo em Redes de Autômatos Estocásticos é baseado em Cadeias de Markov.

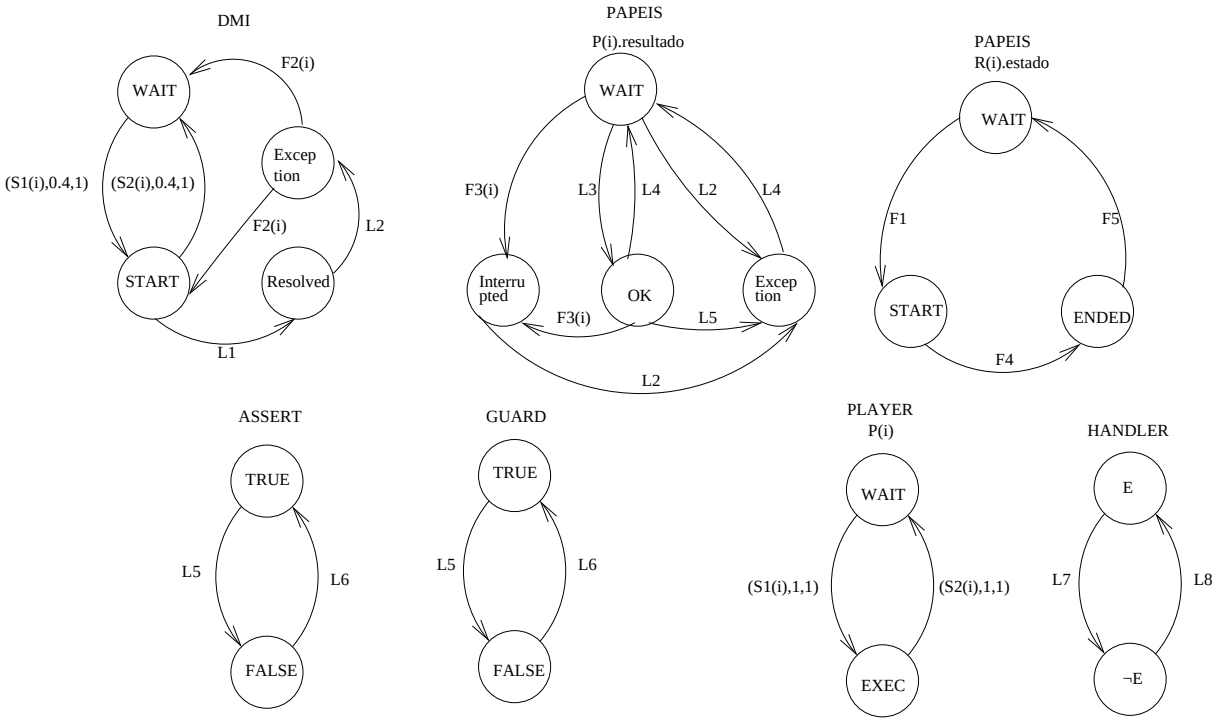


Figura 2: Modelo Gráfico da Especificação da DMI `CarregaGuindaste` em SAN

um papel em uma DMI, encontra-se no estado `WAIT`. A partir do momento que o mesmo deseja executar um papel ele passa do estado `WAIT` e habilita a transição sincronizada correspondente, ou seja S_i. Para que uma DMI esteja habilitada, ou seja passe do estado `WAIT` para o estado `START`, todos os participantes devem ter sincronizado. Os papéis, por outro lado, só serão executados, ou seja passarão do estado `WAIT` para o estado `START`, quando todos os participantes estiverem no estado `EXEC` e a pré-condição (autômato `GUARD`) estiver no estado `TRUE`. Estes testes são realizados pela transição funcional `F1`.

Quando um participante termina a execução de um papel de maneira correta, o autômato `PAPEIS` P_i`.resultado` correspondente passará para o estado `OK`. Quando todos autômatos `PAPEIS` P_i`.resultado` estiverem no estado `OK`, então a transição funcional `F4` será ativada e todos os papéis passarão do estado `START` para o estado `ENDED`. O término da execução de todos os papéis será efetuado quando a pós-condição (autômato `ASSERT`) estiver no estado `TRUE` e todos os papéis estiverem no estado `ENDED`. Este teste é realizado pela transição funcional `F5`.

Quando todos papéis estiverem novamente no estado `WAIT`, então os participantes podem executar suas transições sincronizadas $S2_i$, passando novamente para o estado `WAIT`, e também passando a DMI do seu estado `START` para o estado `WAIT`.

Quando um papel levantar uma exceção, então o autômato `PAPEIS` P_i`.resultado` passa ou do estado `WAIT` ou do estado `OK` para o estado `EXCEPTION`, fazendo que esta exceção seja tratada pela DMI. Caso uma exceção seja levantada por outro papel, então os demais papéis que estiverem ou no estado `WAIT` ou no estado `OK` passam para o estado `Interrupted`

```
declarations {}
  constant tx := 2;
  constant me := 2;
  constant es := 1;

functional F1:= (((st Player1 == EXEC) }
               && (st Player2 == EXEC) \&\& (st Player3 == EXEC)
               && (st Guard == TRUE)){*}2);}
functional F2:= ((st Handler == EXIST) {*}2);
functional F21:= ((st Handler == NOTEXIST) {*}2);
functional F31:=(((st RoleResult2 == EXCEPTION) || (st RoleResult3 == EXCEPTION)){*}2);
functional F32:=(((st RoleResult1 == EXCEPTION) || (st RoleResult3 == EXCEPTION)){*}2);
functional F33:=(((st RoleResult1 == EXCEPTION) || (st RoleResult2 == EXCEPTION)){*}2);
functional F4:=(((st RoleResult1 == OK) && (st RoleResult2 == OK) && (st RoleResult3 == OK)){*}2);
functional f5:=(((st RoleState1==ENDED) && (st RoleState2==ENDED) && (st RoleState3 == ENDED)
                && (st Assert == TRUE)){*}2);}
reachability := 1;

network dmi (continuous, 13){

automaton dmi1(1,4){
  node WAIT (1) { arc (START,3){sync(S11,1,0.4) sync(S12,1,0.3) sync(S13,1,0.3)}}
  node START (2) {arc (WAIT,3){sync(S21,1,0.4) sync(S22,1,0.3) sync(S23,1,0.3)}
  node PAPEIS P_i.resultado {arc (RESOLVED,3){sync(F31,1,0.4) sync(F32,1,0.3) sync(F33,1,0.3)}}
  node EXCEPTION (2){ arc (WAIT,0,F2) arc (START,0,F21)}
  node RESOLVED (1){ arc (EXCEPTION,0,1)}}

automaton Guard(1,2){
 node TRUE (1){ arc (FALSE,0,1)}
 node FALSE (1){ arc (TRUE,0,1)}}

automaton Assert(1,2){
 node TRUE (1){ arc (FALSE,0,1)}
 node FALSE (1){ arc (TRUE,1){ sync(F5,1,1)}}}

automaton Handler(1,2){
 node EXIST (1){ arc (NOTEXIST,0,1)}
 node NOTEXIST (1){ arc (EXIST,0,1)}}

//=============== Para cada automato i ======================

automaton Player_i(1,2){
 node WAIT (1){ arc (EXEC,1){sync(S1_i,me,1)}}
 node EXEC (1){ arc (WAIT,2){sync(S2_i,me,0.5) sync(F5,1,0.5)}}}

//=============== Para cada automato i ======================

automaton RoleState_i(1,3){
 node WAIT (1){ arc (START,1){ sync(F1,1,1)}}
 node START (1){ arc (ENDED,1){ sync(F4,1,1)}}
 node ENDED (1){ arc (WAIT,1){ sync(F5,1,1)}}}

//=============== Para cada automato i ======================

automaton RoleResult_i(1,4){
 node WAIT (3){ arc (INTERRUPTED,1){sync(F3_i,F3_i,1)} arc (OK,0,1) arc (EXCEPTION,0,1)}
 node INTERRUPTED (1){ arc (EXCEPTION,0)}
 node OK (3){ arc (WAIT,0,1) arc (INTERRUPTED,1){sync(F3_i,F3_i,1)} arc (EXCEPTION,0,1)}
 node EXCEPTION (1){ arc (WAIT,0,1)} }

//=============== RESULTS ======================
results { teste := st dmi1 == WAIT;}
```

Figura 3: Especificação textual da DMI CarregaGuindaste1 em SAN

para que a exceção levantada seja tratada por todos os papéis pertencentes a mesma DMI. As transições $\mathsf{L}_{(}i)$ representam transições locais dos autômatos, isto é, dependem apenas da sua taxa de disparo e não da transição de estado de outros autômatos.

Na Figura 2 é representado a possibilidade de uma ou mais exceções serem levantadas durante a execução de dos papéis pelos participantes.

Através dos resultados obtidos no PEPS observou-se que todos os estados da especificação são alcançáveis, pois obtivemos probabilidade de ocorrência desses estados maior que zero o que demonstra a alcançabilidade dos mesmos. A análise dos resultados baseou-se na verificação da alcançabilidade, isto é, verificou-se apenas se os estados são alcançáveis ou não, não levando em consideração os índices de desempenho que foram gerados.

4. Conclusão

Neste trabalho foi apresentada uma descrição geral do formalismo de Redes de Autômatos Estocásticos que foi utilizado para especificar formalmente uma DMI. Foi criado e demonstrado um modelo de especificação formal, e validado com o *software* PEPS [14].

