MENU

PESQUISA

Workshop - WTF

21º SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES

ANAIS

SBRC 2003

Natal, RN

19 a 23 de maio de 2003

Workshop de Testes e Tolerância a Falhas

Contra-Capa

Mensagem do Coordenador

Comitê

Revisores

Índice

SBRC2003

21º Simpósio Brasileiro de Redes de Computadores
Natal, RN
19 a 23 de Maio de 2003

IV Workshop de Testes e Tolerância a Falhas (WTF)

Editores
Francisco Vilar Brasileiro
Walfredo Cirne

Realização
Departamento de Informática e Matemática Aplicada - DIMAp
Universidade Federal do Rio Grande do Norte – UFRN
Universidade Potiguar - UnP

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

Cópias adicionais:

Departamento de Informática e Matemática Aplicada - DIMAp
Universidade Federal do Rio Grande do Norte - UFRN
Centro de Ciências Exatas e da Terra - CCET
Campus Universitário, Lagoa Nova
Natal - RN - 59072-970
Tel: (84) 215-3814 - Fax: (84) 215-3813

Capa: Carlos Soares
Editoração: Carlos Gustavo Araújo da Rocha

ISBN: 85-88442-49-3

Divisão de Serviços Técnicos

Catalogação da Publicação na Fonte. UFRN / Biblioteca Central Zila Mamede

Simpósio Brasileiro de Redes de Computadores (21. : 2003 : Natal, RN).
IV workshop de testes e tolerância a falhas (WTF) / Editores Francisco Vilar Brasileiro, Walfredo Cirne. – Natal, RN : UFRN/DIMAp : UNP, 2003.
135 p.

1. Redes de computadores - Congressos. 2. Sistemas distribuidos - Congressos. 4. Informática - Congressos. 5. Testes e tolerância a falhas - Congressos. I. Brasileiro, Francisco Vilar, II. Cirne, Walfredo. III. Título.

RN/UF/BCZ M CDU 004.7(061.3)

Prefácio

O Workshop de Testes e Tolerância a Falhas foi criado como uma forma de congregar profissionais da área de testes e tolerância a falhas nos anos em que o Simpósio de Computadores Tolerantes a Falhas (SCTF) não é realizado. O principal objetivo deste evento é promover discussões e troca de idéias sobre trabalhos e projetos teóricos e práticos em andamento no país, como também promover uma maior integração dos trabalhos na área desenvolvidos pela academia e pela indústria.

A partir de 2003, o SCTF será substituído pelo *Latin-American Symposium on Dependable Computing* (LADC), um evento mais amplo e internacional, a ser realizado a cada dois anos. Por conta disso, a comunidade reunida no último WTF, realizado em Búzios junto com o Simpósio Brasileiro de Redes de Computadores (SBRC'2002), decidiu transformar o WTF em um evento anual, sempre atrelando-o a um outro evento de porte nacional que tenha alguma relação com os temas discutidos no WTF. Para 2003 foi decidido realizar o evento em conjunto com o SBRC'2003, uma vez que o mesmo congrega diversos profissionais que trabalham em temas relacionados tanto com redes e sistemas distribuídos quanto com testes e tolerância a falhas.

Nesta edição teremos um conjunto de 8 artigos selecionados pelo comitê de programa, de um total de 15 artigos submetidos. Além disso, teremos 2 tutoriais: "Detectores de Falhas em Sistemas Assíncronos" e "*Middleware* para Redes de Sensores Sem Fio". Complementará a programação do workshop uma sessão aberta para discussão de trabalhos em andamento sendo desenvolvidos pela comunidade. Esperamos que as apresentações gerem discussões interessantes e ativas, vindo a contribuir para o desenvolvimento da área de testes e tolerância a falhas no Brasil.

Aproveitamos para agradecer aos organizadores do SBRC'2003 pelo excelente trabalho sendo realizado na organização do evento. Agradecemos também aos autores pela submissão de seus trabalhos, bem como aos membros do comitê de programa do IV WTF pelo trabalho criterioso de revisão e seleção dos artigos.

Francisco Vilar Brasileiro
Walfredo Cirne
Editores, em nome da Comissão de Programa do IV WTF

Comitê de programa

Avelino F. Zorzo (PUCRS)
Cecilia M. F. Rubira (UNICAMP)
Eliane Martins (UNICAMP)
Elias P. Duarte Jr. (UFPR)
Fabiola Greve (UFBA)
Francisco Brasileiro (UFCG)
Ingrid J. Porto (UFRGS)
Joni S. Fraga (UFSC)
Marinho P. Barcellos (UNISINOS)
Patricia Machado (UFCG)
Raimundo Macedo (UFBA)
Rogerio De Lemos (University of Kent, UK)
Sérgio Vale Aguiar Campos (UFMG)
Silvia R. Vergilio (UFPR)
Taisy R. Weber (UFRGS)
Walfredo Cirne (UFCG)

Avaliadores dos artigos

Alysson Neves Bessani
Andreas Kiefer
Avelino F. Zorzo
Cecilia M. F. Rubira
Egon Hilgenstieler
Eliane Martins
Elias P. Duarte Jr.
Fabíola Greve
Francisco V. Brasileiro
Ingrid J. Porto
Joni S. Fraga
Lau Cheuk Lung
Luis C. E. De Bona
Marinho P. Barcellos
Patricia Machado
Raimundo Macedo
Raissa Medeiros
Rogerio De Lemos
Sérgio Vale Aguiar Campos
Silvia R. Vergílio
Taisy R. Weber
Thiago Mello
Walfredo Cirne

Índice

Tutorial 1

Detectores de Falhas em Sistemas Assíncronos
Lívia M. R. Sampaio, Andrey E. M. Brito, Ely W. A. de Oliveira (UFCG).....3

Sessão Técnica 1: Algoritmos

Difusão Atômica com Suporte à Perda de Mensagens
Fabíola G. P. Greve (UFBA).....33

Guaranteeing Fault Tolerance through Scheduling on a CAN Bus
M. P. Oliveira, A. O. Fernandes, S. V. A. Campos A. L. A. P. Zuquim (UFMG).....43

Sessão Técnica 2: Verificação Formal

Verificação Formal de Protocolos TDMA quanto a Características de Tempo Real e Tolerância a Falhas
Alberto R. Beckler, Sérgio V. A. Campos (UFMG).....53

Proposta de uma Abordagem para a Verificação Formal de Sistemas Distribuídos Baseados em Objetos
Osmar M. dos Santos, Fernando L. Dotti (PUCRS).....61

Sessão Técnica 3: Arquiteturas

The MARES Platform: Support for Transactional and Fault-tolerant Execution of Mobile Agent-based Applications
Flávio M. A. Silva, Raimundo J. A. Macêdo, Ana V. P. Freitas (UFBA).....71

Uma Arquitetura Altamente Disponível Aplicada a Sistemas de Controle Embutidos de Tempo Real
Cesar Ida, Taisy Weber (UFRGS).....79

Tutorial 2

Middleware para Redes de Sensores Sem Fio
Antonio A. F. Loureiro, Linnyer B. Ruiz, Fernanda P. Franciscani, Rainer R. P. Couto, José Marcos S. Nogueira (UFMG).....89

Sessão Técnica 4: *Middleware*

Tolerância a Faltas Adaptativa em um Modelo de Componentes
Fábio Favarim, Joni Fraga, Frank Siqueira (UFSC).....119

Uma Implementação Tolerante a Faltas e Transparente da Plataforma J2EE
André A. Costa, Francisco V. Brasileiro (UFCG).....127

Detectores de Falhas em Sistemas Assíncronos

Lívia M. R. Sampaio, Andrey E. M. Brito, Ely W. A. de Oliveira

Universidade Federal da Paraíba
Departamento de Engenharia Elétrica
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
{livia,andrey,ely}@dsc.ufcg.edu.br

Resumo

Um sistema distribuído é comumente definido como um conjunto de computadores que se comunicam através de uma rede de comunicação para realizar, de forma coordenada, atividades em comum. Como em qualquer ambiente computacional, falhas podem ocorrer em um sistema distribuído e se não forem detectadas e tratadas, podem comprometer o sucesso das atividades realizadas pelo sistema. Vários mecanismos têm sido desenvolvidos no sentido de permitir que tais sistemas funcionem adequadamente, apesar da ocorrência de falhas. Detectores de falhas são componentes responsáveis por prover informação sobre processos do sistema que sofreram falhas, sendo assim, importantes construções para a definição de mecanismos para tolerância a falhas. Tais componentes abstraem o processo de detecção de falhas dos mecanismos que utilizam este serviço. Uma abordagem bastante discutida atualmente refere-se aos detectores de falhas não confiáveis, os quais enfocam o problema de tolerar falhas em sistemas distribuídos assíncronos. Este tutorial tem por objetivo promover o aprendizado do seu público alvo na área de detectores de falhas, mais especificamente, detectores de falhas não confiáveis.

1 Introdução

Um sistema distribuído é comumente definido como um conjunto de computadores que se comunicam através de uma rede de comunicação para realizar, de forma coordenada, atividades em comum. Como em qualquer ambiente computacional, falhas podem ocorrer em um sistema distribuído e se não forem detectadas e tratadas, podem comprometer o sucesso de tais atividades. Em nossa sociedade, a crescente demanda pela informatização de

sistemas complexos e altamente confiáveis tem impulsionado o desenvolvimento dos sistemas distribuídos e sua aplicação em diversos setores: hospitais, bancos, comércio eletrônico, bolsas de valores, aviação. A medida que aumenta a dependência em relação ao correto funcionamento de tais sistemas, torna-se indispensável torná-los tolerantes a faltas.

Considere a Internet, mais especificamente, um serviço WWW distribuído de comércio eletrônico de uma grande empresa. Imagine um usuário deste serviço, preenchendo um formulário HTML na intenção de comprar algum produto desta empresa. O usuário envia o formulário com os dados da compra e espera uma mensagem de confirmação do servidor WEB que efetive, ou não, a transação iniciada com a empresa virtual. O servidor WEB recebe a requisição do usuário, realiza o processamento dos dados e deve enviar uma confirmação para o cliente, indicando que a transação foi efetivada com sucesso. Se o servidor WEB falhar em algum ponto desta transação, por exemplo, antes de enviar a confirmação para o cliente, isto pode gerar inconsistências já que a transação foi efetivada e o cliente vai achar o contrário, podendo repetir a compra (e ser surpreendido ao receber a conta de seu cartão de crédito...). No melhor caso, se o servidor WEB falhar antes da transação ser iniciada, o usuário pode desistir do serviço. Nesse caso, é necessário tolerar faltas do servidor WEB. Isto pode ser feito através do uso de redundância [1]. Uma forma de redundância bastante utilizada em tais situações é a replicação primário-cópia (ou replicação passiva) [2]. Este mecanismo requer a existência de um serviço de detecção que detecte falhas do servidor primário e ative o servidor cópia.

Detecção de falhas também pode ser usado como um construtor para outros serviços importantes em sistemas distribuídos tolerantes a faltas, tal qual o consenso [3, 4]. Os protocolos de consenso permitem que vários processos do sistema distribuído atinjam decisões em comum. Uma aplicação do consenso refere-se aos protocolos de confirmação atômica (*atomic commit*) para manipulação de bancos de dados distribuídos. De maneira informal, o protocolo de confirmação atômica funciona da seguinte forma: todos os servidores do banco de dados devem concordar se uma transação será confirmada ou abortada, e no caso de algum servidor optar por abortar a transação, todos os outros servidores devem abortar também. O problema da confirmação atômica é mais difícil de resolver do que o problema do consenso devido à prioridade em relação à decisão de abortar uma transação. Conseqüentemente, qualquer resultado indicando impossibilidade de resolver o consenso pode ser aplicado também ao problema da confirmação atômica [5].

Entendemos que a base para o funcionamento de um protocolo de consenso está na possibilidade de responder se um determinado processo falhou ou não. Isso depende do modelo adotado pelo sistema onde se deseja resolver o consenso. Nos sistemas síncronos, onde é conhecido o limite máximo de tempo para transmissão de mensagens entre os processos, a detecção de processos falhos pode limitar-se ao uso de *timeouts* (limites de tempo de comunicação), considerando falhos os processos que não enviam uma mensagem até o limite máximo de tempo estabelecido. Já em sistemas assíncronos, como a Internet e a maior parte das redes locais, onde tais limites não existem, outras técnicas se fazem necessárias, uma vez que o recebimento de mensagens pode ser atrasado não por falhas, mas por sobrecarga na rede ou lentidão nos processos. Nesses casos, simplesmente basear-se em *timeouts* pode levar ao mau funcionamento do sistema como um todo, uma vez que processos corretos poderão ser interpretados como falhos.

O modelo assíncrono traz algumas vantagens sobre o síncrono, por possuir uma semântica simples e permitir a construção de aplicações com maior portabilidade do que aquelas baseadas em limites de tempo para comunicação entre os processos. Por esse motivo, existe um grande número de aplicações desenvolvidas considerando este modelo. Entretanto, o trabalho apresentado por Fischer *et al.* [6] prova ser impossível, em um sistema assíncrono, criar um protocolo que garanta a obtenção de consenso, de forma determinista. A razão está no fato de que, em tais sistemas é impossível distinguir se um processo não responde mensagens por ter falhado ou por estar apenas muito lento. Esta dificuldade tem motivado muitas pesquisas e levado à criação de novos modelos de sistemas, como o de sistemas parcialmente síncronos [7, 8] e sistemas assíncronos temporizado [9]. Entretanto, é o modelo de sistema assíncrono com detectores de falhas não confiáveis [10] que ultimamente tem recebido mais atenção da literatura.

Este trabalho tem por objetivo explorar o conceito, aplicações e questões relacionadas a detectores de falhas não confiáveis. Nesse sentido apresentamos na Seção 2 o conceito de detectores de falhas não confiáveis. Já na Seção 3, descrevemos modelos e implementações de detectores de falhas. As Seções 4, 5 discutem medidas de qualidade de serviço para serviços de detecção e qual o impacto destes sobre o desempenho dos protocolos de alto nível que os utiliza. Finalmente, a Seção 6 apresenta algumas considereções finais e algumas motivações para futuras pesquisas sobre detectores de falhas não confiáveis.

2 Detectores de Falhas não Confiáveis

Inicialmente apresentados por Chandra e Toueg [10], os detectores de falhas não confiáveis são componentes que servem de oráculos ao sistema, informando os processos falhos. Tais componentes fornecem uma abstração do mecanismo de detecção de falhas, que permite a simplificação dos algoritmos distribuídos, resumindo a detecção a simples consultas ao detector. Originalmente, considerou-se falhas do tipo *crash*, onde o processo interrompe definitivamente sua execução. No entanto, vários trabalhos, como será visto mais adiante, ampliam o escopo de falhas com as quais os detectores são capazes de lidar.

Os detectores são geralmente organizados em módulos distribuídos pelo sistema, que monitoram o estado de subgrupos de processos e mantêm uma lista dos processos suspeitos de terem falhado. Cada processo tem acesso a um módulo local, do qual pode consultar o estado dos demais processos. Os detectores de falhas são chamados de não confiáveis porque podem cometer erros e eventualmente adicionar nesta lista processos corretos. Após certo tempo, se o módulo do detector descobrir que errou ao suspeitar de um determinado processo, este pode ser removido da lista de suspeitos.

2.1 Propriedades e Classes dos Detectores de Falhas

O trabalho de Chandra e Toueg [10] define e classifica os detectores de falhas através de propriedades abstratas ao invés de fornecer implementações específicas. Aspectos de implementação não são levados em consideração neste trabalho. Isso permite que o projeto de aplicações distribuídas seja baseado apenas nessas propriedades, sem se comprometer com

características de rede ou de *software* envolvidas na implementação. O mesmo se aplica à prova sobre a corretude destas aplicações.

As propriedades usadas para classificar os detectores são duas: abrangência e exatidão. Abrangência diz respeito a quantos módulos conseguirão detectar a falha de um processo. Exatidão diz respeito aos erros que um detector pode cometer. Existem dois níveis de abrangência e quatro níveis de exatidão, os quais serão descritos a seguir.

- abrangência fraca: requer que após um tempo, a falha em um processo seja detectada por no mínimo, um módulo de detecção
- abrangência forte: requer que após um tempo, a falha seja detectada por todos os módulos de detecção.
- exatidão forte: requer que os processos não sejam suspeitados por nenhum módulo de detecção antes que falhem.
- exatidão fraca: requer que pelo menos um processo não seja suspeitado antes que falhe;
- exatidão forte após um tempo: requer que a partir de algum tempo, a exatidão forte seja atendida;
- exatidão fraca após um tempo: requer que a partir de algum tempo, a exatidão fraca seja atendida.

Através da combinação dessas propriedades, são definidas oito classes de detectores de falhas, como visto na Tabela 1 [10].

Abrangência	*Exatidão*			
	Forte	Fraca	Forte após um certo tempo	Fraca após um certo tempo
Forte	P	S	$\diamond P$	$\diamond S$
Fraca	Q	W	$\diamond Q$	$\diamond W$

Tabela 1: Classes de detectores de falhas

A classe P atende as propriedades de abrangência forte e exatidão forte, sendo assim, a classe com semântica mais forte de todas. Os detectores desta classe devem ser capazes de suspeitar apenas dos processos que realmente falharem e, todos os seus módulos devem suspeitar de um processo falho, dentro de algum tempo. A classe $\diamond W$ atende as propriedades de abrangência fraca e exatidão fraca após um tempo, sendo considerada como a classe com semântica mais fraca de todas. Os detectores desta classe devem garantir que após um tempo, pelo menos um processo não é suspeitado antes que falhe, além disso, se um processo falhar, ele será suspeitado por pelo menos um módulo do detector. A classe $\diamond S$ é a de semântica mais fraca que permite a solução de consenso de forma determinística em ambientes assíncronos, conforme demonstrado por Chandra *et al.* [11]. Por esse motivo, a maioria das implementações existentes de detectores de falhas é desta classe.

Um outro conceito apresentado por Chandra e Toueg [10] é o de reducibilidade. Para fins de análise, as oito classes podem ser reduzidas a apenas quatro, assumindo que um algoritmo

pode ser empregado para fazer os detectores de uma classe comportarem-se como os de outra. Por exemplo, se considerarmos um sistema X que requer a presença de um detector da classe D', mas só existe disponível um detector da classe D, poderíamos utilizar um algoritmo A que fizesse o detector D se comportar como um detector D'. Desta forma, emularíamos D' para atender ao sistema X (ver Figura 1). Quando isso ocorre, é dito que D' é redutível para D. Se D também for redutível para D', os dois são considerados equivalentes.

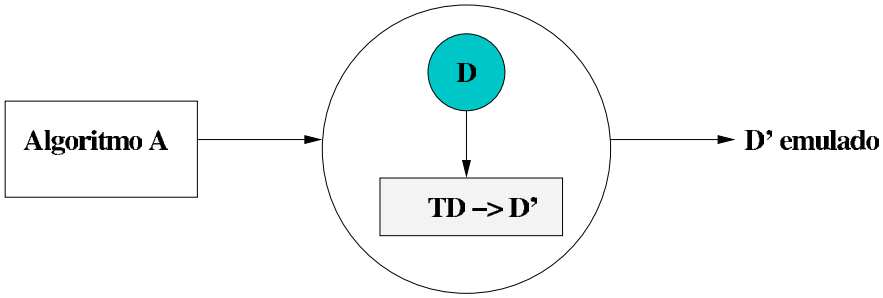


Figura 1: Emulação de detectores de falhas

As quatro classes de detectores com abrangência fraca (Q, W, $\diamond Q$ e $\diamond W$) são equivalentes às classes com abrangência forte (P, S, $\diamond P$ e $\diamond S$). Note que os detectores com abrangência forte podem ser empregados em ambientes que esperam os detectores de abrangência fraca. Isto porque um serviço com abrangência forte é mais completo (abrangente) do que um serviço com abrangência fraca, sendo possível emular os respectivos detectores de falhas sem dificuldades.

De fato, os detectores com abrangência fraca também podem emular aqueles com abrangência forte. Para tal, todos os seus módulos, e não apenas um, devem detectar quando um processo falhou. Nesse caso, basta empregar um algoritmo que faça cada módulo enviar aos demais, a lista dos seus processos suspeitos. Qualquer módulo, ao receber as mensagens dos demais, atualiza a sua lista de suspeitos com as informações recebidas dos demais módulos. Esta atividade é repetida indefinidamente. Se alguma mensagem for perdida, outras serão enviadas em seguida, até que todos os módulos suspeitem de um processo falho. Se uma suspeita for errônea, o processo sairá das listas de suspeitos de todos os módulos e não será mais propagado nas mensagens. Desta forma, se um módulo detectar a falha de um processo, conseqüentemente, todos os demais o farão, em algum momento, exatamente o que caracteriza a abrangência forte. Este algoritmo permite que os detectores com abrangência fraca possam ser empregados em ambientes que esperam os detectores com abrangência forte. Sendo assim, os detectores das classes P, S, $\diamond P$ e $\diamond S$ são redutíveis para as classes Q, W, $\diamond Q$ e $\diamond W$, respectivamente. Por causa desta equivalência, a literatura normalmente referencia apenas os detectores das classes P, S, $\diamond P$ e $\diamond S$.

2.2 Exatidão γ

As propriedades de abrangência e exatidão, definidas no trabalho de Chandra e Toueg [10] se referem a todos os processos do sistema monitorados pelo detector de falhas. O formalismo não se aplica a sistemas sujeitos a particionamento na rede. Durante o particionamento de

uma rede, a probabilidade de que as propriedades de exatidão sejam mantidas é nula. Os processos que fizerem parte da partição A serão, certamente, suspeitados pelos processos da partição B e vice-versa. Neste caso, não haverá nenhum processo correto livre de falsas suspeitas por parte dos demais e isso torna o detector de falhas impossível de ser classificado segundo as oito classes originais descritas anteriormente.

Guerraoui e Schiper [12] estenderam o formalismo original de Chandra e Toueg [10], considerando propriedades de exatidão que podem ser mantidas por subgrupos de processos, em caso de particionamento na rede. Essas propriedades são exatidão forte γ e exatidão fraca γ. A exatidão forte γ exige que nenhum processo do subgrupo γ seja suspeitado por outro processo de γ, antes de falhar. A exatidão fraca γ exige que ao menos um processo, não necessariamente do subgrupo γ, não seja suspeitado por outro processo de γ, antes de falhar. Essas novas propriedades dão origem a novas classes de detectores, apresentadas na Tabela 2.

Abrangência	*Exatidão*			
	Forte γ	Fraca γ	Forte γ após um certo tempo	Fraca γ após um certo tempo
Forte	$P(\gamma)$	$S(\gamma)$	$\Diamond P(\gamma)$	$\Diamond S(\gamma)$
Fraca	$Q(\gamma)$	$W(\gamma)$	$\Diamond Q(\gamma)$	$\Diamond W(\gamma)$

Tabela 2: Classes de detectores de falhas propostas por [12]

Os detectores de falha que respeitam estas propriedades são chamados de **Detectores de Falhas de Exatidão** γ e, na ausência de particionamentos na rede, comportam-se de acordo com os padrões vistos na Tabela 1. Guerraoui e Schiper [12] discutem ainda a relação desses detectores com os padrões definidos por Chandra e Toueg [10] e seu impacto na solução de problemas de acordo em sistemas com particionamentos na rede.

2.3 Ciência de Falhas

Em sistemas distribuídos assíncronos puros é impossível implementar detectores de falhas com semântica perfeita. Sabel e Marzullo [13] mostraram que um detector de falhas capaz de resolver o problema de eleição de líder [14] é suficiente para implementar um detector perfeito P, ou seja, existe um algoritmo de redução que transforma um detector no outro. Por outro lado, através de uma nova propriedade, denominada de "ciência de falhas" (*fail-awareness* [15])[1], é possível resolver o problema da eleição de líder usando um detector de falhas estritamente mais fraco do que um detector perfeito P.

A propriedade de ciência de falhas requer que o módulo detector de falhas de um processo p saiba quando ele é suspeitado por k ou mais processos. Seguindo esta propriedade, se p não suspeita de si mesmo (ou seja, p não é suspeitado pelo seu próprio módulo detector de falhas), no máximo k processos suspeitam dele. Entretanto, caso p suspeite de si mesmo, k ou mais processos podem estar suspeitando de p.

Fetzer e Cristian [15] definem duas novas propriedades para detectores de falhas baseado no conceito de ciência de falhas: forte ciência de falhas e fraca ciência de falhas. A forte

[1] Ciência no sentido de conhecimento, p. ex. "Estar ciente do acontecido, nada ignora a esse respeito" [16].

ciência de falhas requer que um processo p suspeite de si mesmo tão logo algum outro processo suspeite de p, ou seja, $k = 1$. Seja um sistema com n processos, se um processo q suspeita de p no intervalo $[s, t]$, então, p vai suspeitar de si mesmo, ao menos no intervalo $[s, t]$.

Por outro lado, a fraca ciência de falhas requer que um processo p seja considerado suspeito pelo seu próprio módulo de detecção tão logo uma maioria de processos suspeite de p. Seja um sistema com de três processos p, q e r, e tempos (reais) s, t, u e v, tal que $s \leq t \leq u \leq v$. Caso o processo r suspeite do processo p no intervalo $[s, v]$, e o processo q suspeite de p no intervalo $[t, u]$, então, p suspeitará dele mesmo ao menos no intervalo $[t, u]$.

Os detectores de falhas que respeitam uma dessas duas propriedades são chamados detectores de falhas cientes de falhas. Como já foi visto anteriormente, Chandra e Toueg [10] propõem um algoritmo que transforma um detector de falhas que satisfaz a propriedade de fraca abrangência em um detector que satisfaz a propriedade de forte abrangência. Entretanto, esses algoritmos não preservam nenhuma das propriedades de ciência de falhas propostas por Fetzer e Cristian, por isso, é dada atenção apenas a duas novas classes de detectores de falhas, a saber [15]:

- $\diamond S^{WF}$ - contêm os detectores de falhas que satisfazem as propriedades de forte abrangência, forte exatidão após um tempo, e fraca ciência de falhas;
- $\diamond S^{SF}$ - contêm os detectores de falhas que satisfazem as propriedades de forte abrangência, forte exatidão após um tempo, e forte ciência de falhas.

A classe $\diamond S^{WF}$ é estritamente mais fraca do que qualquer detector da classe P, pois um detector $\diamond S_0^{WF}$ da classe $\diamond S^{WF}$ pode ser reduzido para um P_0 da classe P, mas um P_0 não pode ser reduzido para um $\diamond S_0^{WF}$. P_0 satisfaz todas as propriedades de $\diamond S_0^{WF}$, mas não existe um algoritmo que transforme um detector $\diamond S_0^{WF}$ em um detector P_0.

Fetzer e Cristian [15] propõem protocolos de eleição de líder baseados nos detectores de falha apresentados. Se quantidade de processos corretos for majoritária, então, utiliza-se os detectores da classe $\diamond S^{WF}$, isto contraria o resultado de Sabel e Marzullo [13] (que não considera as propriedades de ciência de falhas). Se, pelo menos, um processo permanece correto, então, utiliza-se os detectores da classe $\diamond S^{SF}$. Estes protocolos são válidos desde que os processos consigam detectar alterações nas saídas de seus módulos detectores de falhas, como também, um módulo detector possua um sinalizador $s_q(p)$ e um número de versão $v_q(p)$ tal que:

- o sinalizador $s_q(p)$ é verdadeiro quando o processo q suspeita do processo p;
- o número de versão $v_q(p)$ é incrementado sempre que o valor do sinalizador $s_q(p)$ muda.

3 Modelos e Implementações de Detectores de Falhas

Apesar de não poderem ser usados como uma indicação precisa da falha de um processo em um ambiente assíncrono, os *timeouts* são amplamente utilizados nas implementações de detectores de falhas, pela simplicidade de seu uso e pelo fato de que, em situações práticas,

não é necessário esperar indefinidamente pela mensagem de um processo para ter certeza de sua falha, bastando para isso esperar por um tempo suficientemente longo.

Existem dois modelos baseados nestes limites de tempo para troca de mensagens: o *pull* e o *push*. O *pull* se baseia no envio periódico de mensagens do tipo "você está vivo?" para os processos monitorados. Logo após o envio dessas mensagens, é ativado um temporizador e se o detector não receber de volta uma mensagem do tipo "sim, eu estou vivo" daquele processo, este é adicionado na lista de processos falhos.

Já o método *push* baseia-se apenas no recebimento de mensagens do tipo "eu estou vivo" dos processos, para considerar seu estado. Essas mensagens são muitas vezes chamadas de *heartbeats*, o que não deve ser confundido com o detector de falhas *Heartbeat* visto mais a frente neste trabalho. O não recebimento de tais mensagens em um intervalo de tempo faz com que o processo seja incluído na lista de suspeitos. Uma variação deste método é o *push ad hoc*. Nele, as mensagens do tipo "eu estou vivo" são enviadas apenas quando a detecção da falha é relevante. Por exemplo, um processo p pode requisitar um serviço de q, invocando uma função *f1*. Neste momento, q começa a enviar mensagens do tipo "eu estou vivo", para que o detector de falhas o diagnostique como correto, no caso de p consultar o estado de q enquanto espera pela finalização de *f1*. Em quaisquer dos métodos, se uma mensagem de um processo chega após ele ser considerado suspeito, então ele é retirado da lista de suspeitos.

Vale ressaltar que existe ainda uma quarta abordagem para a detecção de falhas em aplicações distribuídas: a implementação *ad hoc*, sem mensagens específicas para a detecção de falhas. Na verdade, ela não utiliza detectores de falhas para consultar o estado dos processos e a aplicação implementa seu próprio modo de detecção de falhas. Ela baseia-se no fato de que um algoritmo precisa da informação de detecção de falhas apenas em alguns momentos muito específicos. Nesses momentos, o algoritmo envia uma mensagem para o processo do qual a informação de falha é importante. Essa mensagem é chamada mensagem crítica e é uma mensagem normal, com significado para os algoritmos da aplicação. O processo que enviou a mensagem crítica aguarda então uma resposta crítica. Se não receber esta resposta crítica até um determinado momento, o processo que enviou a mensagem crítica marca o processo destino desta mensagem como suspeito. Esta abordagem exige que o algoritmo conheça de antemão as mensagens que serão enviadas e tire proveito do grau de sincronismo do canal (não é implementável em sistemas assíncronos puros).

Desde o trabalho de Chandra e Toueg [10], vários detectores de falhas têm sido implementados. Alguns deles seguem completamente os modelos e especificações apresentadas, outros tentam abordar aspectos não contemplados por ele. A seguir, veremos uma breve descrição de algumas implementações de detectores de falhas.

3.1 ESPRIT OpenDREAMS

Felber *et al.* [17] propõem um serviço de detecção de falhas no nível do sistema operacional, tal como os serviços de nomes, de autenticação e de gerenciamento de arquivos. O detector de falhas seria inacessível diretamente e seu serviço estaria acessível a desenvolvedores através de objetos e um grupo de interfaces, organizadas hierarquicamente, como é apresentado na Figura 2 [10].

Ao invés de considerar o monitoramento de processos, Felber *et al* [17] considera o mo-

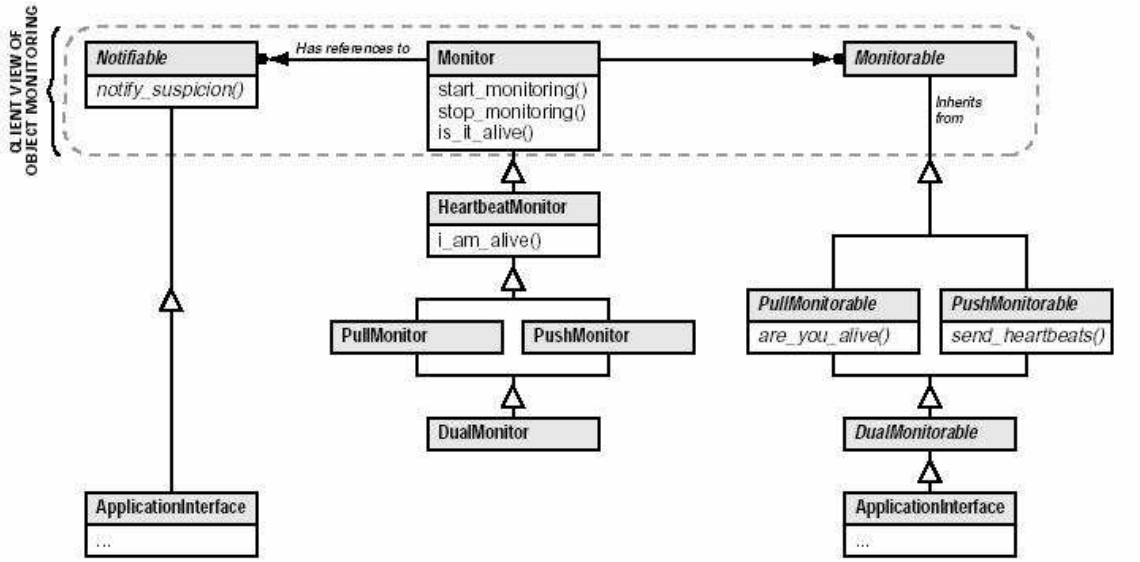


Figura 2: Diagrama de classes do serviço de monitoramento do ESPRIT OpenDREAMS

nitoramento e a notificação de objetos. Isso não impede, entretanto, que um processo seja monitorado ou notificado. Basta que um objeto aja como seu representante e intermedeie a comunicação do processo com o detector de falhas.

Existem três entidades no processo de monitoramento, denominadas de: *Monitor*, *Monitorable* ou *Notifiable*. *Monitor* é o objeto central que representa o detector de falhas e oferece uma interface para o uso do serviço. *Monitorable* é todo objeto que deva ser monitorado para detecção de falhas e *Notifiable* é todo objeto que deva ser notificado da ocorrência de falhas nos objetos monitorados.

O objeto *Monitor* fornece uma interface que permite aos clientes iniciarem ou interromperem o serviço de monitoramento, consultarem o estado de objetos notificáveis e gerenciarem a lista de objetos *Notifiable*. Para tornar-se um objeto *Notifiable*, um objeto deve implementar a interface *Notifiable* e se cadastrar na lista de notificáveis do objeto Monitor. Para tornar-se um objeto *Monitorable*, um objeto deve implementar uma das interfaces filha da interface *Monitorable*: *PullMonitorable*, *PushMonitorable* ou *DualMonitorable*. Este modelo permite que tanto a abordagem do modelo *pull* quanto a abordagem do modelo *push*, ou ambas, sejam adotadas pelo detector de falhas, dependendo das características do sistema e de cada objeto monitorado.

O protocolo de monitoramento proposto por este trabalho é inovador no sentido de permitir que a detecção de falhas se ajuste ao modelo suportado por cada objeto monitorado. Este protocolo é composto por duas fases. Na primeira fase, seguindo o estilo do modelo *push*, o objeto *Monitor* aguarda o envio das mensagens "eu estou vivo" provenientes do objeto monitorado. Após um tempo limitado, se nenhuma mensagem for recebida, então o *Monitor* assume que o modelo adotado pelo objeto é o *pull* e então passa a enviar mensagens "você está vivo?" e esperar as mensagens "sim, eu estou".

No caso do objeto monitorado implementar a interface *PushMonitorable*, tão logo o objeto *Monitor* receba suas mensagens, o modelo *push* será adotado para seu monitoramento. Por

outro lado, se o objeto monitorado implementar *PullMonitorable*, o *Monitor* esperará sem sucesso pela chegada de mensagens do estilo *push* e adotará o modelo *pull* para monitorá-lo. Caso a interface implementada seja *DualMonitorable*, o *Monitor* poderá oscilar entre os dois modelos, se algum destes falhar.

O projeto ESPRIT OpenDREAMS, desenvolvido pelo laboratório de sistemas distribuídos do Swiss Federal Institute of Technology, na Suíça, inclui um serviço de deteccção de falhas baseado no modelo apresentado acima. O ESPRIT OpenDREAMS é um *framework* compatível com CORBA, para supervisão e controle de sistemas distribuídos, que tem influenciado várias propostas feitas à OMG [18] para padronização de mecanismos de tolerância a faltas para o CORBA.