A especificação formal de sistemas torna mais fácil o seu processo de desenvolvimento, pois dessa forma é possível corrigir erros de projeto antes da sua implementação, evitando assim desperdícios de recursos.

Um cuidado que deve-se tomar antes de especificar formalmente um sistema, é na escolha da linguagem de especificação formal, pois uma escolha errada pode ocasionar inconsistência na especificação.

Uma análise informal dos resultados obtidos nas especificações de DMI em TLA - *Temporal Logic of Actions* [10] e SAN (apresentada neste artigo), nos leva a concluir que em TLA a possibilidade de desenvolver especificações genéricas torna a modelagem de um determinado sistema mais abrangente pois por exemplo, num sistema multi-thread é possível modelar o sistema levando em consideração apenas as ações que devem ser executadas e não o número de threads que o sistema contém.

A modelagem de sistemas em SAN, por outro lado, deve ser feita de forma específica, isto é, num sistema multi-thread o número de threads tem de ser levado em consideração e não apenas o comportamento que cada thread tem. Em conpensação utilizando SAN para modelar sistemas é possível realizar simulações através de modelos matemáticos e gerar índices de desempenho e confiabilidade. Essas três características tornam SAN uma ferramenta muito robusta, pois além de validar o sistema é possível simular o modelo proposto.

Este modelo está correntemente sendo aplicado na modelagem de um sistema de controle de um sistema de manufatura controlado por computador existente na PUCRS.

Referências

[1] I Foster. Compositional parallel programming language. *ACM Transactions on Programming Languages and Systems*, 18(4):454–476, 1996.

[2] A. F. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In *OOPSLA Conference Proceedings*, volume 34, pages 435–446, Denver, Colorado - USA, November 1999. The Association for Computing Machinery (ACM), Inc.

[3] A. F. Zorzo. *Multiparty Interactions in Dependable Distributed Systems.* PhD thesis, University of Newcastle Upon Tyne, UK, 1999.

[4] E. A de Souza e Silva and R. R Muntz. Métodos computacionais de solução de cadeias de markov: aplicações a a sistemas de computação e comunicação. Porto Alegre - Brasil, 1992. Instituto de Informática UFRGS.

[5] K. Trivedi. Probability e statistic with reliability, queueing and computer science aplications. *Englewwod Cliffs*, 1982.

[6] B. Plateau and K. Atif. Stochastic automata networks for modelling parallel systems. *IEEE Transactions on Software Engineering*, 17(10):1093–1108, 1991.

[7] W. J. Stewart. Introduction to numerical solutions of markov chains. *Princeton University Press*, 1994.

[8] P. Fernandes, B. Plateau, and W.J. Stewart. Eficient descriptor-vector multiplication in stochastic automata networks. *Journal of the ACM*, 45(3):381–414, 1998.

[9] L. A. Cassol. Especificação formal de interações multi-participantes confiáveis em lógica temporal de ações e redes de autômatos estocásticos. Trabalho individual I, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre - Brasil, 2001.

[10] A. F. Zorzo, A. Romanovsky, and B. Randell. *TLA Specification of a Mechanism for Concurrent Exception Handling.* Kluwer Publ., Holland, 2002.

[11] A. F. Zorzo. Dependable multiparty interactions: A case study. In *29th Conference on Technology of Object-Oriented Languages and Systems - TOOLS29-Europe*, pages 319–328, Nancy, France, 1999. IEEE Computer Society Press.

[12] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, and F. von Henke. Rigorous development of an embedded fault-tolerant system based on coordinated atomic actions. *IEEE Transactions on Computers*, 51(2):164–179, 2002.

[13] L. Cassol. Controle de célula de produção de tempo real com DMIs. In *III Workshop de Testes e Tolerância a Falhas*, Búzios, RJ, Brasil, Maio 2002.

[14] PEPS Project. PEPS project: Performance evaluation of parallel system, ID-INRIA/IMAG/PUCRS. WWW, 30 mar. 2002.

Equivalência de Programas e o Teste de Software: Resultados de um Experimento de Aplicação do Critério Análise de Mutantes

Inali Wisniewski Soares
inali@unicentro.br
UNICENTRO, CP: 730, 85015-430
Guarapuava, PR

Silvia Regina Vergilio
silvia@inf.ufpr.br
DInf- UFPR, CP: 19081, 81531-970
Curitiba, PR

Resumo

O critério de teste Análise de Mutantes, baseado em erros, tem despertado grande interesse e se mostrado bastante efetivo em detectar erros, um dos principais objetivos da atividade de teste de software. Entretanto, a existência de mutantes equivalentes dificulta a sua completa automatização visto que, determinar se dois programas são equivalentes é uma questão indecidível. Essa tarefa necessita ser realizada manualmente e eleva o esforço e o custo da aplicação do critério. Esse trabalho explora a problemática de equivalência de programas na atividade de teste, apresentando resultados de um experimento de aplicação do critério Análise de Mutantes. Os resultados incluem a porcentagem de mutantes equivalentes gerados e a lista de operadores de mutação que mais geram equivalência. Esses resultados são importantes pois contribuem para o estabelecimento de uma estratégia de aplicação do critério Análise de Mutantes, bem como para implementação de mecanismos que auxiliem a identificação automática de mutantes equivalentes e conseqüentemente para reduzir os custos e esforço gastos com o teste.

Palavras-chave: teste de software**,** análise de mutantes, mutantes equivalentes**.**

1. Introdução

Dentre as atividades de verificação e validação, o teste é considerado fundamental para garantir a qualidade do software. Entretanto, essa atividade costuma custar mais caro do que deveria. Na tentativa de reduzir esse custo, diversas técnicas de teste têm sido propostas para selecionar os melhores casos de teste que possam revelar a maioria dos defeitos, conseguindo redução de tempo e esforço.

Às técnicas de teste, geralmente, estão associados critérios de teste, que podem ser utilizados tanto para auxiliar na geração de conjuntos de dados de teste como para auxiliar na avaliação da adequação desses conjuntos, ou seja para oferecer medidas a serem utilizadas para se considerar a atividade de teste encerrada.

A técnica baseada em erros deriva os casos de teste considerando os principais erros que geralmente ocorrem durante o processo de desenvolvimento de software. A Análise de Mutantes e a Semeadura de Erros são critérios típicos que se concentram em erros [2].

A Análise de Mutantes é um critério que utiliza um conjunto de programas ligeiramente modificados, denominados mutantes, obtidos a partir do programa P em teste. O conjunto é utilizado para selecionar e avaliar os dados de teste. O objetivo é encontrar um conjunto de casos de teste T capaz de revelar as diferenças de comportamento existentes entre P e seus mutantes [5]. Os mutantes gerados e executados com o conjunto de casos de teste devem ser mortos, isto é, apresentar resultados diferentes do programa original. Uma medida de cobertura dada pelo número de mutantes gerados e pelo número de mutantes mortos é utilizada para avaliar um dado conjunto de teste.

Resultados de comparações teóricas e empíricas entre diferentes critérios de teste

[10,11,12] apontam o critério Análise de Mutantes como um dos mais eficazes em revelar defeitos. Porém a sua aplicação efetiva requer a existência de uma ferramenta de teste automatizada. Dentre essas, podemos citar as ferramentas Mothra [6], Proteum [4]. A existência dessas ferramentas fez com que, nas últimas décadas, o critério Análise de Mutantes se tornasse cada vez mais popular e utilizado.

Um problema para completamente automatizar a aplicação do critério Análise de Mutantes é a existência de mutantes equivalentes. Um mutante é equivalente se ele apresenta o mesmo comportamento que o programa em teste P independentemente da entrada, ou seja, não existe um caso de teste que é capaz de diferenciar P de um mutante equivalente. Os mutantes equivalentes precisam ser identificados e descontados para calcular o escore de mutação e garantir a satisfação do critério.

A determinação de mutantes equivalentes é uma questão indecidível [2], ou seja não existe algoritmo de propósito geral para determinar a equivalência entre dois programas. Isso resulta em muitos problemas para a atividade de teste relacionados a automatização. Embora estudos teóricos tenham sido conduzidos [1,3,7,8] no intuito de propor heurísticas para determinar a equivalência de programas, a determinação de mutantes equivalentes é a atividade que exige maior esforço do testador, pois nas ferramentas citadas acima, ela é realizada manualmente, o que contribui para elevar o custo do teste de mutação.

Esse trabalho explora a problemática de equivalência de programas na atividade de teste, mais particularmente na aplicação do critério Análise de Mutantes. São apresentados resultados de um experimento realizado com a ferramenta Proteum. Os resultados incluem a porcentagem de mutantes equivalentes gerados e a lista de operadores de mutação dessa ferramenta que mais geraram equivalência. Esses resultados são importantes pois contribuem para o estabelecimento de uma estratégia para reduzir o custo de aplicação do critério Análise de Mutantes, assim como podem ser utilizados para implementar mecanismos na ferramenta Proteum para identificação de mutantes equivalentes.

Esse artigo está dividido do seguinte modo. A Seção 2 mostra resumidamente os principais conceitos relacionados ao critério Análise de Mutantes. A Seção 3 descreve como foi realizado o experimento. Na Seção 4 são analisados os resultados obtidos. A Seção 5 contém as conclusões.

2. Teste de Mutação e Equivalência de Programas

Entre os vários critérios propostos para se conduzir e avaliar a qualidade da atividade de teste, destaca-se o critério Análise de Mutantes [5], baseado em erros.