3.2 OGS The CORBA Object Group Service

Apresentado por Felber *et al.* [19, 20], o OGS é uma plataforma que provê suporte a comunicação em grupo sobre o ambiente CORBA. Esta plataforma é composta por uma série de serviços, quais sejam: *multicast*, *membership*, consenso, troca de mensagens confiável ponto a ponto e detecção de falhas. O serviço de detecção de falhas baseia-se na mesma abordagem usada no projeto ESPRIT OpenDREAMS. A primeira versão da OGS adotava apenas o modelo *pull* de monitoramento, mas a partir da segunda versão, o modelo *push* e o *dual* passaram a ser suportados.

3.3 DOORS Distributed Object-Oriented Reliable Service

O DOORS [21, 22] é um *framework* que provê tolerância a faltas a sistemas distribuídos sob um ambiente CORBA. Este *framework* foi desenvolvido nos laboratórios da Bell, uma das empresas que compõem a OMG e é baseado em um serviço de detecção de falhas, cujos padrões foram incorporados à especificação FT CORBA (CORBA Tolerante a Falhas) [18]. A seguir será apresentada a arquitetura da primeira versão do DOORs, como também as adaptações implementadas para tornar o *framework* compatível com a especificação FT CORBA.

O DOORS foi estruturado em três módulos: *WatchDog*, *ReplicaManager* e *SuperWatchDog* (ver Figura 3 [21]). O módulo *WatchDog* executa em cada máquina do sistema e monitora os objetos através da troca de mensagens seguindo o modelo de detecção *push* ou *pull*, dependendo das características do objeto. Os objetos suspeitos de terem falhado são reportados ao objeto *ReplicaManager*. O *ReplicaManager*, por sua vez, é um objeto centralizado, que tem a função de gerenciar réplicas de objetos, gerenciando a criação e migração de réplicas na ocorrência de falhas. Para cada objeto monitorado, existe uma tabela no *ReplicaManager* com informações sobre as máquinas onde o objeto pode ser ativado, o estilo de replicação, o estado de cada réplica, dentre outras informações. Inicialmente, todos os objetos da aplicação se registram no *ReplicaManager*. O terceiro módulo, *SuperWatchDog*, também é centralizado e é responsável pela detecção de falhas em máquinas. Todos os objetos *WatchDog* se registram no *SuperWatchDog* e periodicamente enviam uma mensagem, seguindo o modelo *push*. Ao detectar uma falha em uma máquina, o *SuperWatchDog* pode recriar seus objetos em outras máquinas, e notificar a ocorrência da falha a objetos interessados e cadastrados previamente.

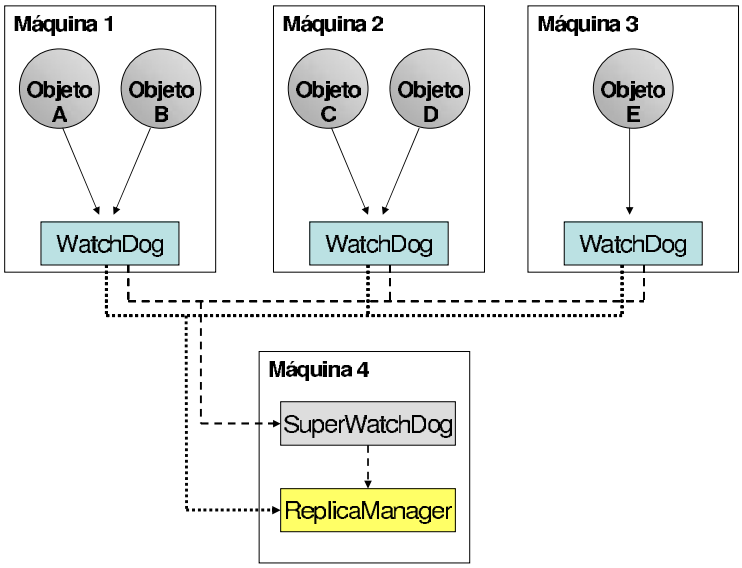


Figura 3: Arquitetura do DOORS

Os objetos centralizados, *SuperWatchDog* e *ReplicaManager*, são replicados para que não haja pontos únicos de falha. O mecanismo de replicação é o primário-cópia, então, no caso de falha destes objetos, suas réplicas(cópias) executam um protocolo de eleição para escolher um novo primário.

O DOORs foi desenvolvido antes do lançamento da especificação FT CORBA, e por isso, uma nova versão do *framework* foi criada para atender à especificação. Poucas alterações foram necessárias. Os objetos *WatchDog*, *SuperWatchDog* e o *ReplicaMaganer* passaram a ser chamados *FaultDetector*, *SuperFaultDetector* e *ReplicationManager*, respectivamente, mas a função de cada objeto, assim como, o comportamento, permaneceram os mesmos. O objeto *FaultNotifier*, previsto pela especificação FT CORBA ainda não foi implementado no DOORS.

3.4 Monitor

O Monitor é uma ferramenta desenvolvida pelo Laboratório de Sistemas Distribuídos da Universidade Federal da Bahia (UFBA). Esta ferramenta é composta por dois serviços: o serviço de diagnóstico de falhas (SDF) e o serviço de gerenciamento distribuído (SGD). Os trabalhos de Batalha [23] e de Oliveira [24] apresentam as arquiteturas dos serviços da ferramenta Monitor e sua implementação na plataforma Java/CORBA.

O serviço SDF é uma implementação de um detector de falhas da classe $\diamond S$ que segue o modelo de detecção *pull*. Existe um módulo SDF em cada estação da rede que executa processos da aplicação distribuída a ser monitorada. Outras aplicações podem se cadastrar em um módulo SDF para receber notificações da ocorrência de eventos envolvendo os processos monitorados por este módulo. Alguns exemplos de notificação incluem: processo falho, inclusão de um novo processo para monitoria, exclusão de um processo monitorado, falha em uma máquina e finalização normal de um processo. O serviço SGD é um dos interessados

em receber notificações do SDF. Sua função é permitir o gerenciamento dos processos de uma aplicação distribuída, possibilitando a um administrador criá-los em qualquer máquina da rede, terminar os que estiverem falhos, movê-los de uma máquina para outra, configurar suas propriedades, além de fornecer uma visão consistente de todas as máquinas e processos monitorados. O SGD pode ser iniciado em qualquer estação da rede, desde que possa consultar o serviço de nomes CORBA e conseqüentemente um módulo SDF. A Figura 4 mostra como funciona a ferramenta Monitor.

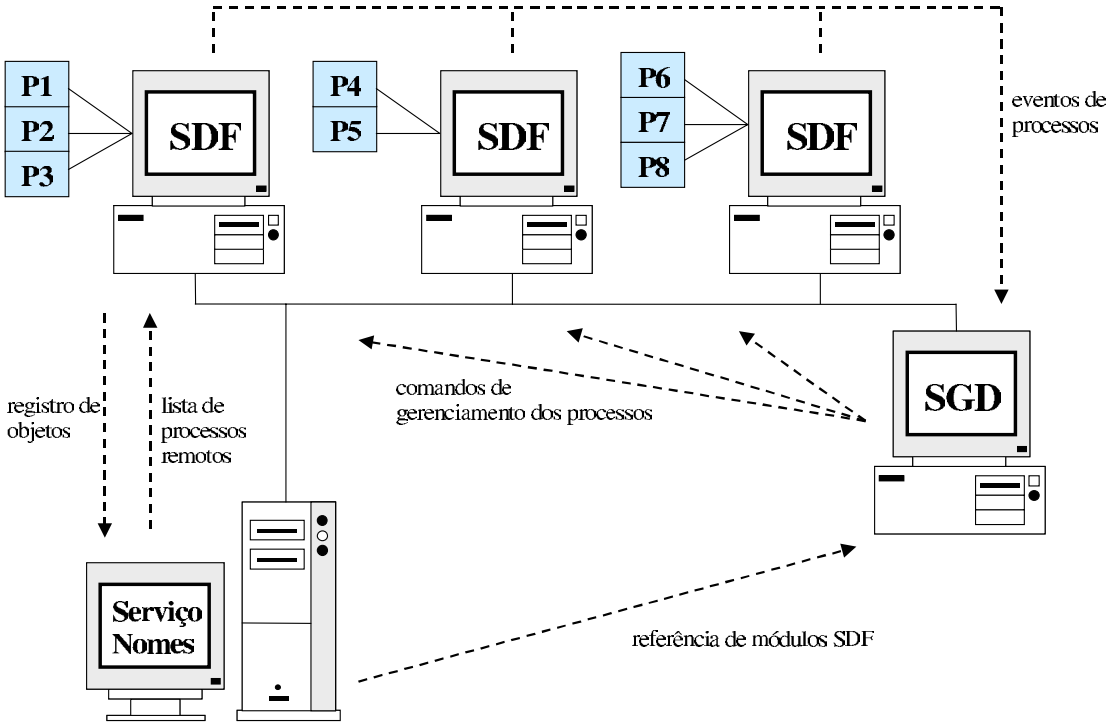


Figura 4: O Monitor em funcionamento

O SDF continua funcionando mesmo quando a maioria de seus módulos falham, restando apenas um ativo. Após ser iniciado, um módulo SDF procura por outro módulo do mesmo tipo presente na rede. Uma vez encontrado, requisita dele a lista de todos os processos monitorados. Obtendo esta lista, passa então a monitorar todos os processos, de forma autônoma. Por isso, a falha de um módulo não implica na interrupção do monitoramento de nenhum processo, inclusive aqueles em execução na máquina cujo módulo falhou.

Um módulo do SDF é dividido em dois sub-módulos que trabalham de forma coordenada: o módulo de Reconfiguração e o módulo de Detecção de Falhas. O módulo de Reconfiguração funciona como o cérebro do SDF. Este módulo é responsável por manter a lista de processos corretos atualizada, iniciar o monitoramento de processos, e interromper o monitoramento de processos detectados como falhos, além de notificar a ocorrência de determinados eventos a outros objetos cadastrados para receber tal notificação. O módulo de Detecção de Falhas, como o próprio nome diz, tem a função de detectar a ocorrência de falhas em processos ou máquinas (no caso, máquinas onde os processos da aplicação executam). Este módulo é responsável por manter informações sobre os níveis de tempo de comunicação aceitáveis para

os processos monitorados, os quais são definidos durante a iniciação de cada processo. Além disso, periodicamente, devem tentar contactar cada processo monitorado e trocar mensagens com as máquinas onde os mesmos residem, a fim de detectar processos que não estejam respondendo dentro dos limites de tempo aceitáveis, seja por falha na máquina ou nos processos.

Cada módulo SDF possui suas próprias listas de processos corretos e de processos falhos. As listas, geralmente, possuem o mesmo conteúdo quando comparadas com seus pares nos outros módulos. Em algumas situações, estas listas podem não apresentar a mesma informação, mas, existem protocolos que garantem a convergência de todas as listas para uma visão única e coerente do estado dos processos [23].

Existem três formas de monitoramento de um processo pelo SDF. Utilizando Java, pode-se implementar uma interface de gerenciamento definida pelo serviço. No caso de outra linguagem, é necessário o uso de um processo intermediário que implemente a interface de gerenciamento do SDF e se comunique com o processo monitorado. A terceira possibilidade é realizar o monitoramento utilizando a especificação CORBA Tolerante a Falhas [18]. Neste caso, também é necessário o uso de um processo intermediário que implemente a interface de gerenciamento do SDF e conheça a interface de gerenciamento prevista pelo FT CORBA para comunicação com o objeto monitorado.

3.5 *Muteness*

Doudou *et al.* [25] realizam uma inovação ao tentar ampliar o conceito de detector de falhas para o tratamento de falhas do tipo *muteness*. A maioria dos detectores de falhas considera apenas falhas do tipo *crash*, onde os processos monitorados simplesmente param de executar definitivamente. No entanto, em um sistema distribuído, outros tipos de falhas podem ocorrer.

As falhas do tipo *crash* são um tipo particular de falhas *muteness*. Uma falha *muteness* ocorre quando um processo se torna "mudo", deixando de enviar mensagens. Um processo pode interromper o envio de toda e qualquer mensagem por ter abortado sua execução, mas condições anormais podem afetar apenas o envio das mensagens relacionadas ao algoritmo que o processo executa, enquanto as mensagens de controle, do tipo "eu estou vivo", as quais são esperadas pelo detector de falhas continuam sendo enviadas. Neste caso, o processo estará parcialmente mudo e o detector de falhas nunca suspeitará dele, porque continua a ouvir suas mensagens "eu estou vivo". Entretanto, do ponto de vista do algoritmo, tal processo poderia ser considerado falho.

A proposta do detector de falhas *Muteness* é promover uma integração maior entre o mecanismo de detecção de falhas e o algoritmo dos processos monitorados. A idéia consiste em fazer com que as mensagens de controle sejam enviadas em pontos estratégicos da execução do algoritmo. A cada passo da execução do algoritmo, uma mensagem identificando o processo e o estágio de execução alcançado seria enviada para o detector, ao invés de simples mensagens do tipo "eu estou vivo". Dessa forma, o detector saberia se o processo está progredindo em sua execução ou não.

3.6 *Heartbeat*

Aguilera *et al.* [26] propõem uma alternativa para o uso de *timeouts* e listas de suspeitos na detecção de falhas. Eles propõem que cada processo monitorado envie periodicamente uma mensagem *heartbeat* para o detector de falhas. Ao receber esta mensagem, o detector incrementa o contador de *heartbeats* transmitidos por aquele processo. Dessa forma o detector não interpreta os *heartbeats* recebidos, apenas acumula, em contadores, a quantidade mensagens recebidas de cada processo monitorado.

O detector de falhas *Heartbeat*, não mantém listas de suspeitos e, ao ser consultado informa o último valor do contador de *heartbeats* para um determinado processo. Os clientes do detector, por sua vez, devem traduzir este valor e determinar se um processo falhou ou não. Eles podem comparar este valor com outro valor lido em um momento anterior. Se os valores forem iguais significa que neste período não ocorreu nenhum novo *heartbeat*, então, o processo pode ser considerado falho. Caso tenha ocorrido algum *heartbeat* (os valores serão diferentes), o processo é considerado correto. Embora não sejam utilizados *timeouts* no recebimento de mensagens pelo detector, limites de tempo são utilizados no julgamento do estado dos processos pelos clientes do detector de falhas.

O modelo *Heartbeat* possui a mesma imprecisão dos modelos anteriores, mesmo assim, revela-se mais adequado para sistemas onde o tempo de envio de mensagens por um processo é oscilante, e por isso poderiam ser freqüente e erroneamente considerados falhos nos outros modelos. Este modelo também é adequado para sistemas com grande número de processos devido ao pequeno número de mensagens geradas. Nos demais modelos, um cenário como este provocaria um aumento exponencial de troca de mensagens o que poderia interferir no desempenho do sistema como um todo. No modelo *Heartbeat*, a troca de mensagens é reduzida pela adoção de uma técnica bastante eficaz, baseada no conceito de vizinhos. A idéia é que um processo envie seu *heartbeat* para processos em máquinas vizinhas. Estes processos por sua vez, repassariam o *heartbeat* para outros vizinhos, que ainda não o tivessem recebido. Antes de repassar um *heartbeat*, cada processo deveria adicionar ao *heartbeat* seu código de identificação, formando assim uma lista da rota por onde passou a mensagem. Ao receber o *heartbeat*, o detector incrementaria, de uma só vez, os contadores de todos os processos com códigos registrados na mensagem.

Esse modelo é criticado por atribuir aos clientes dos detectores a tarefa de suspeitar dos processos falhos. Uma extensão deste modelo objetiva tornar transparente a detecção de falhas, fornecendo uma interface tradicional aos clientes do serviço de detecção, onde é possível consultar a lista de processos suspeitos (ver Figura 5 [26]). A idéia foi adicionar uma camada entre o detector e seus clientes, cuja função seja de consultar, periodicamente, os contadores de *heartbeats* e assim, formar uma lista de suspeitos. Isso permite que este detector seja trocado por outro, se necessário, sem envolver mudanças no algoritmo da aplicação cliente.

3.7 Piranha

Maffeis apresenta Piranha [27], uma ferramenta de monitoramento e gerenciamento de objetos em aplicações CORBA. Através desta ferramenta, um usuário pode acompanhar, através de uma interface gráfica, as alterações no estado dos objetos e executar ações gerenciais

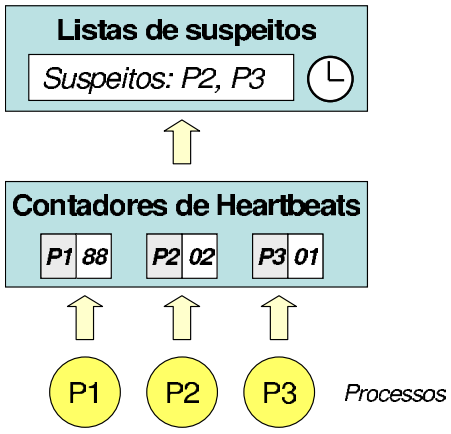


Figura 5: Extensão do modelo Heartbeat

sobre os mesmos, como por exemplo: iniciar objetos que tenham falhado, migrar e replicar objetos. A Figura 6 [27] mostra a arquitetura da ferramenta Piranha.

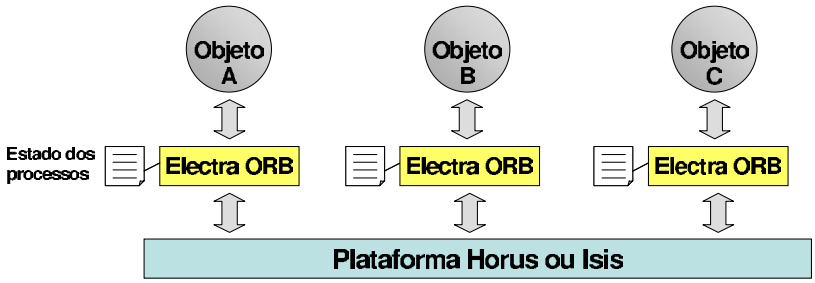


Figura 6: Arquitetura da ferramenta Piranha

A base para o funcionamento da ferramenta é um serviço de detecção de falhas, cuja interface é fornecida por um ORB estendido, chamado Electra. O Electra funciona sobre os subsistemas de comunicação de grupo HORUS ou ISIS, responsáveis, de fato, pela detecção de processos falhos. Esta detecção é baseada no uso de *timeouts* e por esse motivo pode cometer erros, ao considerar processos corretos como falhos. Cada módulo do Electra possui uma lista do estado de todos os objetos monitorados. Através da plataforma HORUS ou ISIS, as mudanças verificadas no estado dos processos são propagadas para todas as listas de estado, mantendo uma visão consistente dos objetos monitorados.