O critério Análise de Mutantes, proposto por DeMillo [5] fundamenta-se em dois pressupostos: 1) hipótese do programador competente: “programadores experientes escrevem programas muito próximos do correto"; e 2) efeito de acoplamento: “erros complexos são decorrentes de erros simples". Assim, espera-se que se existe um conjunto de testes que detecte erros simples em um programa então este poderá também revelar erros complexos.

A idéia básica do critério é gerar para um programa que será testado, vários programas mutantes diferentes desse programa por pequenas alterações sintáticas. Para satisfazer o critério é necessário gerar um dado de teste que faça com que cada mutante comporte-se diferentemente do programa em teste. Neste caso, o mutante é dito morto. Quando não existir nenhum dado de teste capaz de distinguir um mutante do programa em teste, o mutante é equivalente. O escore da cobertura obtida através do número de mutantes mortos e número de mutantes gerados, descontando-se o número de mutantes equivalentes é usada para avaliar o

conjunto de dados de teste sendo utilizado. Aliada ao alto custo computacional e espacial, a existência de mutantes equivalentes dificulta a utilização prática do critério.

A análise dos mutantes equivalentes é o passo que requer mais intervenção humana, e um dos maiores obstáculos para a aplicação do teste de mutação. Sem determinar todos os mutantes equivalentes o escore de mutação nunca será 100%. Assim o testador não terá completa confiança no programa e nos dados de teste. Pior, o testador será incapaz de saber se os mutantes restantes são equivalentes ou se o conjunto de teste é insuficiente. Determinar mutantes equivalentes manualmente consome muito tempo; é necessário um entendimento completo do programa, o que contribui para elevar o custo do teste de mutação.

A ferramenta Proteum [4] apóia o teste de mutação para programas C. A ferramenta Proteum oferece recursos ao testador para, através da aplicação do critério Análise de Mutantes, avaliar a adequação de, ou gerar um conjunto de casos de teste T para determinado programa P.

Em geral, o problema de resolver se dois programas são equivalentes é indecidível; essa limitação teórica não significa que o problema deva ser abandonado por não ter solução. Na verdade alguns métodos e heurísticas têm sido propostos para determinar a equivalência de programas em uma grande porcentagem dos casos de interesse [1,3,7,8].

Os programas equivalentes têm duas vantagens sobre o problema geral de equivalência, primeiro, programas mutantes são muito semelhantes aos programas originais. Budd e Angluin descrevem mutantes como vizinhos dos programas originais e pesquisadores utilizam esse fato para desenvolver técnicas e heurísticas para determinar mutantes equivalentes [7,8]. A segunda vantagem é que o teste de software é inerentemente uma ciência imperfeita; assim, os resultados parciais são de grande importância.

Um estudo teórico dos principais trabalhos que tratam da determinação de mutantes equivalentes foi realizado e uma descrição desses trabalhos está em [9].

3. Descrição do Experimento

Foram utilizados no experimento oito programas utilitários UNIX escritos na linguagem C, que são de domínio público: *cal, checkeq, col, comm, look, spline, tr e uniq*. Esses programas fazem parte de um benchmark e foram utilizados em diversos trabalhos da literatura com o objetivo de comparar diferentes critérios de teste [10,11,12].

Para realizar este experimento foi utilizada a ferramentas Proteum [4] para teste de programas no nível de unidade. A aplicação do experimento consistiu de duas fases principais: satisfazer o critério Análise de Mutantes e verificar os operadores que geraram o maior número de mutantes equivalentes.

I. Aplicação do Critério Análise de Mutantes

1. Geração dos programas mutantes: para geração dos mutantes foram utilizados todos os operadores disponíveis na ferramenta Proteum.

2. Geração de conjuntos AM-adequados: Para gerar os conjuntos AM-adequados fez-se uso dos conjuntos de dados de teste gerados de maneira "ad-hoc" e aleatória utilizados em [11], e submissão desses dados à ferramenta Proteum. Baseando-se nesses conjuntos e nas informações fornecidas pela ferramenta Proteum (mutantes vivos), foram adicionados novos casos de teste ao conjunto até obter-se um conjunto AM-adequado para cada programa. A

determinação de mutantes equivalentes é responsabilidade do testador, visto que a ferramenta Proteum utilizada não possui mecanismos que automatizem essa atividade. Os mutantes foram analisados para cada programa, e a equivalência foi determinada manualmente.

II. Operadores X Equivalência: para verificar a relação operador de mutação e número de mutantes gerados, foi utilizada a opção gerar relatórios da ferramenta Proteum.

4. Resultados

4.1 Porcentagem de Mutantes Equivalentes

Na Tabela 1 são apresentadas informações referentes à aplicação do critério Análise de Mutantes aos programas: número de mutantes gerados, mortos e equivalentes; e na última coluna é apresentada a cobertura, sem descontar os mutantes equivalentes. Esses dados demonstram o grau de complexidade dos programas considerados no experimento, onde é possível observar que:

- o programa spline teve o maior número de mutantes gerados 12560;
- o programa *uniq* teve o menor número de mutantes gerados 1623;
- os programas que tiveram o maior número de mutantes mortos e consequentemente o menor número de mutantes equivalentes (em porcentagem) foram os programas *cal* e *checkeq*;
- programa que teve o menor número de mutantes mortos e consequentemente o maior número de mutantes equivalentes foi o programa *tr*, devido ao grande número de variáveis e constantes existentes nesse programa;
- A média de mutantes equivalentes determinados no experimento representa 11,65% dos mutantes gerados e não deve ser desprezada pois grande quantidade de esforço foi gasto na atividade de determinação.

A determinação manual dos mutantes equivalentes consumiu uma grande quantidade de esforço e tempo. Em média foram gastas 240 horas para essa tarefa, foram analisados pelo menos 4.298 mutantes. Pode-se concluir que é muito importante que as ferramentas de teste ofereçam mecamismos que auxiliem o usuário nessa tarefa.

Tabela 1: Aplicação do critério AM

Programa	Número de Mutantes			Cobertura
	Gerados	Mortos	Equivalentes	
Cal	4334	4015	309 (7%)	93.00
Checkeq	3075	2861	214 (7%)	93.00
Col	6910	6023	887 (13%)	87.00
Comm	1938	1652	281 (15%)	85.00
Look	2030	1814	216 (11%)	89.00
Spline	12560	11134	1426 (11%)	89.00
Tr	4422	3632	790 (18%)	82.00
Uniq	1623	1463	160 (10%)	90.00
Total	36892	32594	4298 (11,65%)	88.35

4.2 Operadores e Equivalência

A Tabela 2 apresenta os totais de mutantes gerados e equivalentes determinados para os operadores da ferramenta Proteum, que geraram o maior e o menor número de mutantes equivalentes, para os programas em teste: *cal, checkeq, col, comm, look, spline, tr e uniq.* Um glossário de siglas e descrição dos operadores utilizados no trabalho é apresentado no apêndice. Através desses dados pode-se observar que:

- Os quatro operadores que geraram o maior número de mutantes equivalentes em relação ao número total de mutantes gerados em porcentagem, em ordem (do maior para o menor) foram: OCOR, VDTR, OLBN, e OEBA.
- Os quatro operadores que geraram o menor número de mutantes equivalentes em relação ao número total de mutantes gerados em porcentagem, em ordem (do menor para o maior) foram: STRP, STRI, SSWM e OABA.

A Tabela 3 apresenta os quatro operadores que mais geraram mutantes equivalentes por programa do experimento.

Tabela 2: Operadores que geraram o maior e o menor número de mutantes equivalentes

Maiores				Menores			
Operador	Mutantes			Operador	Mutantes		
	Gerados	equivalentes	%		gerados	Equivalentes	%
OCOR	312	300	96.15	STRP	941	12	1.28
VDTR	1818	692	38.06	STRI	312	5	1.59
OLBN	144	54	37.5	SSWM	62	1	1.61
OEBA	624	211	33.81	OABA	51	1	1.96

Tabela 3: Operadores que geraram o maior número de mutantes equivalentes por programa

Programa	**1º**	**2º**	**3º**	**4º**
Cal	VDTR	Cccr	OEBA	OEAA
Checkeq	VDTR	OEBA/ ORAN/ ORRN	OEAA	OLAN/ ORBN
Col	OCOR	Cccr	VDTR	Vsrr
Comm	Cccr	OEBA/ VDTR	OEAA	ORAN/ORBN
Look	VDTR	Cccr	OCOR	OEBA
Spline	Vsrr	VDTR	CRCR	OEAA
Tr	Vsrr	Cccr	VDTR	OEBA
Uniq	OCOR	VDTR	Cccr	OEBA/ORRN

A seguir são apresentados, com uma breve explicação, os operadores da ferramenta Proteum que geraram o maior número de mutantes equivalentes em relação ao total de mutantes gerados para os programas em teste através de exemplos das mutações equivalentes ocorridas nos programas do experimento para esses operadores. Maiores informações referentes ao experimento podem ser encontradas em [9].

OCOR - Cast Operator by Cast Operator

Este operador pertence ao grupo das mutações de operadores. Os operadores de *"cast"* envolvem os tipos primitivos da linguagem (*int, char, long, float, etc*) e são trocados por todos os outros tipos primitivos da linguagem. A Figura 1 exemplifica esse operador, onde é utilizada a rotina *canon* do programa *look.* O objetivo da função *tolower()* é devolver o equivalente minúsculo da variável *c* se esta for uma letra, caso contrário *c* será devolvida sem alteração. A mutação do valor de retorno da função *tolower()* que é um tipo *int* por um tipo *float* nesse caso produzirá uma saída do programa igual ao programa original pois um tipo

float pode armazenar sem problemas um tipo *int;* portanto, esse mutante é equivalente ao original.

VDTR - Domain Traps

Este operador pertence ao grupo de mutações de variáveis, onde cada referência escalar é substituída pelas chamadas às funções *trap_on_zero (e)*, *trap_on_positive (e)* e *trap_on_negative(e).* Estas funções abortam a execução dos mutantes caso a expressão tenha valor zero, positivo ou negativo, respectivamente. Caso contrário retornam o valor da expressão.

A Figura 2 exemplifica esse operador, onde é utilizada algumas linhas da rotina *main* do programa *cal.* Nesse exemplo a variável *argc* é um parâmetro que contém o número de argumentos da linha de comando e é um inteiro, *argc* é sempre pelo menos 1 porque o nome do programa é qualificado como primeiro argumento, e não é possível tornar a expressão *argc < 0* verdadeira, e fazer o mutante se comportar diferentemente do programa original.

OLBN - Logical Operator by Bitwise Operator

Este operador pertence ao grupo de mutações de operadores, onde cada operador lógico é substituído por um operador bitwise.

A Figura 3 exemplifica esse operador, onde é utilizada a rotina *main* do programa *cal.* O operador lógico || no comando if(m<1 || m>12) é trocado pelo operador bitwise ∧, independentemente de quais valores a variável *m* possuir o resultado será equivalente ao resultado do programa original; como exemplo, se a variável *m* possuir o valor -1 o comando será executado e a mensagem indicando " Bad month, -1" será impressa porque para os dois programas 0 || 1 e 0 ∧ 1 será 1.

OEBA - Plain Assignment by Bitwise assignment

Este operador pertence ao grupo das mutações de operadores e troca atribuição plana por atribuição bitwise.

A Figura 4 exemplifica esse operador, onde é utilizada a rotina *main* do programa cal. A atribuição plana = do comando for (i=0; i < 6*24; i+=24) é trocada pela atribuição bitwise &=, e o resultado dessa atribuição para a variável *i* continua sendo 0, portanto existe equivalência entre esses programas.

```
Canon(char * src, char * copia)
{  int cnt;  char c;
  for (cnt = len + 1; (c = *src++) && cnt; --cnt)
     if (!dict || isalnum(c))
          *copia++ = fold && isupper(c) ? (int)tolower(c) : c;
■         *copia++ = fold && isupper(c) ? (float)tolower(c) : c;
     *copia = EOS; }
```

Figura 1: Exemplo: Operador OCOR

```
main(int argc, char *argv[ ])
{       register y, i, j;  int m;
        if (argc == 2)
■       if ((TRAP_ON_NEGATIVE(argc) == 2))
            .....}
```

Figura 2: Exemplo: Operador VDTR

```
Main(int argc, char *argv[ ])
{                       register y, i, j; int m;
        . . .
         m = atoi(argv[1]);
         if(m<1 || m>12) {
■        if(m<1 ∧ m>12) {
             fprintf(stderr, "cal: %s: Bad month.\n", argv[1]);
                  exit(1);}}
```

Figura 3: Exemplo: Operador OLBN

```
Main(int argc, char *argv[ ])
{           .....
            for(i=0; i<6*24; i+=24)
■           for(i&=0; i<6*24; i+=24)
                    pstr(string+i, 24);
            .....
```

Figura 4: Exemplo: Operador OEBA

4.3 Estratégia de Aplicação do Critério Análise de Mutantes Baseada em Equivalência

A Proteum permite que o usuário selecione a porcentagem de mutantes a serem gerados por um determinado operador de mutação. Nesse sentido, os resultados aqui descritos podem ser utilizados para estabelecer uma estratégia de aplicação do Critério Análise de Mutantes. Essa estratégia estabelece que porcentagens baixas ou nulas sejam atribuídas a operadores que geraram um alto número (porcentagem) de mutantes equivalentes. São eles: OCOR, VDTR, OLBN, OEBA.

Os operadores que geraram o menor número de mutantes equivalentes foram incluídos na ferramenta Proteum para se garantir a inclusão do critério Todos-Ramos. Em [SOA02] foi proposta uma estratégia incremental de aplicação de diferentes critérios de teste que sugere seguir a relação de inclusão entre critérios para estabelecer uma ordem de aplicação dos mesmos. Parte-se primeiramente de critérios mais "fracos". Uma possível ordem é Critérios Baseados em Fluxo de Controle, em Fluxo de Dados, em Restrições e por último Baseados em Erros, tais como o critério Análise de Mutantes.

Durante a realização do experimento aqui descrito, observou-se que a maioria dos mutantes gerados por operadores que geraram o menor número de mutantes equivalentes, são facilmente cobertos. Se uma estratégia incremental de aplicação dos critérios tal como a descrita em [Soa00] for utilizada, esses operadores poderão ser desconsiderados, reduzindo o número de execuções do programa e consequentemente o custo do critério AM.

5. Conclusões

A determinação de mutantes equivalentes é uma questão indecidível e isso dificulta a completa automatização do critério baseado em erros Análise de Mutantes. Essa tarefa precisa, em geral, da intervenção humana e exige um grande esforço, aumentando os custos da atividade de teste.

Esse trabalho focalizou a problemática de mutantes equivalentes, apresentando resultados de um experimento de aplicação do critério Análise de Mutantes. Os resultados obtidos indicam, para os programas utilizados, que 11,65% dos mutantes gerados pelos operadores da ferramenta Proteum são equivalentes. Essa porcentagem pode indicar uma medida a ser utilizada para se estimar o número de mutantes equivalentes de um programa. Essa medida seria considerada para se encerrar a aplicação do critério Análise de Mutantes. Por exemplo, considerar durante o teste que uma cobertura real em torno de 90% é suficiente. Entretanto, novos experimentos deverão ser conduzidos com o objetivo de se obter um modelo empírico a ser utilizado em estimativas como essa, pois os programas utilizados nesse experimento constituem uma pequena amostra.

Os resultados observados sobre equivalência descritos na seção 4.2 servirão como base para implementar mecanismos e heurísticas para determinação de mutantes equivalentes, facilidades que poderão ser incorporadas à ferramenta Proteum, visto que a automatização completa é indecidível.

A estratégia proposta deverá ser avaliada em experimentos futuros. Sugere-se que a utilização ou não dos operadores que mais ou menos geraram mutantes equivalentes seja avaliada pelo usuário que poderá estabelecer a sua própria estratégia de aplicação para o critério Análise de Mutantes.

Referências Bibliográficas

[1] T.A. Budd, D. Angluin. *Two notions of correctness and their relation to testing*. Acta Informatica, Vol. 18(1):31-45, Nov., 1982.

[2] T.A. Budd. *Mutation Analysis: Ideas, Examples, Problems and Prospects, Computer Program Testing.* North-Holand Publishing Company, 1981.

[3] D. Baldwi, F. Sayward. *Heuristics for Determining Equivalence of Program Mutations*. CT, Res. Rep. 276, Dep. of Comp. Science-Yale University, New Haven, 1979.

[4] M.E. Delamaro, "*Proteum - Um ambiente de teste baseado na Análise de Mutantes*", Dissertação de Mestrado, ICMSC/USP - São Carlos, SP, Brasil, Oct., 1993.

[5] R.A. De Millo, R.J. Lipton, F.G. Sayward. *"Hints on Test Data Selection: Help for the Practicing Programmer"*. IEEE Computer, April, 1978.

[6] K.N. King, A.J. Offutt. *A Fortran Language System for Mutation-Based Software Testing. Software Practice and Experience.* Vol. 21(7):685-718, July 1991.

[7] A.J. Offutt, W. M. Craft. *Using compiler optimization techniques to detect equivalent mutants. The Journal of Software Testing, Verification, and Reliability*. 4(3):131-154. Sept. 1994.

[8] A.J. Offutt, Jie Pan. *Automatically Detecting Equivalent Mutants and Infeasible Paths. The Journal of Software Testing, Verification, and Reliability.* 7(3):165-192. Sept. 1997.

[9] I.W. Soares. *Análise de Mutantes e Critérios Restritos no Contexto de Teste de Software: Resultados de uma Avaliação Empírica*. Dissertação de Mestrado, UFPR, Curitiba, 2000.

[10] I.W. Soares, S.R. Vergilio. *Mutation Analysis and Constraint-Based Criteria: Results from an Empirical Evaluation in the Context of Software Testing.* III IEEE Latin-American Test Workshop, pg 33-38, vol 1. Montevideo, Feb. 2002.

[11] S.R. Vergilio. *Critérios Restritos de Teste de Software: Uma Contribuição para Gerar Dados de Teste mais eficazes. Doctorate Dissertation*, DCA/FEEC/Unicamp, Campinas – SP. July. 1997.

[12] W.E. Wong, A.P. Mathur, J.C. Maldonado. *Mutation versus all-usos: An Empirical evaluation of cost, strength and effectivenes.* In Software Quality and Productivity-Theory, Pratice, Education and Training. Hong Kong, Dec. 1994.