Piranha possui o objetivo de ser uma ferramenta gráfica para gerenciamento de aplicações CORBA, não sendo seu objetivo fornecer um serviço de detecção de falhas bem definido, que sirva de suporte para desenvolvimento de aplicações distribuídas.

3.8 Detectores de Falha com Semântica Perfeita

Detectores de falhas com semântica perfeita P não podem ser implementados em sistemas assíncronos [28]. Caso contrário, o consenso poderia ser resolvido em um ambiente assíncrono, o que invalida o resultado de impossibilidade comprovado por Fischer *et al.* [6].

Portanto, tais detectores são implementáveis, apenas, em sistemas síncronos. Para oferecer um serviço de detecção de falhas perfeito em ambientes assíncronos, é necessário impor restrições ao ambiente. Estas restrições devem criar um subsistema síncrono, que possa garantir as propriedades da classe P. Isto significa introduzir alterações no sistema operacional, adotar um canal extra de comunicação dedicado à troca de mensagens do serviço, ou ainda, controlar o número de processos e máquinas suportados no sistema.

Chandra e Toueg provaram que os detectores da classe $\diamond S$ são suficientes para resolver o consenso [10], e que esses detectores não impõem muitas restrições ao ambiente onde serão implementados. Esses resultados tendenciaram à implementação de detectores de falhas com semânticas mais fracas, especialmente detectores da classe $\diamond S$. Entretanto, conforme explicado na Seção 2, os detectores de falha com semântica perfeita são os únicos que podem informar com exaditão se um processo falhou ou não.

Algumas aplicações necessitam de detectores de falhas que não cometam erros. Por exemplo, sistemas de *backup* podem utilizar um detector de falhas perfeito para detectar a falha de um sistema primário. Quando o *backup* detecta a falha do primário, realiza-se a substituição sem o risco do primário ainda estar operacional. Vejamos o caso dos sistemas de *backup* e primário utilizarem um mesmo endereço IP/Ethernet ou ambos acessarem um mesmo dispositivo de armazenamento (por exemplo, um disco SCSI externo). Utilizando um detector de semântica mais fraca, propenso a cometer falsas suspeições, poderia ocorrer uma situação onde os dois sistemas se considerem primários. Os dois sistemas funcionando como primários acessariam em paralelo o disco compartilhado, gerando inconsistências, ou ainda, utilizariam o mesmo endereço IP/Ethernet, gerando conflito.

3.8.1 Simulando um Detector de Falhas Perfeito em Sistemas Assíncronos

O modelo *fail-stop* simulado, desenvolvido por Sabel e Marzullo [28], constitui uma forma de simular um detector de falhas perfeito em um sistema assíncrono. No modelo de falhas *fail-stop*, todo processo que falha, o faz silenciosamente e essa falha pode ser detectada por outros processos. Neste modelo, as falhas são menos malignas do que no modelo de falhas do tipo *crash*, que não pode ser implementado em sistemas assíncronos. Um sistema com um modelo de falhas *fail-stop* é equivalente a um sistema equipado com um detector de falhas perfeito. Embora Sabel e Marzullo [28] mostrem que o modelo de falhas *fail-stop* não é implementável em sistemas assíncronos (o mesmo é válido para detectores de falhas perfeito), eles propõem um modelo de falhas, que é indistinguível do modelo de falhas *fail-stop* do ponto de vista de qualquer processo dentro do sistema assíncrono. Para tal, suspeitas, mesmo incorretas, levam os processos suspeitados a serem forçados a falhar. Desta forma, seria possível definir um detector de falhas indistinguível em relação a um detector perfeito.

Embora este resultado pareça promissor, o mesmo é válido, apenas, em um ambiente onde a comunicação entre os processos seja confiável e proceda por meio de troca de mensagens em um canal controlado. Nesse caso, o canal de comunicação por onde trafegam as mensagens não perde, gera ou corrompe mensagens. Além disso, o canal preserva a ordem das mensagens. Nesse modelo, se um processo p suspeita de outro processo q, então tal suspeita "acontece antes" de ações locais em p. A relação "acontecer antes" [29] não está associada com a existência de um relógio global, nem pode ser aplicada em cenários onde existam

canais ocultos de comunicação. Caso haja troca de mensagens através de canais ocultos ou a presença de um observador externo, pode ocorrer falsas suspeições. Desta forma, abordagem *fail-stop* não resolve, por exemplo, os problemas de conflito de endereço e compartilhamento de discos exemplificados acima.

3.8.2 Utilização de Licenças e *Watchdogs* para Implementação de um Detector Perfeito

Fetzer propõe em [30] uma abordagem baseada no uso de *watchdogs* em *hardware*, como componentes síncronos, para viabilizar a construção de detectores de falhas perfeitos. Nesta abordagem, os *watchdogs* são programados para desligar suas respectivas máquinas, após um determinado período. Para impedir que a máquina seja desligada, um protocolo executando nesta máquina deve conseguir licenças, as quais são concedidas por outras máquinas. Cada licença tem um tempo de expiração. Quando uma licença expira, e não é renovada, a máquina realmente falhou. Então, uma máquina será suspeitada pelo serviço de detecção apenas quando esta falhar.

Em termos de implementação, esta abordagem requer uma rede com três máquinas. A renovação de licenças pode ser feita por uma das máquinas. Como todas as máquinas têm que renovar suas licenças com pelo menos uma outra máquina, é assegurado que uma falha será percebida pelas duas máquinas restantes em um intervalo de tempo previsível. Ao renovar uma licença, a máquina provedora da licença sabe que a máquina requisitante está correta. A renovação de licenças acontece por meio de um mecanismo que garante o tempo de expiração da licença, ou seja, não se pode renovar licenças cujo tempo de vida (desde sua concessão) tenha ultrapassado um certo limite de tempo. Caso a idade de uma solicitação não pudesse ser determinada, seria possível conceder licenças a uma máquina que já falhou, considerando-a, erroneamente, como operacional (ou correta). Ao conseguir uma licença, o protocolo pode reprogramar o *watchdog* por mais um período fixo.

Em um ambiente assíncrono não existem garantias sobre o tempo máximo para entrega das mensagens. Dessa forma, determina-se a idade de uma solicitação utilizando um serviço de comunicação especial, que aproveita o mínimo de sincronismo do sistema no sentido de perceber se uma mensagem foi entregue dentro de um limite de tempo especificado [31].

Nesta abordagem, a detecção de falhas é feita em relação a máquinas. Além disso, o número máximo de máquinas onde o protocolo executa é limitado e igual a três. O tempo de detecção de falhas mínimo obtido foi de $2s$, o médio foi de $3.93s$ e o máximo de $5.9s$.

3.8.3 Delphus

O Delphus [32] é um serviço de detecção de falhas, desenvolvido no Laboratório de Sistemas Distribuídos - Universidade Federal de Campina Grande - que oferece garantias de qualidade de serviço e se destina a redes locais com pouca dispersão geográfica, onde as máquinas executam o sistema operacional Linux.

O serviço é implementado no nível do sistema operacional e não faz nenhuma restrição quanto à versão do kernel utilizado. Além disso, é construído com o suporte de uma rede de comunicação adicional utilizada, exclusivamente, para o tráfego de mensagens do serviço. A utilização desta rede adicional é cuidadosamente controlada, além disso, é dada uma atenção

especial ao escalonamento das *threads* do sistema operacional que implementam o núcleo do serviço de detecção de falhas em cada máquina. Dessa forma, é possível definir atrasos máximos na comunicação com probabilidade muito alta, criando um ambiente síncrono de execução sobre o qual o serviço de detecção de falhas é implementado. A rede síncrona, que serve de suporte para o serviço de detecção, é implementada por um dispositivo de comunicação sem fio de baixo custo e especialmente desenvolvido. Na ausência do dispositivo, placas de rede adicionais podem ser utilizadas para oferecer uma amostra do serviço.

O Delphus é utilizado por dois tipos de entidades, denominadas de *monitoráveis* e *notificáveis* [17]. Tanto os processos como as máquinas podem ser entidades monitoráveis, *i.e.* podem ser submetidos a monitoramento, e neste caso, suas falhas devem ser detectadas pelo serviço. Por outro lado, entidades notificáveis são processos interessados em ser notificados da ocorrência de falhas das entidades monitoráveis. Além disso, qualquer processo pode explicitamente consultar o serviço de detecção de falhas sobre a ocorrência de falhas de quaisquer entidades monitoráveis.

O serviço de detecção de falhas é implementado por módulos de detecção de falhas independentes, executando em cada máquina participante de um domínio de detecção em particular[2]. Cada módulo implementa um detector de falhas do estilo *push* [17] que periodicamente envia mensagens *heartbeat* de suas entidades monitoráveis locais. Essas mensagens são enviadas pela rede síncrona. A utilização da rede síncrona obedece ao protocolo TDMA (*Time Division Multiple Access*). No TDMA a utilização do canal é dividida em fatias de tempo (*slots*). Cada módulo tem sua própria fatia de tempo (de comprimento fixo) para transmitir suas mensagens *heartbeat*. Quando uma entidade monitorável parar de enviar *heartbeats*, será percebida a ausência de tais mensagens e todo módulo correto irá detecta a falha desta entidade. Então, cada módulo notifica todas as entidades notificáveis locais que solicitaram uma notificação para a falha daquela entidade monitorável cuja falha foi detectada. Cada módulo tem ao menos uma entidade monitorável; esta entidade é a máquina na qual o módulo executa. Desta forma, em cada período, todo módulo deve utilizar sua fatia de tempo no canal para transmitir ao menos o *heartbeat* da sua máquina. A Figura 7 mostra a arquitetura da ferramenta Delphus.

Um módulo local do serviço é implementado por três *threads* em um módulo do sistema operacional (Linux). A *thread do_period* é responsável pela implementação das principais funcionalidades do serviço. Além disso, ela envia as mensagens na rede síncrona. A *thread sync_rec* é responsável por receber mensagens pela rede síncrona. Esta *thread* permanece a maior parte do tempo bloqueada, esperando pela chegada de mensagens no canal síncrono. Depois de receber uma mensagem, identifica-se o tipo da mensagem, e invoca-se o manipulador específico para processá-la. Foram definidos quatro tipos de mensagens síncronas[3], quais sejam: *proposta de liderança*, *sincronização*, *anúncio* e *heartbeat*.

As duas *threads* descritas acima implementam o núcleo do serviço de detecção de falhas, e são configuradas para terem a maior prioridade de tempo real disponível. *Threads* com esta prioridade têm privilégios na política de escalonamento do Linux, e são escalonados tão logo

[2]A rede síncrona que dá suporte ao serviço define um domínio de detecção. Todas as entidades associadas a um domínio de detecção em particular são unicamente identificadas por um identificador global.

[3]Utilizam-se os termos mensagens síncronas e mensagens assíncronas para indicar mensagens enviadas através da rede síncrona e assíncrona, respectivamente.

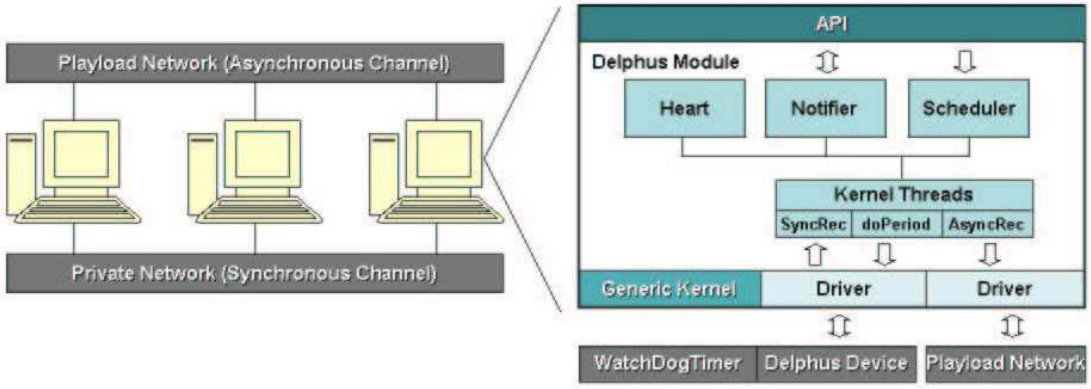


Figura 7: A arquitetura do Delphus

se tornem prontas para executar, *i.e.* seu atraso de escalonamento é limitado pela resolução das interrupções do relógio ($1ms$ no nosso sistema).

A terceira *thread*, *async_rec*, é a versão assíncrona do *sync_rec*. Esta *thread* recebe mensagens através da rede assíncrona. Existem cinco tipos de mensagens que podem ser recebidas: *requisição de inscrição*, *requisição de remoção*, *recusa de inscrição*, *atualização* e *confirmação*. A única tarefa desta *thread* é implementar os procedimentos de inscrição e remoção, gerenciando as mensagens de anúncio que devem ser realizadas na fatia de tempo destinada aos anúncios pelo líder. Como não está associada a prazos, a *thread async_rec* não tem uma prioridade de tempo real.

Para a utilização do serviço, definiu-se uma API (*Aplication Programming Interface*), dividida em três grupos de funções: administração do serviço, configuração de monitoramento e notificação de falhas. A API foi implementada em C e em Java e pode ser consultada através da Internet, pelo site http://www.dsc.ufcg.edu.br/delphus.

3.8.4 TCB (*Timely Computing Base*)

Uma abordagem semelhante ao Delphus é adotada pelo TCB (*Timely Computing Base*), um *framework* apresentado por Veríssimo e Casimiro [33], cujo objetivo é fornecer suporte a aplicações que necessitam executar tarefas com requisitos temporais e qualquer grau de sincronismo. Nesta abordagem, um sistema é composto de uma parte assíncrona, onde aplicações sem garantias de tempo executam utilizando uma rede assíncrona, e a porção síncrona, onde executa o TCB. A detecção de falhas com semântica perfeita está entre os serviços disponibilizados pelo TCB.

A base do TCB é a existência de um módulo síncrono dentro de cada máquina, não importando quão síncrona seja o resto da máquina. Os módulos do TCB se comunicam por uma rede síncrono exclusiva. Como os módulos são síncronos e são interligados por uma rede também síncrona, a implementação de um detector de falhas perfeito é trivial. Embora a detecção de falhas seja realizada em tempo real, não existem garantias sobre o tempo decorrido até que as aplicações, executando na porção assíncrona do sistema, realmente recebam as informações do TCB.

Em outro trabalho [34], Casimiro *et al.* mostram como construir o módulo síncrono

utilizando Real-Time Linux. O TCB é construído a partir de um módulo de *software* que executa em cada máquina sobre um sistema operacional de tempo real (RTLinux). Cada máquina é chamada de site e utiliza uma rede síncrona privativa do TCB para comunicação, denominada de *Control System Network*. As máquinas são também conectadas por um canal assíncrono, chamado *Payload Network*, onde ocorre a comunicação assíncrona entre as aplicações do sistema. A Figura 8 [34] mostra a arquitetura estendida do TCB.

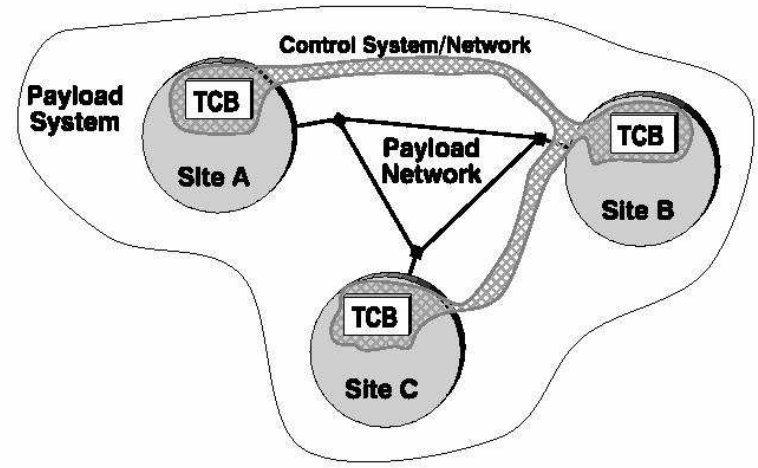


Figura 8: Arquitetura estendida do TCB

Para a detecção de falhas, o TCB exige que as aplicações informem, antes de iniciar, seu prazo e o instante que serão iniciadas, e ao terminar, informem que concluíram a tarefa. Essa estratégia de acompanhamento da execução das aplicações permite a detecção de falhas de desempenho (que englobam as falhas do tipo *crash*). Entretanto, ela exige que as aplicações sejam (re-)escritas para a utilização do TCB, aumentando seu acoplamento com o serviço de detecção.

4 Qualidade de Serviço de Detectores de Falhas

Os detectores de falhas definidos por Chandra e Toueg [10] são freqüentemente especificados em termos de um comportamento "eventual" (i.e. após um tempo finito, o detector apresentará determinado comportamento). Tal especificação é adequada para sistemas assíncronos, onde não existe o conceito de tempo ou relógio. Entretanto, aplicações freqüentemente possuem restrições de tempo, mesmo que essas restrições não sejam rígidas. Neste caso, detectores de falhas que apresentam um comportamento "eventual" não são muito úteis. Um detector de falhas que suspeita de um processo várias horas após sua falha pode ser útil em um sistema assíncrono puro, mas certamente não será de valia em um sistema que realiza diversos consensos por segundo. Sendo assim, algumas aplicações necessitam de detectores de falhas cuja qualidade de serviço (*QoS - Quality of Service*) possa ser quantificada.

Chen *et al.* [35] propõem medidas de QoS para detectores de falhas não confiáveis. Essas medidas quantificam o quão rapidamente o detector de falhas detecta falhas reais; e quão freqüentemente o detector comete erros detectando falhas que não aconteceram.