Apêndice – Descrição dos Operadores da Ferramenta Proteum

Operador	**Descrição**
Cccr	Troca constantes por todas as constantes do programa
CRCR	Troca constantes por 0, 1 e –1, dependendo do tipo de referência
OABA	Troca atribuição aritmética por operador de atribuição bitwise
OCOR	Troca o tipo primitivo do operador cast
OEAA	Troca atribuição plana por atribuição aritmética
OEBA	Troca atribuição plana por atribuição bitwise
OABA	Troca operador aritmético por operador booleano
OLAN	Troca operador lógico por operador aritmético
OLBN	Troca operador lógico por operador bitwise
ORAN	Troca operador relacional por operador aritmético
ORBN	Troca operador relacional por operador bitwise
ORRN	Troca operador relacional por operador relacional
SSWM	Força a execução de todos as alternativas do comando switch
STRI	Força a execução de true e false para cada if
STRP	Força a execução de todos os caminhos do programa
VDTR	Força cada referência escalar ser negativa, positiva e zero
Vsrr	Substitui referências a vetores pro variáveis escalares, globais e locais do programa

Simulation of a Distributed Connectivity Algorithm for General Topology Networks

Elias Procópio Duarte Jr.
Andréa Weber

Federal University of Paraná, Dept. Informatics
P.O. Box 19081 Curitiba 81531-990 PR Brazil
e-mail: {elias,andrea}@inf.ufpr.br

Abstract

The Distributed Network Connectivity *algorithm allows every node in a general topology network to determine to which portions of the network it is connected, and which portions are unreachable. The algorithm consists of three phases: test, dissemination, and connectivity computation. During the testing phase each fault-free node tests all its neighbors at each testing interval. Upon the detection of a new fault event, the tester starts the dissemination phase, in which a reliable broadcast algorithm is employed to inform the other connected nodes about the event. At any time, any working node may run the third phase, in which a graph connectivity algorithm shows the complete network connectivity. Extensive simulation results are presented, considering various topologies such as the hypercube and random graphs. Results are compared to those of other algorithms.*

Keywords: Distributed Diagnosis, Network Connectivity, Reliable Broadcast, Distributed Algorithms.

1 Introduction

Organizations and individuals increasingly depend on the correct behavior of network systems. Given the topology, it is important to determine at any instant of time which portions of the network are connected and which portions are unreachable. In this work we present a new algorithm for computing Distributed Network Connectivity (DNC). This algorithm is based on previously published results in System-Level Diagnosis of General Topology Networks [4].

The first model of a diagnosable system is known as the PMC model [2]. This model considers a fully connected system which can be represented by a complete graph. The vertices represent system units, also called nodes in this work, which are capable of executing tests on any other unit. An edge represents a communication channel, also called link, between two units. In this model, a node may assume one of two states: *faulty or fault-free*, and links are always fault-free.

The main assumption of the PMC model is that a fault-free node is reliable in the sense that whenever such a node executes a test on another node, it correctly determines the state of the tested node. Furthermore the tester is also capable of correctly reporting test outcomes. Based on the test reports sent by all nodes, a central monitor performs the diagnosis itself, determining which nodes are faulty and which are fault-free.

Distributed diagnosis eliminates the need for a central observer [6]. The nodes that execute the tests also perform the diagnosis. Adaptive diagnosis is another approach that allows testers to determine which tests should be executed in the next testing interval based on the results of the tests executed in the previous interval [7]. A number of algorithms for fully connected networks are at the same time distributed and adaptive [5].

Local Area Networks (LAN's) are usually modeled as fully connected systems, while Wide Area Networks (WAN's) have general topologies. In a general topology network there is not necessarily a communication channel between every two nodes. Both nodes and links may be either faulty or fault-free at a given instant of time. In this way, a fault may partition the network. Consider a pair of nodes connected by a link. If the tests executed on one of those nodes by the other results in a time-out, it is impossible to determine whether the tested node or the link is faulty. In this way, faults in a general topology network are said to be ambiguous [3].

In [6] Bagchi and Hakimi introduced an algorithm for system-level diagnosis of networks of general topology. The algorithm is executed *off-line*. In [7] Bianchini *et.al.* introduced and evaluated through simulation the Adapt algorithm. The Adapt algorithm can be executed on-line: when a given node becomes faulty, a new phase begins in which other nodes reconnect the testing graph. The algorithm employs a distributed procedure that requires massive amounts of large diagnostic messages.

Rangarajan *et.al.* [8] introduced the RDZ algorithm for system-level diagnosis for networks of arbitrary topology that can be executed on-line. The algorithm builds a testing graph that guarantees the optimal number of tests, i.e., each node has one tester. Furthermore it presents the best possible diagnosis latency by using a parallel dissemination strategy. Whenever a node detects an event, it sends diagnostic information to all its neighbors, which in turn send the information to all its neighbors, and so on. Although the RDZ algorithm presents the best possible diagnosis latency, and the best possible number of testers per node, it does not diagnose a fault configuration called by the authors *jellyfish faulty node configuration*.

The Distributed Network Connectivity (DNC) algorithm introduced in this paper is based on the Non-Broadcast Network Diagnosis (NBND) algorithm [3, 4, 9]. The algorithm is structured in three phases: test, dissemination and diagnosis. Nodes test adjacent links every testing interval. Upon the detection of a new event, information is disseminated to all nodes through a distributed breadth-first tree, instead of using flooding such as in NBND. Based on that information each fault-free node may compute the network connectivity. The algorithm allows dynamic events, i.e. during the dissemination phase new events may occur and dissemination remains guaranteed. This is done by means of a reliable broadcast algorithm [10] for event dissemination. The proposed strategy is compared with two other approaches: dissemination based on flooding [12] and a sequential strategy based on a distributed Chinese Agent [13]. The strategy employed by DNC is nearly always better than the other two.

The rest of the paper is organized as follows. In section 2 the DNC algorithm is specified. In section 3 simulation results are presented, comparing the cost of the dissemination phase in three algorithms: DNC, Chinese Agent and Flooding. Section 4 concludes the paper.

2 The DNC Algorithm

In this section, we introduce a new distributed connectivity algorithm for general topology networks. The algorithm has three phases: testing, dissemination of new event information, and local connectivity computation. The algorithm considers a synchronous system and crash faults. Both nodes and links may be either faulty or fault-free. However, if there is no reply to a given test over a link, the tester is not able to determine whether it is the tested link or the tested node connected by that link that is faulty. If there is no reply to tests executed over all links to a given node, then the node is *unreachable*. Thus links may be in one of two states *fault-free* or *unresponsive* and nodes may be *fault-free* or *unreachable*.

Every link is tested every testing interval. There is one tester per link. The algorithm thus requires the minimum number of tests for any network topology. The algorithm employs a *token-based testing strategy* [9]. The two nodes connected by a link execute tests over that link at alternating intervals. The tests employed are also said to be *two-way tests*, in the sense that when one node executes a test over a link, not only the tester determines the state of the tested node, but also the tested node determines the state of the tester.

Each node keeps a timestamp which is a state counter for each link in the system, which is initially zero, and is incremented at each new event information received for that link. This permits a node to identify redundant messages. A redundant message when it is not the firt one about an event. After a new event is discovered, the tester propagates event information to its neighbors. Every node keeps the complete network topology. The parallel dissemination strategy employed is based on a distributed breadth-first tree. Each diagnostic message carries the following information (1) the tester identifier (2) the tested node identifier (3) the timestamp for the tested link. Each node running the algorithm keeps a *link table* indexed by link identifier, containing the timestamp for the link. An even timestamp indicates a fault-free link; an odd timestamp indicates a faulty link.

Whenever a node is a leaf, after receiving the message from its parent it sends back an acknowledgement called hence forth *ack*. After receiving acks from all its children, a node sends an acknowledgement to its parent in the tree. The dissemination completes when the root receives acknowledgements from all its children.

When link a–b becomes faulty, the dissemination proceeds in the following way. Consider that node a is the first to detect the event. Node a starts a dissemination tree, informing the event on link a–b. After this dissemination message reaches node b, this node will determine that link b–a is faulty. Node b gives up the previous dissemination, appending the previous message to the new one, and takes its role in the second dissemination tree. Whenever a node that has a pending dissemination receives a message that contains information about the pending dissemination plus new information, it gives up the first dissemination, and takes its role in the second dissemination.

At any given time a fault-free node running the algorithm may compute the local network connectivity after removing the links that are in the *unresponsive* state, i.e. have an odd timestamp, from the network topology.

2.1 An Example Execution

Consider the example topology in figure 1. This subsection contains a description of the execution of the algorithm, considering a fault event on one link, and subsequent dissemination of new event information.

A fault event on link 1–3 is considered to happen at instant of time t_0. In the next testing interval after the event occurs, consider that node 1 is the tester, and thus detects the fault. Then node 1 starts the dissemination phase in order to communicate the event information to all other reachable nodes. It does so by building a breadth-first traversal tree rooted at itself, yet considering the fault event. The tree is illustrated in figure 2.

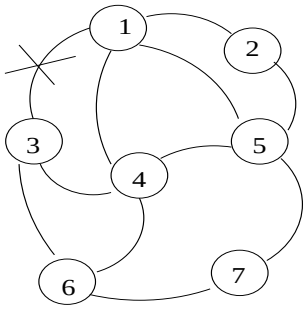


Figure 1: An example topology. Link 1–3 becomes faulty.

The dissemination tree has node 2, node 4 and node 5 at the second level. At the third level, node 4 has two children: node 3 and node 6; node 5 has one child: node 7. Its one time unit for a node to process a dissemination message and for this message to be received by this node's children. So, one time unit after the sending of the dissemination message by node 1 it reaches nodes 2, 4 and 5 simultaneously.

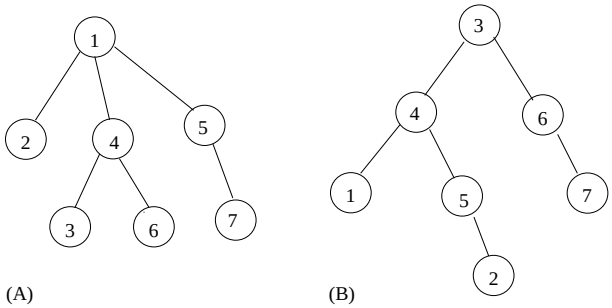


Figure 2: The dissemination trees built by node 1 (A) and node 3 (B).

In this example dissemination, node 2 is a leaf in the dissemination tree, so the message it receives is acknowledged in the next time unit. Nodes 4 and 5 disseminate the information for its children the next time unit, so that when the acknowledgement of node 2 reaches node 1, dissemination messages reach nodes 3, 6 and 7 at the bottom of the tree. Nodes 6 and 7 are ordinary leaves, thus they acknowledge the receipt of the message to nodes 4 and 5, respectively. Node 3, although, once informed about the fault on link 1–3 tests its adjacent links, and realizes that link 3–1 is "also" faulty. Thus, this information must be disseminated. For this to be done, node 3 builds another dissemination tree, as is illustrated in figure 2.

At this point there are two simultaneous dissemination trees. The first dissemination tree was started by node 1, and disseminates information about the fault event on link

1–3, which is called first dissemination message. The first dissemination tree is gradually being abandoned, as the second dissemination tree, which was started by node 3, has complete information about both events (link 1–3 is faulty, link 3–1 is faulty, called the second dissemination message), takes its place.

As the simulation continues, at the same time the acknowledgement messages from nodes 6 and 7 reach nodes 4 and 5 of the first dissemination tree, new information messages are sent by node 3 in the second dissemination tree. Nodes 4 and 6 receive this messages. Node 4 sends the dissemination message to nodes 1 and 5. When node 6 receives the second dissemination message, it has already completed its task in the first dissemination tree. So, it sends the second dissemination message to node 7, its child in the second tree. This happens simultaneously with the sending of the second dissemination message to nodes 1 and 5 by node 4.

At the third level of the second dissemination tree, nodes 1 and 7 are leaves which acknowledge the received messages to its parent nodes. At the same time these acknowledgements move up one level in the tree, node 5 sends the second message to node 2, its child in the second dissemination tree, which sends an acknowledgement after its receipt. At the same time node 5 receives the acknowledgement message from node 2, the root of the tree receives the acknowledgement message from node 6. The next time unit, node 5 acknowledges the message from node 2, and node 4 does the same after receiving the acknowledgement from node 5. After that the root at node 3 receives the last acknowledgement and considers the dissemination as finished. Once begun at the first root at node 1, the whole dissemination is completed in 8 time units.

3 Experimental Results

In this section, experimental results are presented and a comparison is made with results of two other approaches: a parallel algorithm based on flooding [12] and a sequential algorithm based on the Chinese Agent [13]. Various network topologies were considered: initially results were obtained for the example graph shown in figure 1. Next, results were obtained for hypercubes with 16, 64 and 128 nodes; then the algorithm was simulated on a 50-node random graph, the $D_{1,2}$ graph with 9 nodes, and a subset of the topology of the Brazilian National Research Network: RNP (*Rede Nacional de Pesquisa*).

Simulations were built using SMPL *SiMulation Programming Language* [11], a discrete event simulation library. For each simulation experiment, one communication link fault event was scheduled. Testing intervals of 30 time units, and dissemination intervals of 1 time unit between nodes connected by one link were employed.

3.1 An Example Topology

For the 7-node example topology shown in figure 1, results of the algorithm execution of a fault event on link 1–3 such as described in the previous section are shown in table 1.

	Total of Messages	Redundant Messages	Latency
DNC	12	0	9
Flooding	28	16	7
Chinese Agent	7	1	7

Table 1: Results for an example topology.

3.2 Graph $D_{1,2}$

For the $D_{1,2}$ graph topology [8], a fault event was scheduled for link 6–8. Node 6 starts the dissemination building a four-level breadth-first tree rooted at itself. In this way, it takes 2 time units for the dissemination to reach node 8. At that time, node 8 builds another dissemination tree and begins disseminating information about the event on link 6–8 plus the information of the fault event on link 8–6. For this dissemination, another four-level tree is built. Thus, it takes three time units for the information to reach the leaves of that tree, and three time units more for confirmations to reach node 8 at the root. Comparative results with the other mentioned algorithms are shown in table 2.

	Total of Messages	Redundant Messages	Latency
DNC	16	0	9
Flooding	52	36	7
Chinese Agent	13	5	13

Table 2: Results for the 9-node $D_{1,2}$.

3.3 Hypercubes

Results were obtained and are described below for hypercubes with 16, 64, and 128 nodes. For the 16-node hypercube a fault event on link 5–7 was scheduled. Node 5 detects the event and begins the dissemination. A five-level tree is built, with node 7 at the fourth level. Three time units after the beginning, node 7 is informed of the event on link 5–7 and starts the dissemination of the event on link 7–5. This dissemination takes another five-level tree, so that four time units after node 7 initiates the dissemination, the information reaches the bottom of the tree. Four time units after that the last acknowledgement message reaches the top. Results for the other approaches are shown in table 3.

	Total of Messages	Redundant Messages	Latency
DNC (16 nodes)	30	0	12
Flooding (16 nodes)	94	64	9
Chinese Agent (16 nodes)	28	14	28
DNC (64 nodes)	126	0	24
Flooding (64 nodes)	478	352	15
Chinese Agent (64 nodes)	156	93	156
DNC (128 nodes)	254	0	36
Flooding (128 nodes)	990	736	23
Chinese Agent (128 nodes)	348	221	348

Table 3: Results for the 16, 64 and 128 nodes hypercubes.

Considering the 64-node hypercube with a fault event detected on link 5–7 by node 7, the dissemination was performed with 126 messages in 24 time units. Considering the 128-node hypercube topology, with a fault event at link 13–21, the dissemination begins at node 13 and takes 254 messages and 36 time units to complete. Comparative results are shown in table 3.

3.4 A Random Graph

For this simulation a random graph was built with a probability of ten percent that a link exists between any pair of nodes. The number of nodes is equal to 50. The graph, nevertheless, consists of one connected component.

For such a graph, an event on link 30–31 was scheduled and the dissemination began on node 31. This node employed a five-level breadth-first traversal tree, with node 30 at the fourth level. Thus, three time units after the beginning, the dissemination reaches node 30. At this time, node 30 builds another five-level tree and begins disseminating the information of the fault event on link 30–31 together with the received information of the event on link 31–30. Four time units after this dissemination starts, it reaches the leave nodes, and other four time units after that are required for the acknowledgement messages to reach node 30 on the root. Comparative results are shown in table 4.

	Total of Messages	Redundant Messages	Latency
DNC	143	0	14
Flooding	418	320	8
Chinese Agent	109	60	109

Table 4: Results for a random graph.

3.5 A Subset of The Brazilian *RNP*

The *RNP* topology employed for the simulations was that available at http://www.rnp.br in February, 2001. That was a 28-node topology, and a fault event was scheduled for the Sao Paulo–Brasília link.

Once the fault is detected, the Brasilia node starts the dissemination phase employing a four-level breadth-first traversal tree. Two levels below on the tree is the Sao Paulo node, that receives the information two time units after the beginning of the dissemination. Once that happens, the Sao Paulo node starts another dissemination for informing the Sao Paulo–Brasilia fault event and does so by building another four-level breadth-first tree rooted at itself. Three time units after the second dissemination is started it reaches the leaves. Three time units after that, the acknowledgement messages arrive the top of the tree, completing the dissemination. Comparative results are shown in table 5.

	Total of Messages	Redundant Messages	Latency
DNC	54	0	9
Flooding	94	40	6
Chinese Agent	55	27	55

Table 5: Results for a subset of the Brazilian *RNP*.

4 Conclusion

The Distributed Network Connectivity algorithm described in this paper allows any node of a general topology network to compute to which portions of the network it is connected, and which portions are disconnected. Links are tested continually, at a testing interval, disseminating fault event information through a distributed breadth-first tree.

A link fault, and subsequent dissemination was simulated for a number of different topologies: an example network; the $D_{1,2}$ graph; 16, 64, and 128-node hypercubes; a random graph, and a subset of the Brazilian RNP. Results were compared with those of other two algorithms: Flooding and the Chinese Agent.

Flooding always completes the dissemination in the shortest possible interval of time. However it always employs a larger number of messages, with many redundant messages. The number of messages employed by the Chinese Agent is always much lower than the number of messages employed by Flooding. The Chinese Agent presents the lowest

impact on the network as at most one message is being disseminated at a given instant of time. The number of messages employed by DNC is similar to the number employed by the Chinese Agent, while DNC's latency is similar to Flooding's. So DNC has good latency at a low impact on the network. Beyond that, both Flooding and the Chinese Agent employ redundant messages, while DNC does not.

Future work includes developing a new strategy that allows the algorithm to work in the presence of multiple concurrent fault and repair events. A practical tool based on the Internet management protocol, SNMP (Simple Network Management Protocol) is currently being developed. This tool should present a Web-based interface that allows any user to obtain network connectivity information from any network node. Effective testing strategies for WAN links must be developed, possibly using artificial intelligence tools.

References

[1] G. Masson, D. Blough, and G. Sullivan, "System Diagnosis," in *Fault-Tolerant Computer System Design,* ed. D.K. Pradhan, Prentice-Hall, 1996.

[2] F. Preparata, G. Metze, and R.T. Chien, "On The Connection Assignment Problem of Diagnosable Systems," *IEEE Transactions on Electronic Computers*, Vol. 16, pp. 848-854, 1968.