Considere um serviço de detecção implementado por um conjunto de módulos independentes em um sistema com n processos. Cada processo p_i tem acesso a $n-1$ módulos locais de detecção de falhas,e cada módulo monitora um dos outros $n-1$ processos do sistema. As medidas de QoS propostas em [35] são aplicadas a cada módulo local de detecção de falhas $DF_{i,j}$ de um processo p_i que monitora outro processo p_j. Como o funcionamento do sistema é probabilístico as medidas são variáveis aleatórias. Foram definidas três medidas de QoS básicas [35], a saber:

- **Tempo de detecção** (*detection time* - $T_{D(i,j)}$): variável randômica que representa o tempo transcorrido entre a ocorrência de uma falha e sua detecção (suspeita permanente).
- **Tempo de recorrência de erros** (*mistake recurrence time* - $T_{MR(i,j)}$): variável randômica que representa o tempo transcorrido entre duas falsas suspeições consecutivas.
- **Tempo em erro** (*mistake duration* - $T_{M(i,j)}$): variável randômica que representa o tempo no qual um processo permanece suspeitando (erroneamente) de um outro processo.

A Figura 9 ilustra as medidas de QoS definidas acima.

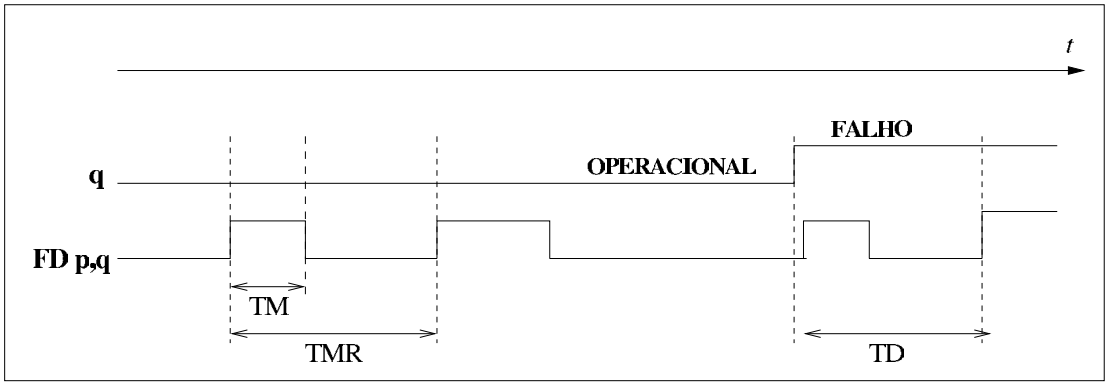


Figura 9: Medidas de QoS

Em outro trabalho [36], Chen descreve os detalhes técnicos relativos à teoria de processos estocásticos sobre as medidas apresentadas. É interesssante ressaltar que tais medidas são também utilizadas como uma forma de possibilitar a comparação do desempenho de diferentes implementações de detectores de falhas.

5 Análise de Desempenho de Protocolos Baseados em Detectores de Falhas

Atualmente, diversos trabalhos publicados na literatura tem sido dedicados à análise de desempenho de protocolos distribuídos que usam detectores de falhas não confiáveis [37,

38, 39, 40]. A maioria dos resultados apresentados até então, analisam como a QoS do serviço de detecção, ou a carga extra de comunicação gerada pela implementação do mesmo, influencia o desempenho de protocolos de consenso[4]. Além disso, estes resultados são obtidos de experimentos realizados sobre um ambiente dedicado, onde, apenas o serviço de detecção e o protocolo de consenso consomem recursos do sistema, *i.e.* um ambiente onde não existe nenhuma outra fonte de contenção de recursos, seja de processamento ou comunicação.

Sergent *et al.* [37] discutem o impacto de diferentes implementações de detectores de falhas sobre o desempenho do protocolo de consenso proposto por Chandra e Toueg em seu artigo clássico [10]. Neste trabalho considera-se a contenção de recursos do sistema por parte do serviço de detecção, permitindo, assim, uma análise de desempenho mais realística. Coccoli *et al.* [38] também estudam o desempenho deste protocolo de consenso, no caso, analisam o impacto de falhas, como também, da QoS do serviço de detecção sobre o desempenho do protocolo. Os resultados foram obtidos por meio de simulação e medidas experimentais coletadas a partir de uma implementação do protocolo considerando três classes de execução: i) nenhuma ocorrência de falhas, e nenhum caso de falsa suspeição; ii) alguma ocorrência de falhas, mas, nenhum caso de falsa suspeição; e iii) nenhuma ocorrência de falhas, e casos de falsas suspeição. O protocolo de consenso apresentou baixo desempenho no cenário com falsas suspeições. Hayashibara *et al.* [39] também analisam o impacto de falhas sobre o desempenho de protocolos de consenso. Este artigo compara o desempenho do protocolo de Chandra e Toueg [10] e o protocolo de consenso denominado de Paxos [42, 43], assumindo um detector de falhas que não comete erros (falsas suspeições) e considerando execuções com/sem ocorrências de falhas.

Seguindo uma abordagem diferente dos trabalhos supracitados, Sampaio *et al.* [40] analisam o desempenho de protocolos de consenso em um ambiente não dedicado, onde o poder de processamento e a largura de banda de rede disponíveis para os processos executando um protocolo de consenso (e o serviço de detecção requerido) não são conhecidos a priori, e não são necessariamente os mesmos para todos os processos. Como o modelo de sistema é assíncrono, o ambiente considerado neste trabalho é mais realístico. Os resultados demonstram que questões de implementação dos detectores de falhas[5] devem ser tratadas separadamente das questões de eficiência dos protocolos baseados em tal serviço. A partir deste resultado, os autores propõem que os protocolos de alto nível baseados em detectores de falhas não confiáveis sejam definidos de maneira a conseguirem se adaptar às condições do ambiente no qual executam. Nesse sentido, apresentam o conceito de *slowness oracles*[6], e também, um exemplo de protocolo adaptativo baseado nessa abordagem.

De fato, a idéia de separar aspectos de eficiência dos aspectos de projeto de protocolos baseados em detectores de falhas não confiáveis já havia sido comentada em [37]. Neste trabalho, os autores mostram que, resolver o consenso em sistemas assíncronos com detectores

[4] O interesse em relação aos protocolos de consenso é justificado pelo fato de que estes são considerados importantes construtores para a implementação de diversos protocolos de acordo tolerantes a faltas [10, 41].

[5] Um ponto crucial no projeto e implementação de serviços de detecção de falhas não confiáveis refere-se à definição dos temporizadores utilizados para identificar processos falhos. Isto conduz ao dilema entre evitar falsas suspeições (aumentando os valores para os temporizadores) e minizar o tempo de detecção de falhas reais (diminuindo os valores para os temporizadores) [37, 38, 39].

[6] *Slowness oracles* podem ser definidos como oráculos que informam sobre a situação de carga do sistema, idenficando processos lentos ou rápidos ao longo do período de operação do sistema.

de falhas não confiáveis não exclui a necessidade de considerar aspectos temporais, normalmente associados à implementação dos detectores de falhas. O próprio modelo de Chandra e Toueg [10] favorece essa "separação de conceitos", no caso, separação entre aspectos lógicos da aplicação (prova sobre a corretude da aplicação) e aspectos de engenharia (desempenho, restrições de tempo). Em se tratando de aspectos de engenharia, é importante considerar a implementação do detector de falhas associada ao protocolo que usa o detector [37]. Sergent *et al.* observaram melhores índices de desempenho do consenso quando se utilizaram implementações *ad hoc* do detector de falhas, onde, o serviço de detecção é projetado especialmente para um protocolo. Note que, nesse caso, o desempenho do protocolo está associado à implementação do detector de falhas. A abordagem de Sampaio *et al.* [40] apóia a "separação conceitos", mas, defende que, aspectos de desempenho dos protocolos de alto nível são mais influenciados por características do protoclo do que por características das implementações do serviço de detecção de falhas.

6 Considerações Finais

Neste trabalho foram apresentados os principais aspectos associados à teoria e prática de detectores de falhas não confiáveis. Além disso, foram abordados algumas pesquisas sobre análise de desempenho de protocolos distribuídos baseados em tais detectores de falhas. De fato, observamos vários resultados de implementações de detectores de falhas não confiáveis e estudos sobre o desempenho de protocolos que utilizam detectores de falhas desta natureza, como também, foram apresentadas novas abordagens no sentido de melhorar o desempenho dos referidos protocolos.

A grande maioria das implementações de detectores de falhas não confiáveis são classificadas como $\diamond S$. Este resultado se justifica pelo fato da classe $\diamond S$ oferecer as condições mínimas (em termos de detecção de falhas) para resolver o consenso em sistemas assíncronos [10]. Além disso, serviços com a semântica $\diamond S$ são mais simples de implementar. Por outro lado, é crescente o número de aplicações que requerem um maior nível de confiança no funcionamento e conseqüentemente exigem mecanismos mais robustos de tolerância a faltas, e ainda, precisam ser implementadas em ambientes assíncronos. Tais mecanismos são baseados em serviços de detecção que não podem cometer erros, ou seja, serviços de detecção perfeitos. Nesse caso, faz-se necessário usar detectores de falhas da classe P. De fato, existem poucos trabalhos na literatura sobre implementações de detectores de falhas da classe P, isto porque, serviços com semântica perfeita são mais difíceis de implementar. Portanto, o grande desafio nessa área é propor implementações de serviços de detecção da classe P que exijam poucas restrições sobre o ambiente onde executarão, contemplem aspectos de qualidade de serviço e permitam baixo acoplamento com as aplicações clientes. Caminhando nesta direção tem-se o projeto Delphus [32]

No que diz respeito aos resultados sobre análise de desempenho de protocolos que usam detectores de falhas não confiáveis, os desafios são ainda maiores. Uma grande parte da comunidade acadêmica tem investido em aspectos de implementação de detectores de falhas na tentativa de favorecer um melhor desempenho dos protocolos de alto nível. É sabido que definir os *timeouts* utilizados em um serviço de detecção de falhas não é uma tarefa fácil [37, 38, 39]. Além disso, bons valores de *timeouts* que favoreçam um certo equilíbrio

entre o número de falsas suspeições e o tempo de detecção de falhas reais, não implicam, necessariamente, em melhoria no desempenho dos protocolos de alto nível. Em alguns casos, o desempenho pode até ser menor [40]. Parece sugestivo que questões de desempenho sejam tratadas separadamente das questões de implementação dos detectores de falhas. E ainda, o desempenho dos detectores de falhas está relacionado com sua capacidade de detecção de falhas e não deve influenciar no desempenho dos protocolos de alto nível, como muitos acreditam. Alguns autores defendem a idéia de tratar questões de desempenho dos protocolos baseados em detectores de falhas, no nível dos próprios protocolos. Nesse caso, a implementação de serviços de detecção de falhas e o desempenho dos protocolos que usam tais serviços são coisas distintas e podem ser tratadas separadamente. Sampaio *et al.* [40] propõem uma forma de definir protocolos adaptativos em ambientes assíncronos com detectores de falhas não confiáveis, visando a um melhor desempenho de tais protocolos em ambientes não dedicados.

Acreditamos que os dois tópicos comentados acima, implementações de detectores de falhas perfeitos e análise de desempenho de protocolos que usam detectores de falhas, revelam tendências para os próximos trabalhos sobre detectores de falhas não confiáveis.

Referências

[1] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[2] R. Baldoni, S. Bonamoneta, and C. Moarchetti. Implementing highly available www servers based on passive object replication. Technical Report TR14-98, Università di Roma "La Sapienzza", December 1998.

[3] R. Guerraoui and A. Schiper. Consensus: The big misunderstanding. In *Proceedings of the 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)*, pages 183–188, Tunis, Tunisia, Oct 1997.

[4] R. Guerraoui and A. Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, Jan 2001.

[5] J. Turek and D. Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, 25(6):8–17, June 1992.

[6] 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, April 1985.

[7] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[8] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[9] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, June 1999.

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

[11] 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.

[12] R. Guerraoui and A. Schiper. Gamma-accurate failure detectors. In *Proceedings of the* 10^{th} *International Workshop on Distributed Algorithms (WDAG-10), LNCS 1151*, Bologna, Italy, Oct 1996.

[13] Laura S. Sabel and Keith Marzullo. Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Cornell University, February 1995.

[14] C. Fetzer and F. Cristian. A highly available local leader election service. *IEEE Transactions on Software Engineering*, 25(5):603–618, 1999.

[15] C. Fetzer and F. Cristian. Fail-aware failure detectors. In *The* 15_{th} *Symposium on Reliable Distributed System, SRDS-1996*, pages 200–209, Niagara-on-the-Lake, Ontario, Canada, October 1996.

[16] A. B. de H. Ferreira. Novo aurélio, o dicionário da língua portuguesa, 2003.

[17] P. Felber, R. Guerraoui, X. Défago, and P. Oser. Failure detector as first class objects. In *International Symposium on Distributed Objects and Applications*, L'Aquila, Italy, Sep 1999.

[18] Notification service specification, version 1.0. Object Management Group, Jun 2000.

[19] P. Felber, B. Garbinato, and R. Guerraoui. The design of a CORBA group communication service. In *Proceedings of the* 15^{th} *IEEE Symposium on Reliable Distributed Systems*, pages 150–160, Niagara-on-the-lake, Canada, Oct 1996.

[20] P. Felber, R. Guerraoui, and A. Schiper. The implementation of a CORBA group communication service. *Theory and practice of object systems*, 4(2):93–105, 1998.

[21] P. Chungm, Y. Huang, S. Yajnik, D. Liang, and J. Shih. DOORS: providing fault tolerance to CORBA objects. In *The IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 98')*, The Lake District, England, 1998. poster session.

[22] B. Natarajan, A. S. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: Towards high-performance fault tolerant CORBA. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA2000)*, pages 39–48, Antwerp, Belgium, September 2000.

[23] M. S. G. Batalha. Serviço CORBA de diagnóstico de falhas. Dissertação de mestrado, COPIN - Universidade Federal da Paraíba, Campina Grande, outubro 2001.

[24] E. W. A. de Oliveira. Monitor - serviços de diagnóstico de falhas e gerenciamento de aplicações distribuídas. Monografia de curso de especialização, Universidade Federal da Bahia, Salvador, novembro 2001.

[25] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: specification and implementation. In *Proceedings of the 3rd European Dependable Computing Conference (EDCC-3), LNCS 1667*, Prague, Czech Republic, Sep 1999.

[26] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In *Proceedings of the 11th Workshop on Distributed Algorithms (WDAG'1997)*, pages 126–140, Saarbrücken,Germany, September 1997.

[27] S. Maffeis. Piranha: A CORBA tool for high availability. *Computer*, 30(4):56–66, April 1997.

[28] L. Sabel and K. Marzullo. Simulating fail-stop in asynchronous distributed systems. In *Proceedings of the 13th IEEE Symposium on Reliable Distributed Systems*, pages 138–147, 1994.

[29] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[30] Christof Fetzer. Enforcing perfect failure detection. In *Proceedings of the 21st International Conference on Distributed Systems*, Phoenix, AZ, 2001.

[31] C. Fetzer and F. Cristian. A fail-aware datagram service. In *Proceedings of the 2nd Annual Workshop on Fault Tolerant Parallel and Distributed Systems*, Geneva, Switzerland, April 1997.

[32] A. E. M. Brito, E. W. de Oliveira, and F. V. Brasileiro. Projeto e implementação de um serviço de detecção de falhas perfeito. In *a ser publicado nos Anais do Simpósio de Redes de Computadores*, Natal, Brasil, maio 2003.

[33] P. Veríssimo and A. Casimiro. The Timely Computing Base model and architecture. *Transactions on Computers - Special Issue on Asynchronous Real-Time Systems*, 51(8):916–930, Aug 2002.

[34] A. Casimiro, P. Martins, and P. Veríssimo. How to build a timely computing base using real-time linux. In *Proceedings of the IEEE International Workshop on Factory Communication Systems*, pages 127–1343, Porto, Portugal, September 2000. IEEE Industrial Electronics Society.

[35] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of sevice of failure detectors. In *International Conference on Dependable Systems and Networks (DSN'2000)*, pages 191–200, New York, USA, Jun 2000.

[36] W. Chen. *On the quality of service of failure detectors*. PhD thesis, Cornell University, 2000.

[37] N. Sergent, X. Défago, and A. Schiper. Impact of a failure detection mechanism on the performance of consensus. In *Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing (PRDC'2001)*, Seoul, Korea, Dec 2001.

[38] A. Coccoli, P. Urbán, A. Bondavalli, and A. Schiper. Performance analysis of a consensus algorithm combining stochastic activity networks and measurements. In *International Conference on Dependable Systems and Networks (DSN'2002)*, Washington, D.C., USA, Jun 2002.

[39] N. Hayashibara, P. Urbán, A. Schiper, and T. Katayama. Performance comparison between the paxos and chandra-toueg consensus algorithms. Technical Report IC-2002-61, École Polytechnique Fédérale de Lausanne, Switzerland, Aug 2002.

[40] L. M. R. Sampaio, F. V. Brasileiro, W. da C. Cirne, and J. C. A. de Figueiredo. How bad are wrong suspicions? towards adaptive distributed protocols. San Franciso, California, USA, June 2003. Proceedings of the International Conference on Dependable Systems and Networks (DSN'2003).

[41] R. Guerraoui, M. Hurfin, A. Mostefaoui, R. Oliveira, M. Raynal, and A. Schiper. Consensus in asynchronous distributed systems: A concise guided tour. *Springer LNCS: Advances in Distributed Systems*, (1752):33–47, 2000.

[42] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[43] R. D Prisco, B. Lampson, and N. Lynch. Revisiting the paxos algorithm. *Theoretical Computer Science*, 243(1-2):35–91, Jul 2000.