[3] E.P. Duarte Jr., G. Mansfield, T. Nanya, and S. Noguchi, "Non-Broadcast Network Fault Monitoring Based on System-Level Diagnosis," *Proc. IEEE/IFIP IM'97*, pp.597-609, San Diego, May 1997.

[4] E.P. Duarte Jr., "Um algoritmo para Diagnóstico de Redes de Topologia Arbitrária," *in portuguese*, In Proceedings of the *1st SBC Workshop on and Fault Tolerance*, SBC-WTF'1, pp. 50-55, Porto Alegre, Brazil, 1998.

[5] E. P. Duarte Jr., and T. Nanya, "A Hierarchical Adaptive Distributed System-Level Diagnosis Algorithm," *IEEE Transactions on Computers*, Vol. 47, pp. 34-45, No. 1, Jan 1998.

[6] A. Bagchi, and S.L. Hakimi, 'An Optimal Algorithm for Distributed System-Level Diagnosis," *Proc.* 21^{st} *Fault Tolerant Computing Symp.*, June, 1991.

[7] M. Stahl, R. Buskens, and R. Bianchini, "Simulation of the Adapt On-Line Diagnosis Algorithm for General Topology Networks," *Proc. IEEE* 11^{th} *Symp. Reliable Distributed Systems*, October 1992.

[8] S.Rangarajan, A.T. Dahbura, and E.A. Ziegler, 'A Distributed System-Level Diagnosis Algorithm for Arbitrary Network Topologies," *IEEE Transactions on Computers*, Vol.44, pp. 312-333, 1995.

[9] J. I. Siqueira, E. Fabris, E. P. Duarte Jr., "A Token Based Testing Strategy for Non-Broadcast Network Diagnosis", *1st IEEE Latin American Test Workshop*, pp. 166-171, Rio de Janeiro, 2000.

[10] P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994.

[11] M.H. MacDougall, *Simulating Computer Systems: Techniques and Tools*, The MIT Press, Cambridge, MA, 1987.

[12] E.P. Duarte Jr., and G.O. Mattos, "Diagnóstico em Redes de Topologia Arbitrária: Um Algoritmo Baseado em Inundação de Mensagens", *in portuguese*, In Proceedings of the *2nd SBC Workshop on Test and Fault Tolerance*, SBC-WTF'2, pp. 82-87, Curitiba, Brazil, 2000.

[13] E.P. Duarte Jr., and J.M.A.P. Cestari, "O Agente Chines para Diagnstico de Redes de Topologia Arbitrária" *in portuguese*, In Proceedings of the *2nd SBC Workshop on Test and Fault Tolerance*, SBC-WTF'2, pp. 88-93, Curitiba, Brazil, 2000.

Simulação de Sistemas Distribuídos em Cenários com Defeitos

Renata de Moraes Trindade[1] Marinho Pilla Barcellos[1,2] Ingrid Jansch-Pôrto[1]

trindade@inf.ufrgs.br marinho@exatas.unisinos.br ingrid@inf.ufrgs.br

[1] Pós-Graduação em Ciência da Computação – UFRGS
[2] Programa Interdisciplinar de Pós-Graduação em Computação Aplicada – UNISINOS

1. Introdução

Um sistema distribuído (SD) é definido como um conjunto de vários computadores autônomos que não compartilham relógio nem memória, estão separados geograficamente e trocam informações através de mensagens sobre a rede de comunicação que os interliga [1]. Um SD deve lidar com várias fontes de indeterminismo: programas são executados em velocidades diferentes, utilizando um número qualquer de processadores, que estão distribuídos sobre uma topologia de rede desconhecida e com tempos de propagação e ordenamento de mensagens imprevisíveis [2]. Portanto, desenvolver protocolos distribuídos é uma atividade complexa. Além disso, os mecanismos para detectar e mascarar defeitos representam tipicamente uma grande parcela do desenvolvimento desses protocolos. Assim, testar, depurar e validar tais protocolos são tarefas que ainda apresentam desafios.

Simuladores contribuem para a solução destes problemas, pois oferecem um ambiente controlado para validar o comportamento de protocolos existentes, fornecem infra-estrutura para desenvolvimento de novos protocolos e oportunizam o estudo de suas interações. Entretanto, devido à complexidade dos sistemas distribuídos, ainda existem poucas iniciativas na área de simulação nesse contexto, notadamente com suporte adequado para o tratamento de situações com defeitos.

O Network Simulator versão 2 – ***ns***-2 [3] é uma ferramenta acadêmica iniciada em 1989 na Universidade da Califórnia em Berkeley. Em 1995, seu desenvolvimento foi apoiado pelo DARPA através do projeto VINT (Virtual InterNetwork Testbed [4]), envolvendo USC/ISI, Xerox PARC, LBNL, e UC Berkeley. Esta ferramenta tem sido amplamente utilizada por pesquisadores da área de redes, para estudo de políticas de filas, controle de congestionamento, desempenho de protocolos, protocolos de *multicast* confiável, entre outros. Além disso, tem código aberto e pode ser estendido para aplicações específicas.

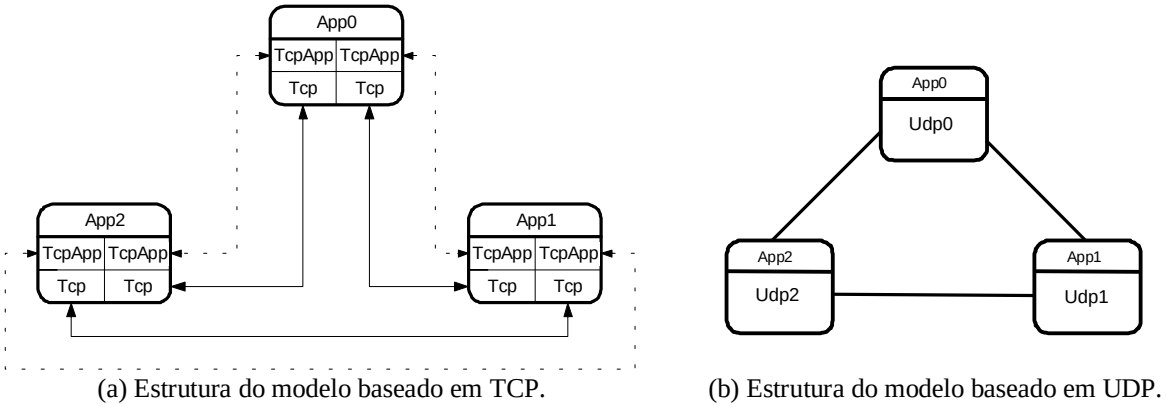
Este trabalho visa explorar a viabilidade de uso do ***ns*** como ambiente para simulação de SDs, particularmente em cenários sujeitos à ocorrência de defeitos. No presente artigo, são tratados apenas defeitos de colapso (*crash failures*); mas prevê-se a extensão da solução para outros tipos de defeitos.

2. Simulação de sistemas distribuídos no ns

Neste trabalho, dependendo do protocolo de transporte utilizado, são considerados dois modelos distintos para simulação, baseados: no protocolo TCP (confiável) ou no protocolo UDP (não confiável). Essas duas versões, modeladas em um sistema assíncrono [2], serão explicadas a seguir. Por premissa, cada nodo contém somente uma aplicação (processo que está na camada de aplicação).

No **modelo baseado em TCP**, o transporte de dados é feito por agentes TCP. Em um sistema com *n* nodos, cada nodo tem até *n-1* agentes TCP: um para cada conexão. Cada agente TCP é associado a um componente da camada de aplicação, chamado **`TcpApp`**, que é encarregado de fazer a emulação da transmissão de dados entre as aplicações. Além disso, em cada instância da aplicação (**`Appn`**), é mantida uma tabela de conexões, onde são armazenadas

todas as aplicações às quais ela está conectada, juntamente com a **`TcpApp`** encarregada de transmitir dados para a aplicação-destino. A Figura 1(a) mostra três nodos que trocam dados entre si, a distribuição dos agentes e aplicações dentro de cada nodo, e suas conexões. **`Appn`** representa um processo da aplicação distribuída, **`TcpApp`** é a classe encarregada de enviar os dados e TCP é o agente de transporte.



(a) Estrutura do modelo baseado em TCP. (b) Estrutura do modelo baseado em UDP.

Figura 2: Estruturas dos modelos.

Nesse modelo, a comunicação é bidirecional e feita por meio de *unicast*. O agente TCP usado é um agente bidirecional do ***ns*** chamado **`FullTcp`**. Para a transmissão de dados, estes percorrem o seguinte fluxo: a aplicação (por exemplo, **`App0`**) passa os dados à sua **`TcpApp`**, que os encaminha ao agente TCP. Este, por sua vez, os envia para o agente TCP da aplicação a qual está conectado. Quando um agente TCP recebe dados, ele os entrega à **`TcpApp`**, que os repassa à aplicação destinatária (**`App1`**).

O **modelo baseado em UDP** emprega agentes UDP para o transporte de dados; cada nodo tem somente um agente UDP que envia dados para destinatários. A comunicação não é confiável, mas tem a vantagem de oferecer também *multicast*, ao contrário do modelo baseado em TCP. A Figura 1(b) mostra como o agente UDP e a aplicação estão distribuídos dentro do nodo, nesse modelo.

No caso de *multicast*, cada agente UDP envia dados para o endereço de um grupo. Na Figura 1(b), por exemplo, **`App0`** (através de seu agente **`Udp0`**) envia dados para um grupo que é composto por **`Udp1`** e **`Udp2`**. O fluxo de dados é o seguinte: a aplicação (**`App0`**) passa os dados para seu agente UDP (**`Udp0`**) que, por sua vez, os envia para os agentes UDP de seu grupo-destino. Quando um agente UDP recebe dados, repassa-os para sua aplicação.

3. Simulações em cenários com defeitos

Para as atividades de simulação de defeitos em SDs, foram consideradas as seguintes definições: ocorre um defeito de **colapso (*crash*) em um processo**, quando este pára prematuramente e não faz mais nada a partir desse ponto, portanto não produz resultados incorretos. Defeito de **colapso em *link*** ocorre quando ele pára de transportar mensagens [5].

Verificou-se que o ***ns*** não oferece possibilidade direta para simular defeitos de nodo, mas apenas defeitos de *link*. Para estes, há um comando específico, cujos parâmetros definem o tipo de operação (rompimento/restabelecimento do *link*), identificação do *link* e o momento da ocorrência do defeito. Pode-se configurar defeitos permanentes ou temporários. O manual do ***ns*** indica a possibilidade de simular defeitos de nodos, utilizando o comando para defeito de *link*, identificando um nodo ao invés do *link*, dentre os parâmetros. Entretanto, a semântica desse comando indica que todos os *links* desse nodo serão "rompidos", o que impede o controle sobre o estado interno do nodo e de seus componentes. Isto é especialmente

problemático em variações tais como *crash-recover* ou casos de perda de conteúdo (*amnesia*).

Para simulação de defeitos de colapso em processos, foi criado um mecanismo que possibilita a expansão do modelo de defeitos. Para as versões TCP e UDP, foram desenvolvidas estruturas diferenciadas, apresentadas a seguir.

No modelo baseado em TCP, ao enviar dados, uma aplicação os entrega para **`TcpApp`**, que os repassa ao **`FullTcp`**, o qual os envia ao **`TcpApp`** remoto. Na recepção, o dado percorre o caminho inverso até atingir a aplicação (**`FullTcp`** → **`TcpApp`** → **`App`**). Portanto, o elemento mais baixo nas camadas atravessadas pelos dados, nesse modelo, é o agente **`FullTcp`**. A partir desta constatação, decidiu-se desenvolver uma classe derivada da classe **`FullTcpAgent`**, na qual os métodos que fazem o envio e a recepção dos dados foram reescritos para simular defeitos (**`sendmsg`** e **`recv`**, respectivamente).

A estrutura para simulação de defeitos, na versão baseada em UDP, diferencia-se devido aos elementos componentes dos nodos. Em cada nodo UDP, existe uma aplicação e somente um agente UDP: esse agente UDP é de uma nova classe – **`UdpData`** – derivada da classe **`UdpAgent`**. Sendo assim, o elemento da camada mais inferior é o agente **`UdpData`**. Assim, foi criada uma classe denominada **`UdpDataFail`**, derivada de **`UdpData`**, na qual os métodos de envio e recepção dos dados (**`send`** e **`recv)`** foram rescritos.

A Figura 2 apresenta as classes **`UdpDataFail`**, **`FullTcpFail`** e sua hierarquia em UML (Unified Modeling Language [6]), como será explicada a seguir. O método **`command`** é chamado quando é executado um comando do *script* da simulação OTcl para uma determinada classe. Por exemplo, assumindo-se que "**`tcp`**" é uma instância da classe **`FullTcpFail`**, o comando **`$tcp crash-failure`** atribui *true* à variável **`crash`**, indicando que existe defeito de colapso nesse agente/aplicação. No método **`recv`**, as ações dependem do valor da variável **`crash.`** Em *false*, indica que o nodo não está com defeito, nesse caso, é realizada a chamada para o método **`recv`** de sua superclasse. Se houver defeito no nodo (se **`crash=true`**), o pacote recebido será eliminado. Portanto, na recepção de dados, essa função verifica o valor de **`crash`** e, dependendo desse valor, passa o pacote para a classe **`FullTcpAgent`** (ou **`UdpAgent`**) ou o elimina. No método **`sendmsg/send`**, o procedimento é semelhante ao feito no método **`recv.`** Se o valor da variável **`crash`** for igual a *false*, então é chamado o método **`sendmsg/send`** de sua superclasse. Senão, se o agente está com defeito (valor de **`crash=true`**) os dados não são enviados para o agente **`FullTcp`** e, portanto, a aplicação-destino não irá recebê-los.

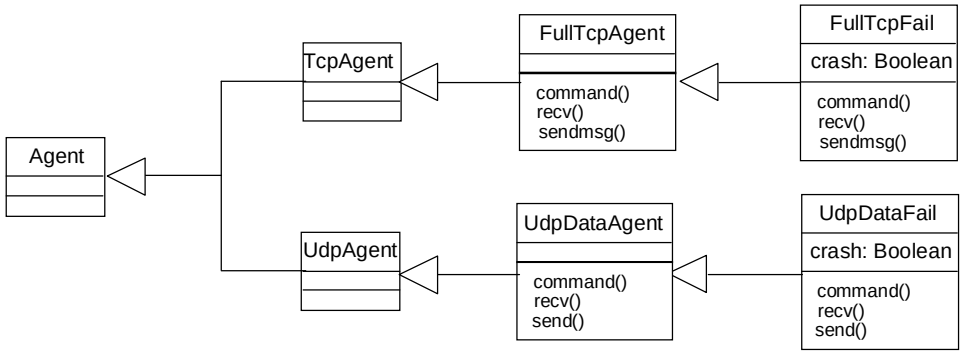


Figura 2: Modelo simplificado de classes.

4. Estudo de casos

Para exemplificar os modelos de simulação apresentados, foram implementados, no ***ns***, os seguintes algoritmos distribuídos baseados nos algoritmos propostos por Lynch[2]: (a) **algoritmo de eleição em anel assíncrono**: desenvolvido para solucionar o problema de

eleição de um processo líder em um anel unidirecional em uma rede assíncrona. Esse algoritmo é chamado *AsynchLCR*; (b) **algoritmo de inundação assíncrono**: soluciona o problema de eleição de líder em qualquer tipo de topologia; (c) **algoritmo para solucionar do problema de concordância em sistemas assíncronos**: atinge concordância mesmo na presença de defeitos de colapso em processos (chamado de *BenOr*).

Estes algoritmos foram implementados utilizando ambos modelos baseados em TCP e UDP. Nota-se que a estrutura básica para simulação de SDs nos diferentes casos é semelhante, os casos diferenciam-se pelos algoritmos distribuídos escolhidos.

Foi desenvolvido um arquivo de *script* de simulação que serve como padrão para desenvolvimento de qualquer aplicação que utiliza TCP, em qualquer tipo de topologia física. Este *script* foi feito para facilitar a criação e o estabelecimento de conexões dos agentes e a criação de aplicações contidas nos nodos do sistema. Com o uso desse arquivo, é possível criar todos os nodos, agentes e conexões do experimento. Além desse arquivo *script* padrão, foi criado outro arquivo que facilita a simulação de defeitos. Um de seus principais procedimentos é o **`crash_failure <n_node> <failure_node>`**, onde **`<n_node>`** é a quantidade de nodos e **`<failure_node>`** é o número do nodo onde deve ocorrer o defeito. Com esse procedimento, são feitas chamadas ao **`comando $tcp crash-failure`** para todos os agentes **`FullTcpFail`** do nodo **`<failure_node>`**, que é responsável por simular defeito de colapso em toda a aplicação do nodo.

A simulação desses algoritmos no ***ns*** produziu resultados funcionalmente corretos. Mas apenas o algoritmo *BenOr* prevê a ocorrência de defeitos. Nesta implementação, foram realizados vários experimentos funcionais, inserindo-se defeitos em vários momentos da simulação. Foi constatado que, também neste cenário com defeitos, respeitadas as hipóteses adotadas no desenvolvimento do algoritmo, os resultados gerados também são corretos.

5. Conclusões

O presente trabalho tem como objetivo estudar a adequação do simulador de redes ***ns*** como instrumento para a verificação e desenvolvimento de protocolos em SDs. Além da riqueza de funções e aplicações, a ferramenta foi escolhida por suas características acadêmicas: código aberto e possibilitar o desenvolvimento de extensões.

Foram identificados os mecanismos necessários para a simulação de sistemas com topologias variadas; comenta-se a implementação de uma rede totalmente interconectada, cuja rede de comunicação permite a opção entre protocolos UDP e TCP. Foi proposta uma solução para simulação de defeitos de colapso, seu desenvolvimento e resultados obtidos com casos exemplo. Além de testes adicionais em diferentes arquiteturas e variações de implementação, o prosseguimento deste trabalho inclui a expansão do modelo de defeitos suportado pelo simulador. O código-fonte das alterações no ***ns*** e a implementação dos algoritmos será disponibilizada para *download* e livremente distribuída segundo a licença GPL.

Referências Bibliográficas

[1] Jalote, P. Fault Tolerance in Distributed Systems. New Jersey: Prentice Hall, 1994.

[2] Lynch, N. Distributed Algorithms. San Francisco: Morgan Kaufmann Publ., 1996.

[3] Network Simulator – NS-2: http://www.isi.edu/nsnam/ns/index.html.

[4] VINT – Virtual InterNetwork Testbed: http://www.isi.edu/nsnam/ vint/index.html.

[5] Hadzilacos, V.; Toueg, S. Fault-Tolerant Broadcasts and Related Problems. Distributed Systems. New York: Addison-Wesley, 1995. chap.5, p.99-102.

[6] Booch, G; Rumbaugh, J; Jacobson, I. The Unified Modeling Language User Guide. New York: Addison-Wesley, 1999.