Difusão Atômica com Suporte à Perda de Mensagens*

Fabíola Gonçalves Pereira Greve

[1]LaSiD - Laboratório de Sistemas Distribuídos
Universidade Federal da Bahia
Campus de Ondina, 40170-110 Bahia, Brasil

fabiola@ufba.br

Resumo. *Apresentamos um protocolo de* difusão atômica *que suporta a perda de mensagens provenientes dos clientes e implementa diretamente a entrega atômica das mesmas sem recorrer ao uso de uma primitiva de* difusão confiável. *Ao nosso conhecimento, nenhum outro protocolo similar até então proposto apresenta tais mecanismos para lidar diretamente com as perdas. Este protocolo foi obtido a partir da especialização de um serviço genérico de consenso. Além disso, ele foi utilizado na implementação do componente de replicação ativa da biblioteca de componentes de acordo* ADAM *[1].*

Abstract. *We provide an efficient and realistic* atomic broadcast *protocol which supports the loss of messages from clients. As soon as we know, this is the only protocol proposed in the literature that deals directly with losses without using the* reliable broadcast *primitive as a resource to deliver messages in a reliable manner. This protocol was designed as a specialization of a general agreement framework. Moreover, it was used in the implementation of an active replication component that belongs to a library of agreement components called* ADAM *[1].*

1. Introdução

Uma maneira clássica de tornar um servidor confiável consiste em replicá-lo em diferentes máquinas de um sistema distribuído. O estado do servidor é compartilhado entre as réplicas que executam ações coordenadas a fim de implementar o serviço requerido. Se as máquinas falham de maneira independente, a invocação do serviço pelo cliente será bem sucedida mesmo se algumas das réplicas falham antes de terminar as ações requeridas. Na técnica da *replicação ativa* [2] todas as réplicas têm o mesmo papel. Para preservar o estado coerente do servidor, uma primitiva de *difusão atômica* [3] deve ser usada a fim de garantir que as mensagens provenientes dos clientes sejam entregues numa mesma ordem total ao grupo de servidores.

Neste artigo, apresentamos um protocolo de difusão atômica obtido a partir de uma redução a um serviço genérico de consenso, de nome GAF (*General Agreement Framework*) [4]. O problema do consenso é um denominador comum entre diversos problemas práticos presentes na concepção de sistemas tolerantes a faltas. As soluções baseadas no consenso são atrativas, tanto do ponto de vista prático quanto teórico, pois além do caráter modular e elegante, exibem uma caracterizarão precisa das propriedades de *liveness* (vivacidade) e *safety* (precisão) ligadas aos problemas.

O protocolo de difusão atômica apresentado suporta a perda de mensagens provenientes dos clientes. Ao nosso conhecimento, nenhum outro protocolo similar até então proposto [5,

*Este trabalho foi realizado com financiamento parcial do CNPQ/Brasil (200.323-97).

6, 7, 8, 9, 10] fornece mecanismos para lidar diretamente com a possibilidade de tais perdas. Todos eles se fundamentam na existência de canais de comunicação confiáveis e/ou se baseiam no uso de uma primitiva de difusão confiável [3] para a entrega segura destas mensagens. A implementação de tal primitiva tem um alto custo: para cada mensagem proveniente do cliente, $O(n^2)$ mensagens são retransmitidas às n réplicas do servidor. Trabalhos foram propostos com o intuito de diminuir o custo da difusão confiável [11]. Nosso interesse, entretanto, é o de evitar o seu uso na implementação da difusão atômica.

Nas próximas seções, descrevemos inicialmente o modelo de replicação considerado (seção 2.), em seguida fornecemos uma descrição sucinta dos parâmetros necessários à utilização do framework GAF (seção 3.). Posteriormente, descrevemos o protocolo de difusão atômica (seção 4.) e finalmente concluimos.

2. Modelo de Replicação Ativa

Designamos por S um servidor único particular. Para tolerar falhas definitivas (do inglês, *crash*) [12, 5], o servidor é replicado em n processos: cada processo p_i executa uma réplica de S. $\Pi = \{p_1, ..., p_n\}$ é então visto como o grupo de processos associados ao servidor S. A cardinalidade $|\Pi| = n$ é o grau de replicação do servidor. Os processos se comunicam e cooperam pela emissão de mensagens através de canais de comunicação segundo um modelo de comunicação assíncrono. Os canais não criam, não alteram, nem duplicam as mensagens que ali trafegam. Entretanto, eles admitem perdas de mensagens de forma eqüitativa. Ou seja, se um processo p_i envia a um processo correto p_j uma mensagem m uma infinidade de vezes, então p_j recebe m uma infinidade de vezes. Tal modelo de falhas é conhecido em inglês como *fair-lossy* [13]. Finalmente, um *processo correto* é aquele que obedece à sua especificação e não falha durante toda a execução do sistema.

2.1. O Problema do Consenso

Informalmente o problema do consenso [12, 5] é definido da seguinte maneira: cada processo correto p_i propõe um valor v_i e todos os processos corretos devem "decidir"por um único valor v, escolhido dentre aqueles que foram propostos. Esse problema fundamental não tem solução determinista num sistema assíncrono, mesmo em presença de uma única falha [12]. Uma das estratégias adotadas para contornar tal resultado de impossibilidade consiste em estender o modelo assíncrono com algum grau de "sincronismo". Com este intuito, um dos avanços mais significativos é a proposta de uso dos *detectores de falhas não confiáveis* [5].

2.2. Detecção de Falhas

Informalmente, um detector de falhas não confiável é um conjunto de "oráculos"que fornece dicas aos processos sobre quais deles estão falhos. A cada processo p_i é associado um módulo de detecção que fornece informações de falhas através de uma lista, chamada $suspected_i$, contendo os processos suspeitos. Como em [5], consideramos que "p_i suspeita um processo p_j se $p_j \in suspected_i$ num instante t". Qualquer implementação de detector que satisfaça às exigências acima descritas pode ser usada. Um protocolo simples que emite mensagens periódicas do tipo "eu estou vivo", e utiliza um mecanismo de temporização para registrar as suspeitas, atende a essas exigências.

Chandra e Toueg [5] definem formalmente algumas classes de detectores. A classe $\Diamond\mathcal{S}$ (*eventually strong*) garante que todo processo falho será finalmente suspeito por todos os processos corretos; além disso, existirá um instante a partir do qual algum processo correto não será considerado suspeito por nenhum outro processo correto. O interesse da classe $\Diamond\mathcal{S}$ reside na sua importância para a resolução do consenso: ela representa a classe de detectores mais fraca

a permitir uma resolução do consenso e à condição que uma maioria de processos esteja correta ($f < n/2$) [14]. Os algoritmos aqui apresentados consideram o modelo assíncrono estendido com os detectores de falhas do tipo $\Diamond\mathcal{S}$; além disso, supõe-se que exista pelo menos uma maioria de processos corretos no sistema ($f < n/2$).

2.3. Difusão Atômica

Informalmente, um serviço de difusão atômica (*atomic broadcast*) assegura que os processos de um grupo de servidores entregarão um mesmo conjunto de mensagens enviadas por clientes e na mesma ordem total. Formalmente, esse serviço é definido da seguinte maneira [3, 5]:

- AB_Terminação: se um processo correto *envia* uma mensagem m então todos os demais processos corretos *entregam* m;
- AB_Integridade: um processo só *entrega* uma mensagem no máximo uma vez;
- AB_Acordo_Uniforme: se um processo *entrega* uma mensagem m então todos os demais processos corretos também entregam m;
- AB_Validade: se um processo *entrega* uma mensagem m então algum processo *enviou* m;
- AB_Ordem_Total: se dois processos p_i e p_j *entregam* as mensagens m e m', então p_i entrega m antes de m', se e somente se, p_j entrega m antes de m';

3. GAF: um Framework para Especialização de Protocolos de Acordo

GAF (do inglês, G*eneral* A*greement* F*ramework*) [4] é um framework genérico para a realização de um acordo a partir do protocolo clássico de Chandra e Toueg [5]. O framework dispõe de alguns parâmetros genéricos que devem ser instanciados de maneira estática (em tempo de compilação) para a geração automática dos protocolos. Os processos que desejam construir uma lógica de acordo particular devem instanciar estes parâmetros com valores adaptados à semântica do problema. A seguir, descrevemos sucintamente os principais parâmetros de GAF.

- GET: graças a esta função a camada da aplicação irá transmitir ao framework as proposições para o acordo. Durante a execução do protocolo de acordo, esta função poderá ser chamada diversas vezes. Assim, um processo pode mudar seu valor de proposição sempre que ele desejar. Isto permite, dentre outros, que valores cada vez mais significativos sejam propostos sem necessidade de esperar pelo fim de um consenso.
- $\mathcal{F}$: função aplicada sobre o conjunto de valores propostos (de entrada) e cujo objetivo é o cálculo do valor de decisão (de saída). Ela é aplicada quando uma quantidade suficiente de proposições foram recolhidas ao longo do consenso.
- ACCEPTABLE: predicado cujo objetivo é a verificação da aceitação do valor escolhido (decisão efetuada); graças a ele, um processo pode participar a um consenso sem que ele possua um valor significativo.
- EXCUSED: predicado que autoriza um processo a não participar do consenso. Ele estabelece circunstâncias em que o valor proposto pelo processo é dispensável. Por exemplo, para resolver o problema do consenso propriamente dito, basta que uma maioria de valores seja coletada.

4. Difusão Atômica com Suporte à Perda de Mensagens

Grande parte dos protocolos de difusão atômica propostos a partir de uma redução a um serviço de consenso [5, 6, 7, 8] utilizam canais confiáveis. Na prática, a abstração de canais confiáveis exige meios de comunicação seguros nos quais a mínima perda de mensagens da aplicação torna-se

inaceitável. Infelizmente, tal contexto não corresponde às características das redes e dos ambientes de computação atuais. Além disso, mesmo quando esses protocolos admitem a perda de mensagens [9, 10], eles costumam se apoiar no uso de uma primitiva de difusão confiável (*reliable broadcast*) [3] para propagar ao grupo de servidores as mensagens provenientes dos clientes. Quando um processo recebe uma mensagem m pela primeira vez, ele difunde m aos demais membros do grupo e somente depois é que ele entrega m localmente. Tal primitiva garante que uma mensagem enviada será entregue por todos os processos corretos ou por nenhum deles (atomicidade na entrega). Sua implementação é entretanto custosa: para cada mensagem enviada pelo cliente, $O(n^2)$ mensagens serão transmitidas ao grupo. Como na prática a maioria das mensagens não se perdem, o uso sistemático de tal primitiva custosa deveria e pode ser evitado.

Com o objetivo de conceber uma solução tão realista quanto eficaz, propomos um protocolo de difusão atômica que admite a perda de mensagens de maneira eqüitativa, ou seja, estaremos considerando canais do tipo *fair-lossy*; além disso, diferentemente de todos os demais protocolos, não faremos uso da primitiva de difusão confiável para difundir as mensagens provenientes dos clientes. Um mecanismo de retransmissão de mensagens adequado, que utiliza as facilidades do próprio serviço de consenso, se encarregará de garantir por um lado, a atomicidade da entrega das mensagens e, por outro lado, o re-envio das mensagens somente aos processos que não as possuem.

Interface com a aplicação. Supomos que os clientes geram um fluxo contínuo de requisições através de alguma primitiva de difusão ao grupo de servidores. O processo servidor que recebe uma mensagem m do cliente chama o procedimento A_BROADCAST(m) para divulgar m de maneira atômica a todos os demais membros do grupo. Os processos entregam as mensagens localmente por intermédio do procedimento A_DELIVER(), chamado pela camada da aplicação responsável pela execução das requisições.

Princípio. A ordenação das mensagens emitidas pelos clientes, e disseminadas a partir da primitiva A-BROADCAST(), é realizada passo a passo pelo protocolo. A cada passo, várias novas mensagens são ordenadas graças ao consenso. Por motivos de eficiência, somente as identidades das mensagens são passadas para o acordo. Para dar início a um novo consenso, espera-se o fim do anterior. Cada nova seqüência de mensagens ordenadas estende a seqüência global das mensagens anteriormente observadas. A fim de garantir a entrega atômica das mensagens emitidas pelos clientes (propriedade AB_Acordo_Uniforme) somente mensagens propostas por uma "maioria"de processos estarão sendo ordenadas. Isto porque, em caso de perda, garante-se que existirá ao menos um processo correto que poderá retransmitir a mensagem.

À demanda da aplicação, a partir da operação A-DELIVER(), o protocolo entrega a primeira mensagem ordenada que esteja disponível. Os processos que ainda não receberam da rede as mensagens já ordenadas esperam que elas sejam recebidas antes de entregá-las à aplicação. Um mecanismo de retransmissão de mensagens foi incorporado para evitar que os processos fiquem bloqueados à espera de mensagens perdidas.

Funcionamento. O protocolo está ilustrado pela figura 1. Cada processo p_i controla localmente as seguintes variáveis:

- k: o número do consenso atual;
- $Received_i$: conjunto de mensagens recebidas dos clientes;
- $A\_Delivered_i$: conjunto de identidades de todas as mensagens ordenadas dentro do grupo;

- $Undelivered_i$: conjunto de identidades das mensagens já ordenadas mas que ainda não foram entregues à aplicação. Neste conjunto as mensagens são classificadas primeiramente em função do número do consenso onde elas foram decididas e em seguida pela aplicação de uma função determinista, conhecida a priori por todos os participantes (e.g. a identidade das mensagens).

O protocolo é composto de dois *threads* concorrentes: AB1 et AB2. O *thread* AB1 se ocupa da interface com a aplicação e com a camada de comunicação. O *thread* AB2 efetua a ordenação das mensagens com ajuda do protocolo GAF e atualiza o estado local do processo a partir das decisões tomadas.

Thread AB1: interface com a aplicação e com a camada de comunicação. Toda mensagem recebida (pela invocação da operação A_BROADCAST(m)) será diretamente armazenada em $Received_i$ (linhas 2-3). Diferentemente dos protocolos clássicos de difusão atômica, as mensagens recebidas dos clientes não são difundidas no início do protocolo por uma primitiva de difusão confiável. Quando a operação A_DELIVER() é solicitada, a primeira mensagem no topo da fila $Undelivered_i$ é entregue para a aplicação; esta entrega só é efetuada após a efetiva recepção da mensagem por parte do processo (linhas 5-6). O processo que tenha recebido REQUEST_MSG(p_j, m)(linha 7) retransmite a mensagem m ao processo p_j que a solicitou através do envio de A_MSG(m) (linha 8). O processo que recebe esta mensagem A_MSG(m) (linha 9) armazena diretamente m em $Received_i$ (linha 10).

Thread AB2: ordenação das mensagens. Cada processo p_i lança, em *background*, o serviço de acordo GAF (linha 13) para o cálculo do novo conjunto de mensagens. Identificamos pela variável k cada consenso efetuado, sabendo-se que um consenso de número k não começa antes que o consenso anterior (de número $k-1$) tenha terminado. O processo inicia ou participa do acordo k somente quando ele possui um valor significativo a propor. Cada valor representa as mensagens recebidas localmente, mas ainda não ordenadas: a diferença entre os conjuntos $Received_i$ e $A\_Delivered_i$. Este valor é o resultado retornado pela função GET() (linhas 16-17). Quando o acordo k termina, o processo atualiza os subconjuntos locais a partir do novo conjunto de mensagens decidido ($Decided^k$) pela chamada ao procedimento Entrega-Confiável (linha 14).

O procedimento Entrega-Confiável As mensagens de $Decided^k$ que ainda não foram entregues (linha 18) são adicionadas à fila $Undelivered_i$ (linha 19) e em seguida ao conjunto $A\_Delivered_i$ (linha 20). Devido à possibilidade de perda de mensagens, algumas das mensagens podem ser recebidas somente por alguns processos. Dois casos devem ser então considerados:

- Caso a [um processo recebeu do cliente uma mensagem que os outros ainda não receberam] – a cada vez que uma decisão é tomada, o processo p_i verifica localmente se a decisão levou em conta todo o subconjunto de mensagens que ele havia proposto ao consenso. Caso esta condição não se verifique ($m \in Proposed^k$ e $m \notin Decided^k$), p_i difunde a mensagem do cliente ao grupo por intermédio da primitiva não confiável *broadcast*(m).

- Caso b [um processo não recebeu do cliente a mensagem que outros receberam] – quando um processo p_i verifica que uma mensagem foi ordenada sem que ele a tenha recebido ($m \in Decided^k$ e $m \notin Received_i$), ele solicita ao grupo, pela emissão de REQUEST_MSG(p_i, m), a mensagem que lhe falta. Na prática, esta solicitação ao grupo pode ser substituída por uma protocolo de requisição ponto-a-ponto até que a mensagem m seja obtida por p_i. Como supõe-se uma maioria de processos corretos, p_i terminará por receber m.

Atomic Broadcast
(1) $Received_i \leftarrow \emptyset$; $A\_Delivered_i \leftarrow \emptyset$; $Undelivered_i \leftarrow \emptyset$; $k \leftarrow 0$;
cobegin
thread AB1:
% Interface com a aplicação %
(2) **upon** a call to A_BROADCAST(m) **do**
(3) $Received_i \leftarrow Received_i \cup \{m\}$; **enddo**
(4) **upon** a call to A_DELIVER() **do**
(5) **wait until** ($Undelivered_i \neq \emptyset$); $m \leftarrow$ remove_first($Undelivered_i$);
(6) **wait until** ($m \in Received_i$); *return* (m); **enddo**
% Interface com a camada de comunicação %
(7) **upon** reception of REQUEST_MSG (p_j, m) **do**
(8) **if** ($m \in Received_i$) **then** *send* A_MSG (m) to p_j; **endif enddo**
(9) **upon** reception of A_MSG(m) **do**
(10) $Received_i \leftarrow Received_i \cup \{m\}$; **enddo**

thread AB2: % Cálculo do conjunto de mensagens e da sua ordem de entrega %
(11) **while** ($true$) **do**
(12) $k \leftarrow k + 1$;
(13) $Decided^k \leftarrow$ GAF();
(14) Entrega_Confiável ($Decided^k$);
(15) **enddo**
coend

Function GET() % Determina um valor de proposição significativa %
(16) **wait until** ($Received_i \setminus A\_Delivered_i \neq \emptyset$) *or* (*expired timeout*);
(17) $Proposed^k \leftarrow Received_i \setminus A\_Delivered_i$; $return(Proposed^k)$;

Procedure Entrega_Confiável($Decided^k$)
(18) $Ordered^k \leftarrow Decided^k \setminus A\_Delivered_i$;
(19) Queue ($Undelivered_i, Ordered^k$) % coloca mensagens na fila do conjunto %
(20) $A\_Delivered_i \leftarrow A\_Delivered_i \cup Ordered^k$;
(21) **for each** ($m \in Proposed^k \setminus Decided^k$) **then**
(22) *broadcast*(m); **endif** % difunde as mensagens recebidas mas ainda não ordenadas %
(23) **for each** ($m \in Decided^k \setminus Received_i$) **then**
(24) *broadcast* REQUEST_MSG(p_i, m); **endif** % solicita mensagem que falta %

Figura 1: Difusão Atômica com Perda de Mensagens

Definição dos Parâmetros para o Framework GAF As principais funções de GAF para resolver a difusão atômica são apresentadas na tabela 1 e são descritas a seguir.

• A função $\mathcal{F}$: a escolha desta função é crucial para a satisfação das propriedades AB_Acordo_Uniforme e AB_Terminação definidas para o problema. Para garantir o acordo é importante considerar no cálculo da decisão as mensagens propostas por ao menos uma "maioria"de processos. Dado que supostamente uma maioria é correta (condição imposta para permitir a resolução do consenso) então ao menos um processo entre aqueles que possuem a mensagem será correto e poderá retransmiti-la futuramente aos outros processos que ainda não a receberam. Assim, a função $\mathcal{F}$ é definida como sendo a *intersecção* das proposições coletadas.

• A função ACCEPTABLE(v) – Ela retorna $verdadeiro$ quando o valor v não é o conjunto vazio $\emptyset$. Sua aplicação é importante, pois a função $\mathcal{F}$ definida anteriormente pode gerar um valor $\emptyset$. Se isso acontece, este valor será rejeitado pelos processos e o protocolo continuará a ser executado até que uma decisão válida possa ser tomada.

• A função GET() – Retorna o conjunto de mensagens recebidas localmente pelo processo mas que ainda não foram ordenadas: $Received_i \setminus A\_Delivered_i$ (linhas 16-17). Devido à possibilidade de perda de mensagens, alguns processos podem ter recebido mensagens provenientes do cliente, enquanto outros não as receberam. Assim, alguns processos darão início ao consenso enquanto outros restarão bloqueados à espera de um valor de entrada significativo ($\neq \emptyset$). Para evitar tal bloqueio do protocolo GAF, autorizamos valores de proposição não significativos ($= \emptyset$). Isto é necessário, pois a chamada ao procedimento de retransmissão de mensagens perdidas (linhas 21-22), que poderia eventualmente desbloquear GAF, se faz somente quando uma decisão é tomada. O framework GAF autoriza assim um processo a propor inicialmente um conjunto vazio ($v = \emptyset$) e posteriormente a completar este valor inicial à medida que novas mensagens chegam ao processo durante a execução do protocolo. Vale ressaltar que se proposições de conjuntos vazios são freqüentemente emitidas, podemos comprometer a qualidade do acordo realizado; isto é, valores não significativos serão decididos. Uma só proposição vazia irá tornar inútil o cálculo da decisão efetuada pela aplicação da função $\mathcal{F}$. Para evitar tal situação, antes de propor um valor não significativo, os processos deverão esperar pela expiração de um valor de *timeout*. Este valor é definido em função do tempo estimado de transmissão de uma mensagem na rede; como ele é utilizado localmente, os processos poderão modificá-lo em função da percepção que possuam do comportamente da rede de comunicação.

Parâmetro	Descrição
GET	Retorna ($Received_i \setminus A\_Delivered_i$) ou (*expiração de timeout*)
$\mathcal{F}$	Retorna intersecção de todos os conjuntos de entrada: $\bigcap Proposed_i$
ACCEPTABLE	Retorna $verdadeiro$ quando não é aplicado sob um subconjunto vazio
EXCUSED	EXCUSED(p_i) retorna $verdadeiro$ após a expiração de um certo tempo (timeout)

Tabela 1: Parâmetros GAF para a Difusão Atômica com Suporte à Perda de Mensagens

Estabilização das Mensagens. O procedimento de estabilização de mensagens atende a dois objetivos: i) retransmissão/recuperação de mensagens perdidas e ii) eliminação local de mensagens estáveis, i.e., que tenham sido recebidas da rede por todos os processos do grupo. Nesse último caso, evita-se o crescimento infinito dos conjuntos de mensagens utilizados pelo protocolo. As mensagens ordenadas são entregues à aplicação sem necessidade de esperar que elas sejam estáveis no grupo. Tem-se, entretanto, que ao menos uma maioria de processos as possuem, pois o protocolo adia as ordenações até que uma maioria de processos as tenham recebido. O procedimento de estabilização utiliza então as informações provenientes do próprio acordo para evitar a retransmissão inútil de mensagens. Inicialmente, um processo que recebe uma mensagem do cliente espera o fim do próximo acordo para então retransmiti-lo aos outros membros (no caso

da mensagem não ter sido ordenada). Para as mensagens que já foram ordenadas, elas só serão re-enviadas aos processos que não as receberam da rede.

5. Conclusão

Apresentamos um protocolo original de difusão atômica que, diferentemente dos demais protocolos similares até então propostos, considera a possibilidade de perda de mensagens dos clientes e implementa diretamente a entrega atômica das mensagens sem fazer uso da primitiva de difusão confiável. O algoritmo obtido foi utilizado na confecção e implementação do componente de replicação ativa da biblioteca de componentes de acordo ADAM [1]. Este componente encontram-se atualmente em fase de testes e de avaliação de desempenho. Pretendemos, através desta análise quantitativa, corroborar nossas afirmações com relação à eficiência do protocolo proposto.

Agradecimentos

A autora gostaria de agradecer a contribuição de Michel Hurfin e de Frederic Tronel, pesquisadores da equipe *Adep* do IRISA-INRIA, França, a este trabalho.

Referências

[1] F. G. P. Greve, *Réponses efficaces au besoin d'accord dans un groupe*. PhD thesis, Université de Rennes I, France, Nov. 2002.

[2] F. Schneider, *Distributed Systems*, ch. Replication Management using the State Machine Approach, pp. 169–198. Addison-Wesley, 1993.

[3] V. Hadzilacos and S. Toueg, *Distributed Systems*, ch. Fault Tolerant Broadcasts and Related Problems, pp. 97–145. Addison-Wesley, 1993.

[4] M. Hurfin, R. Macêdo, M. Raynal, and F. Tronel, "A generic framework to solve agreement problems," in *Proc. of the* 19^{th} *IEEE Symposium on Reliable Distributed Systems (SRDS'99)*, (Lausanne, Switzerland), pp. 56–65, Oct. 1999.

[5] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of ACM*, vol. 43, pp. 225–267, Mar. 1996.

[6] E. Anceaume, "A lightweight solution to uniform atomic broadcast for asynchronous systems," in *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, (Washington - Brussels - Tokyo), pp. 292–303, IEEE, June 1997.

[7] F. Pedone and A. Schiper, "Optimistic atomic broadcast," in *Proceedings of the 12th International Symposium on Distributed Computing (DISC'98, formerly WDAG)*, Sept. 1998.

[8] A. Mostefaoui and M. Raynal, "Low-cost consensus based atomic brodcast," in *Proceedings of IEEE Pacifi Rim Intern. Symposium on Dependable Computing (PRDC-00)*, (Los Angeles, CA), IEEE, Dec. 2000.

[9] R. Guerraoui and A. Schiper, "Consensus service: A modular approach for building fault-tolerant agreement protocols in distributed systems," in *Proceedings of the 26th International Symposium on Fault-Tolerant Computing (FTCS-26)*, (Sendai, Japan), pp. 168–177, June 1996.

[10] R. Guerraoui and A. Schiper, "The generic consensus service," *IEEE Transactions on Software Engineering*, vol. 27, pp. 29–41, Jan. 2001.

[11] L. Rodrigues and P. Veríssimo, "Topology-aware algorithms for large scale communication," *LNCS: Advances in Distributed Systems*, no. 1752, pp. 1217–1256, 2000.

[12] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of ACM*, vol. 32, pp. 374–382, Apr. 1985.

[13] A. Basu, B. Charron-Bost, and S. Toueg, "Simulating reliable links with unreliable links in the presence of process crashes," in *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG96)*, pp. 105–122, 1996.

[14] T. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *Journal of ACM*, vol. 43, pp. 685–722, July 1996.

A Prova Informal da Correção do Protocolo de Difusão Atômica

Na demonstração que se segue, estamos assumindo a correção do protocolo GAF [4]. A demonstração das propriedades de AB_Integridade e AB_Validade são triviais e ficam a cargo do leitor. A propriedade AB_Ordem_Total é satisfeita porque i) as mensagens são entregues por ordem linear de consenso e pela aplicação de uma função determinista sobre as suas identidades ii) pela propriedade *Acordo_Uniforme* do consenso, cada processo recebe o mesmo conjunto de mensagens ordenadas. Uma prova informal das demais propriedades é dada a seguir.

Teorema 1 AB_Acordo_Uniforme*: Se um processo* entrega *uma mensagem m então todos os demais processos corretos também entregam m;*

Prova Um processo só entrega mensagens que foram anteriormente ordenadas através do protocolo GAF. Pela propriedade de *Acordo_Uniforme* do consenso, todos os processos corretos terão acesso ao mesmo conjunto de mensagens ordenadas. Após a ordenação, estas mensagens serão incorporadas à fila $Undelivered_i$ (linhas 18-19), para serem posteriormente entregues a partir da execução da primitiva A_DELIVER() (linhas 4-6). Devido à possibilidade de perda de mensagens, alguns processos podem não ter recebido da rede estas mensagens; neste caso, após a ordenação, eles irão solicitá-las ao grupo (linhas 23-24). Sabe-se, pela definição da função $\mathcal{F}$ de GAF, que ao menos uma maioria de processos possui as mensagens. Além disso, supõe-se que uma maioria de processos seja correta, logo tem-se que ao menos um processo correto possuirá a mensagem que foi ordenada e poderá transmiti-la aqueles que não a possuem. Isso é realizado pelo protocolo das linhas 7-10. Desta maneira, todos os processos corretos terminarão por receber e entregar todas as mensagens ordenadas e o teorema segue. $Teorema\ 1$

Teorema 2 AB_Terminação*: se um processo correto* envia *uma mensagem m então todos os demais processos corretos* entregam *m;*

Prova As mensagens são enviadas ao grupo de processos pela invocação da primitiva A-BROADCAST(). Toda mensagem m recebida através desta primitiva é armazenada em $Received_i$ (linhas 2-3). Posteriormente, ela será proposta ao consenso, através da função GET() (linhas 16-17). Sabe-se, pela definição da função $\mathcal{F}$ de GAF, que nem todas as mensagens propostas ao consenso serão ordenadas. Nesse caso, através do protocolo definido nas linhas 21-22, o processo fará a retransmissão de m até que ela venha a ser ordenada. Como assume-se a existência de canais com perdas eqüitativas, eventualmente m será recebida por uma maioria de processos corretos. Nesse caso, pela definição de $\mathcal{F}$, m será ordenada; pelo Teorema 1, m será eventualmente entregue a todos os processos corretos.

$Teorema\ 2$

Guaranteeing Fault Tolerance through Scheduling on a CAN bus

M. P. Oliveira, A. O. Fernandes, S. V. A. Campos, A. L. A. P. Zuquim

Departamento de Ciência da Computação, Universidade Federal de Minas Gerais

Caixa Postal 702 - 30123-970 - Belo Horizonte - MG - Brasil
{marcospo, otavio, scampos, ana@dcc.ufmg.br}

Abstract. *Prioritizing tasks in Hard-Real-Time Systems is a problem belonging to NP-hard class. Scheduling and resource allocation in real-time systems are difficult problems due to the timing constraints of the tasks involved. Scheduling policies in hard real-time systems need to ensure that tasks will meet their deadlines under all circumstances, even in the presence of faults.*

This work presents techniques to enhance the fault tolerance capability of multiprocessor hard real-time systems, in the presence of transient and permanent faults. As a special case, in the present paper we propose a new method to obtain a high level of fault-tolerance in the CAN bus by incorporating time redundancy and task schedulability tests, which may be used concurrently with processor redundancy and any other hardware redundancy.

1 Introduction

Real-time systems are systems that depend on the result of computation as well on the deadline by which this result is reached. Real-time systems include sensors and actuators; task deadlines are typically derived from the required responsiveness of the sensors and actuators, which are monitored and controlled by the system. Examples of such systems include signal processing, process control, flight control, telecommunication, automotive and life-support medical systems. Hard real-time systems (HRTSs) have stringent timing constraints, and the consequence of missing task deadlines may be catastrophic.

Scheduling and resource allocation in real-time systems are difficult problems due to the timing constraints of the tasks involved. Fault tolerance is an especially vital requirement for HRTS development. Scheduling policies in hard real-time systems need to ensure that tasks will meet their deadlines under all circumstances, even in the presence of transient or permanent faults. The time requirements and fault model depend on a high knowledge of the application and its environment.

This work presents an overview, extension and application of techniques to enhance the fault tolerance capability of multiprocessor hard real-time systems based on the CAN protocol. We introduce extensions to the CAN (Controller Area Network) bus communication protocol and apply a set of techniques to improve its reliability. A well-known problem of the CAN bus is on delivering low priority messages, which may be compromised if the bus is flooded with higher priority messages. In this work, we use task schedulability and time redundancy to optimize fault-tolerance requisites for multiprocessor hard real-time systems. In this scenario, a new technique is proposed to enhance the fault-tolerance capability of the CAN protocol by incorporating time redundancy, which can be used in conjunction with hardware and software redundancy to tolerate faults in hard real-time systems.

This work was developed over the CAN protocol mainly because of it is present in more than 90% of microcontrollers and DSPs (digital signal processors) incorporating real-time protocols. A comparison of real-time fault-tolerant communication protocols is also presented, highlighting the advantages of the proposed method.

This paper is organized in the following form. In Section 2 related work including real-time protocols with fault tolerance requisites are presented. In Section 3 the new approach to improve reliability in the CAN bus is shown. Finally the results are briefly analyzed and some conclusions are presented.

2 Related Work

One of the essential services provided by real-time fault-tolerant distributed architecture is communication of information from one distributed component to another; a communication bus is one of its principal components, and the protocols used for control and communication on the bus are among its principal mechanisms. In truth, these architectures are the safety-critical core of the applications built above them, and the choice of services to provide to those applications, and the mechanisms of their implementation, are issues of major importance in the construction and certification of safety critical embedded systems [12][13]. In a distributed hard real-time system, communication between tasks on different processors must occur in bounded time.

Redundant busses are often used in safety-critical environments to handle device faults. Besides fault tolerance, many applications require real-time guarantees such as deterministic message latency. There are various protocols for such purposes, with different complexities, which are used by the avionics industry, such as Airbus and Boeing, and automobile industry, such BMW and Audi. Some of the more representative real-time communication protocols today are TTP/C (Time-Triggered Protocol) [17], FlexRay [6], CAN (Controller Area Network) [4] and TTCAN (Time-Triggered CAN) [1][3][11].

Some of the busses considered here are primarily *time triggered* which means that all activities involving the bus, and often those involving components attached to the bus, are driven by the passage of time. In *event-triggered* busses, the activities are driven by the occurrence of events. A time-triggered system interacts with the world according to an internal schedule, whereas an event-triggered system responds to stimuli.

The Time Triggered Architecture (TTA) was developed by Hermann Kopetz and colleagues at the Technical University of Vienna [7]. Commercial development of the architecture is being deployed for safety-critical applications in cars and for flight-critical functions in aircrafts. In a Time-Triggered Architecture, the communication system decides autonomously and according to a static schedule when to transmit a message.

FlexRay [6] is a new real-time protocol, not yet released to the public, being developed by a consortium of companies (BMW, Motorola, etc.) aiming to be more flexible than TTP/C. Although used primarily for automotive applications, it is representative of state-of-the-art safety critical real-time protocols. FlexRay introduces some dynamism through the combination of time-triggered and event-triggered operation.

The controller area network (CAN) protocol [4] uses a serial multimaster communication where prioritized messages with up to eight bytes data length can be sent using an arbitration protocol and an error-detection mechanism for a high level of data integrity. The CAN bus is ideal for applications operating in noisy and harsh environments, such as in the automotive and other industrial fields that require reliable communication.

A new development in CAN technology is the TTCAN protocol [1][3][11], a higher-layer protocol above the unchanged standard CAN protocol that synchronizes the communication schedules of all CAN nodes in a network and that provides a global system time, avoiding the transmission collisions commonly found in standard CAN networks. In TTCAN, all the message instances are transmitted only on previously allocated time-slots, just like the TTP/C protocol, without competing with other messages for the bus.

2.1 Rate Monotonic Scheduling applied to the CAN bus

The dynamic scheduling algorithm used by the CAN protocol is virtually identical to scheduling algorithms commonly used in real-time systems to schedule computation on processors [14]. In fact, the analysis of the timing behavior of such systems can be applied almost without change to the problem of determining the worst-case latency of a given message queued for transmission on CAN.

Tindell *et. al.* [15] developed a CAN analysis based on RMA, showing how to find the response time for messages being transmitted in a CAN bus.

As defined by Tindell, the worst-case response time is composed of two delays: the queuing delay and the transmission delay. The queuing delay is the longest time that a message can be queued in a station and be delayed because other higher and lower priority messages are being sent on the bus, which are known as interference and blocking time respectively. The transmission delay is the time taken to actually send the message on the bus.

A model for error handling must also be included, once it is important in a fault tolerant scenario. In a CAN bus, an error detected by either the sender of a message or a receiver station is signaled to the sender station, which must re-transmit that message. The costs of error handling are given as the most probable bound on the overheads due to errors in an interval of duration t, and it includes the cost of re-transmission. An extended analysis can also be found in [15] which includes also remote transmission request messages.

3 Improving Reliability in a CAN bus

A perceived problem with the CAN protocol for use in distributed real-time control applications is the inability to bind the response times of messages. While CAN is very good at transmitting the most urgent data, it is unable to provide guarantees that deadlines are met for less urgent data [7] [8] [9], once the most urgent data may flood the bus avoiding the transmission of the less urgent data.

In a CAN bus, the delivery of low priority messages may be compromised if the bus is flooded with higher priority messages. In this sense, we need to guarantee that an overload in the bus will not occur and a deterministic package response time can be achieved.

3.1 Applying Fault tolerance requisites to a CAN bus

Due to the critical nature of the tasks in hard real-time systems, it is essential that faults be tolerated. Transient faults in real-time systems are generally tolerated using time redundancy, which involves the re-execution of any task running during the occurrence of a transient fault [5].

Ghosh [2] showed a recovery scheme for single and multiple faults that ensures the re-execution of any task after a fault has been detected. Once the dynamic scheduling algorithm used by the CAN protocol is virtually identical to scheduling algorithms commonly used in real-time systems

to schedule tasks, this recovery scheme may be also used to ensure the re-transmission of any message in a CAN bus

The general approach to fault tolerance is to maintain enough slack (backup time) in the schedule so that any message instance can be re-transmitted if a fault occurs during its transmission. If no faults occur, messages are transmitted just following the usual RMS scheme and the slack is not used. If a fault occurs in the transmission process of a message, a recovery scheme is used to re-transmit that message. The ratio of slack S available over an interval of time L is thus constant and can be imagined to be the utilization of a backup message B. If the backup utilization is U_B, then the slack available during an interval L, denoted by B_L, is $B_L = U_B L$.

In order to apply this approach the following conditions must be satisfied:

[Sl]: There should be sufficient slack for every instance of each message to be re-transmitted. That is, the slack between kT_i and $(k + 1) T_i$ should be at least C_i for any value of k and i, what ensures the availability of sufficient slack for a message to be re-transmitted.

[S2]: When any instance of τ_i finishes executing, all the slack available within its period (at least C_i if [Sl] holds) should be available for the re-transmission of τ_i. This slack can be used after a message finishes transmitting to re-transmit that message before its deadline, if a fault is detected.

[S3]: When a message re-transmits, it should not cause any other message transmission to miss its deadline, allowing all tasks to meet their deadlines even when a high priority task needs to re-execute.

If these three conditions are met, then it is possible to re-transmit a faulty message and meet its deadline. However, a recovery scheme must define also how the slack should be used and a very straightforward scheme consists on the faulty message simply being re-transmitted at its own priority.

This scheme is a general approach to distribute slack in the schedule, and it can be applied to any non-fault-tolerant scheduling scheme for preemptive, periodic tasks where the RMS assumptions hold. Any transmission time C_i in the non-fault-tolerant scheme can be split into two parts for the fault-tolerant scheme: a new transmission time $C_i' = C_i(1 - U_B)$ (where U_B is the backup utilization) and a slack equal to $C_i U_B$. To guarantee the re-transmission of a message before its deadline, its *critical instance* is considered, which is defined as the time at which the message's transmission is maximized - it happens when the message starts its transmission process simultaneously with all higher priority messages [10]. The total slack available for any message at its critical instance is equal to the total slack available within a period boundary, which is defined as the beginning of a period.

Ghosh showed also that, by splitting up each transmission time C_i into a new transmission time C_i' and slack, as described above, the utilization of each task τ_i is reduced to $U_i(1- U_B)$, and thus the following general fault tolerance boundary for an RMS ($U_{G\text{-}FT\text{-}RMS}$) is obtained:

$$U_{G\text{-}FT\text{-}RMS} = n(2^{1/n} - 1)(1 - U_B) = U_{LL\text{-}RMS}(1- U_B) \quad (1)$$

The above equation is a general one applicable to an RMS for any value of U_B. If $U_B = \max\{U_i\}$, $i = 1,..., n$, then any message transmission in the system can tolerate a single fault. Any number of faults can be tolerated if [S1] holds.

Multiple faults within two consecutive period boundaries are also guaranteed to be tolerated using the scheme described above. If several backups are provided in the system, and the total backup utilization is U_{BT}, then a general boundary for the message set can be derived by replacing U_B with U_{BT} in $U_{G\text{-}FT\text{-}RMS}$; that is, the new boundary is $U_{LL\text{-}RMS}(1- U_{BT})$.

3.2 Limiting the maximum transfer rate of the CAN bus

Considering the CAN protocol, the absence of a message is identified by the CPUs connected to the bus as a fault. In this case, a failure model is applied, implying in the re-transmission of the message that failed in its transmission or even in the execution of an alternative action such as the reconfiguration of the bus.

The RMS scheme can be applied to the CAN bus since the following premises can be assumed:

- The messages are independent, which means in other words that they are asynchronous to each other.
- The messages will have their priority ascertained by RMA.
- Each message has a maximum transmission time and deterministic period.

The independency of the messages is guaranteed by the fact that we are dealing with periodic control messages. We must also guarantee that [S1], [S2] and [S3] are satisfied, as presented in Section 3.1. The maximum transmission time is, in the worst-case, the worst-case transmission time defined by Tindell *ET. al* in [14], and will help us define the backup slack size and ensure the availability of sufficient slack for a message to be re-transmitted. This slack can be used after a message finishes transmitting to re-transmit that message before its deadline, if a fault is detected. This means that [S1] and [S2] hold for the CAN protocol.

The third condition [S3] may not hold for a CAN bus, once a higher priority message re-transmission may prevent the transmission of a lower priority message, causing the latest to miss its deadline. Thus, to guarantee that a CAN bus can tolerate faults, we define a maximum transmission rate for each message instance; instead of pre-defining a time-driven slot as is the main idea of TTP/C. With this idea in mind, we will not limit a node transmission to its slot, but allow it to transmit at any time if its transmission frequency allows. We will call this extension of the CAN bus as RMCAN, which means *Rate Monotonic CAN*.

From the protocol point of view, the retransmission of a finite number of messages in the case of a transmission failure may not compromise the bus bandwidth and the RMS may be applied deterministically.

This way, any message corruption or further errors indicate that a fault occurred, and the message must be re-transmitted and also meet its deadline. Moreover, if the fault persists, a failure is detected and an alternative process must be executed in another processor to prevent a global failure.

The following example shows how RMCAN can be used to determine whether a message can be re-transmitted before its deadline with guarantees in a CAN bus. Consider a set of 10 processors - N = 10 - sending 5 periodic messages with utilizations U_i = 1% for each message i. Each processor is also limited to a maximum data transmission volume of 10% of the bus bandwidth. Once this limit is reached, the processor that is sending messages stops sending messages and higher layers of the processor that should be receiving the messages will tolerate the fault taking the appropriate action in the context of the specific application.

If we assume that a message fault needs to be tolerated and re-transmitted up to 5 times, then U_B = 5%. Equation (2) gives us a bound of 66% while the sum of utilizations of the task is 50%.

$$100\ (2^{1/100} - 1) * (1 - 0{,}05) = 0{,}66 \qquad (2)$$

Since $\Sigma U_i < U_{G\text{-}FT\text{-}RMS}$, the messages are schedulable.

4 Comparison of Real-Time Fault-Tolerant Communication Protocols – advantages of the proposed method

This comparison does not reiterate the common design decisions, but focuses on the differences between TTP/C, CAN, TTCAN and FlexRay protocols and shows the advantages of the adaptation proposed over the CAN protocol in this work (RMCAN).

From the buses considered just above, only TTP/C is solely time-triggered while the CAN bus is event-triggered. TTCAN and FlexRay combine time-triggered and event-triggered operation aiming to be more flexible than TTP/C and safer than CAN protocol. This time-triggered versus event-triggered decision is a fundamental design choice that influences many aspects of their architectures and mechanisms. The mechanism adopted by each protocol to resolve transmission concurrency between nodes is decisive to indicate if collisions or concurrency occur during runtime.

Analyzing the performance of these protocols, we may see that latency is constant and known at design time for TTP/C, while it may increase with load in a CAN bus. The main problem with the CAN bus is that it cannot prevent an overload of the communication system, which may cause a disastrous result when the delivery of low priority messages is prejudiced by higher priority messages re-transmission. RMCAN solves this problem by limiting the transmission frequency of a node to a maximum value. In this case, the worst-case and latency are precisely known.

In this sense, little is known about the FlexRay protocol, which has not been released yet. All that is known is that FlexRay provides no services to its applications beyond best efforts message delivery. A never give up strategy inside FlexRay leaves the control of the communication system with the application, and latency will be constant and precisely known at design time for the TDMA window.

Resuming, we may say that TTP/C provides an off-line communication design yielding guaranteed latency for all messages in the system, but presents low flexibility once bandwidth is distributed at design time by assigning frames of specific length to each node. In a CAN bus, otherwise, priorities are distributed at design time by assigning unique identifiers and a full control by the application over the bandwidth distribution. The CAN protocol is highly flexible and widely available, although some extensions must be done to guarantee a reliable mechanism to build fault-tolerant safe-critical systems. TTCAN is a compromise and represents the necessary evolution of CAN for dealing with higher loads on the bus. However, synchronizing nodes is not a simple task, and brings a new complexity to the CAN bus. RMCAN was developed to be as efficient as TTCAN without incorporating extra hardware or difficulties. Both protocols can be implemented using a regular CAN microcontroller, although for TTCAN it is also necessary an extra hardware for the time-triggered portion of the protocol. Synchronizing nodes is not a simple task, and brings a new complexity to the CAN bus in TTCAN. Implementing RMCAN is much easier, and does not require any extra hardware. The transmission frequency of each message set can be controlled through software.

FlexRay can be considered the state-of-art in the real-time fault-tolerant communication protocols area, although it has not been released yet. It promises a higher bit rate than TTCAN an increase in flexibility when compared to TTP/C.

The use of the CAN protocol in the development of applications is favored by the high availability of microcontrollers incorporating the bus. Today, the biggest advantages of CAN

compared to other networks are the costs and the price/performance ratio. The enhancements proposed by TTCAN and RMCAN are examples of how CAN problems can be circumvented and its spread presence in the market can be explored.

	TTP/C	CAN	TTCAN	RMCAN	FlexRay
Media access strategy	Time-triggered	Event-triggered	Time and Event-triggered	Event-triggered and transmission frequency directed	Time and Event-triggered
Dynamic bandwidth sharing among nodes	No	Yes	Yes	Yes	Yes
Market presence	< 1%	> 99%	May explore the CAN protocol market presence		Not available
Data Efficiency vs. Latency	Constant and known at design time	Increases with load	Constant	Constant	Constant and known at design time
Response Time	Deterministic	Non-deterministic	Deterministic	Deterministic	Deterministic

Table 1 – Comparison between TTP/C, CAN, TTCAN, RMCAN and FlexRay protocols

5 Conclusions

Tasks in real-time systems must meet their deadlines under all circumstances, even in the presence of transient or permanent faults. This work has shown that time redundancy through scheduling is a powerful tool to deal with faults in real-time systems. The harmonious integration of the available techniques enhance the fault tolerance capability of multiprocessor hard real-time systems.

Relating to real-time communication protocols, the time-triggered and event-triggered approaches find favor in different application areas, and each has strong advocates [13]. The CAN protocol may have a non-deterministic response time for an arbitrary low priority message. Researchers sometimes say that the CAN protocol is more appropriate for soft real-time systems (flexible requirements), while appropriate protocols for hard real-time systems include TTP/C. RMCAN, the extension proposed to the CAN protocol, shows that it is possible to bound the message transmission time, thus making possible its use on HRTSs.

In RMCAN there is a limit on the node transmission rate, making the transmission time deterministic, even for low priority messages. We do not limit a node transmission to its slot, but allow it to transmit at any time if its transmission frequency allows. This way one can guarantee that a message will arrive at its deadline or it will not arrive anymore, in which case a backup action is taken.

One advantage of the CAN protocol over time-triggered protocols is the extensibility aspect. New nodes can be added to the bus, while in the TTP/C protocol, for example, a slot for a new node has to be reserved at design time. Other advantages are the high availability of microcontrollers incorporating the CAN bus, and the price/performance ratio. The enhancements proposed to the CAN protocol show that its reliability can be increased and its spread presence in the market can be further explored.

Another aspect in multiprocessor real-time systems relates to the interdependence of task execution time and message transmission time. The release jitter of a receiver task depends on the

arrival time of a message, which in turn depends on the interference from higher priority messages, which in turn depends on the release jitter of the sender tasks. A future work may be the analysis of the relationship of these times, considering worst-case situations.

6 References

[1] Fuhrer T., Muller B., Dieterle W., Hatwitch F., Hugel R., Weiler H., Walther M., GmbH R. B.; Time Triggered Communication on CAN; Proceedings 7th International CAN Conference; 2000.
[2] Ghosh Sunondo, "Guaranteeing Fault Tolerance Throgh Scheduling in Real-Time Systems", Ph.D. Thesis, University of Pittsburgh 1996.
[3] Hartwich F., Futhrer T., Hugel R., Muller B., GmbH R. B.; Timing in the TTCAN Network; Proceeding 8th Intenational CAN Conference; 2002, Las Vegas.
[4] ISO 11898:1993 Road vehicles -- Interchange of digital information -- Controller area network (CAN) for high-speed communication.
[5] Kopetz H., Kantz H., Grunsteidl G., Puschener P., Reisinger J.; Tolerating Transient Faults in MARS. In Symp. On Fault Tolerant Computing(FTCS-20), pages 466-473. IEEE, 1990.
[6] Kopetz H., A Comparation of TTP/C and FlexRay. Research Report. Institut fur Tevhnische Informatik. Technische Universitat Wien, Austria. 2001.
[7] Kopetz, H., Griinsteidl, G., TTP- A Protocol for Fault-Tolerant Real-Time Systems, IEEE Computer, January 1994, pp. 14-23
[8] Kopetz H.. Communication Protocols for Fault-Tolerant Distributed Real-Time Systems. 1994. Nortic Seminar on Dependable Computing, Technical University of Denmark, Lyngby, Denmark.
[9] Kopetz, H., *"A Solution to an Automotive Control System Benchmark",* Institut fur Technische Informatik, Technische Universitat Wien, research report 4/1994 (April 1994)
[10]Liu C.L., Layland J.W., " Scheduling Algorithms for multiprogramming in a Hard-Real-Time Environment," J.ACM, vol. 20, pp.46-61, 1973.
[11]Muller B., Fuhrer T., Hatwitch F., Hugel R., Weiler H., GmbH R. B.; Fault Tolerant TTCAN Networks; Proceedings 8th International CAN Conference; 2002; Las Vegas.
[12]Rushby J., Bus Architectures for Safety-Critical Embedded Systems. SRI International Computer Science Laboratory. 2001 Menlo Park USA.
[13]Rushby J., A Comparison of Bus Architectures for Safety-Critical Embedded Systems. SRI International Computer Science Laboratory. CSL Technical Report. 2001 Menlo Park USA.
[14]Tindell K., Burns A., Guaranteeing Message Latencies on Control Area Network (CAN). University of York, Department of Computre Science, York, England, 1994.
[15]Tindell K., Burns A., Wellings A.,. Calculating Controller Area Network (CAN) Message Response Times. University of York, Department of Computre Science, York, England, 1994.
[16]Tindell K., Fixed Priority Scheduling of hard real-time systems. Ph. D. Thesis. University of York, Department of Computer Science, York, England. 1994.
[17]TTTech Computertechnik, TTP/C Protocol. Specification of TTP/C Protocol. http://www.tttech.com AG 1999.

Verificação formal de protocolos TDMA quanto a características de tempo real e tolerância a falhas

Alberto Rubens Beckler, Sérgio Vale Aguiar Campos

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
{rubinho, scampos}@dcc.ufmg.br

Abstract *- This paper describes a methodology for modeling TDMA protocols and possible faults that can occur in fault tolerant real time systems using automatic methods of verification. This allows guarantee several qualitative properties and quantitative upper bound time constraints. As case study, presents the modeling for TTP/C protocol, identifying the upper bound for reach agreement for group-membership property in a fault scenario.*

Resumo *- Este trabalho apresenta uma metodologia para representar protocolos TDMA e possíveis falhas que possam ocorrer em sistemas de tempo real tolerantes a falhas utilizando métodos automáticos de verificação. Isto permite garantir diversas propriedades qualitativas e resultados quantitativos como o limite superior de tempo para atingir tais propriedades. Como estudo de caso, apresenta a representação do protocolo TTP/C, determinando o limite de tempo máximo para se atingir a auto-estabilização das visões que cada nodo possui do sistema (Group-membership), possibilitando garantir o tempo máximo de recuperação do sistema, em um cenário de falhas.*

1. Introdução

Recentemente, tem havido uma crescente utilização de protocolos TDMA (*Time division Multiple Access*) em sistemas distribuídos para aplicações de tempo real, como em telefonia celular, por exemplo. Esta estratégia se baseia no escalonamento do canal de comunicação entre os nodos do sistema distribuído, o que torna possível determinar um limite mínimo de qualidade de serviço na comunicação. Cada nodo do sistema possui periodicamente um *slot* TDMA, que é um intervalo de tempo onde lhe é permitido acessar o canal de comunicação e difundir mensagens aos outros nodos do sistema.

Uma outra propriedade importante desta estratégia é permitir a detecção de falhas em algum dos nodos do sistema quando não é possível receber uma mensagem dentro do intervalo de tempo previamente definido, permitindo sua utilização em sistemas tolerante a falhas.

A partir da detecção da falha de um nodo, é possível reconfigurar o sistema distribuído de forma a suprir a falha de um dos nodos. Neste cenário de falha, o sistema permanece em um estado inválido até que a reconfiguração esteja concluída. Esta estratégia de recuperação de falhas, denominada redundância dinâmica [6], pode ser utilizada em sistemas de tempo real desde que o atraso gerado pelo processo de recuperação não viole os limites de tempo (*deadline)* de execução das tarefas do sistema.

Um exemplo é o protocolo TTP/C [5] (*time-triggered protocol - class C*), recentemente desenvolvido por um consórcio de empresas denominado TTTech para aplicação nas indústrias automobilística e de aviação. Este protocolo prove uma infra-estrutura básica para a redistribuição de tarefas entre os nodos restantes do grupo, como a propriedade de *Group Membership*, que garante a mesma visão em cada nodo de todos os nodos corretos em um *cluster*. Entretanto, em um cenário de falha, o protocolo estabelece que o tempo máximo para a

estabilização destra propriedade, ou seja, que todos os nodos detectem a falha de um nodo, é de 2n-1 slots TDMA, onde n é o número de nodos do sistema.

Estabelecer o intervalo de tempo máximo em que o protocolo permanece em um estado inválido e o atraso provocado pelo gerenciamento do sistema de tolerância a falhas é fundamental para a execução correta de um protocolo para sistemas tolerante a falhas em aplicações de tempo real. Como as falhas ocorrem de maneira aleatória e incontrolável, apenas com testes e simulações é praticamente impossível validar protocolos para utilização em situações críticas.

Existem ferramentas automáticas capazes de realizar a verificação formal de diversas propriedades em sistemas distribuídos. Estas ferramentas modelam a execução de um protocolo em fórmulas matemáticas, de maneira precisa e não ambígua. Realizam operações matemáticas sobre estes modelos a fim de determinar se uma determinada propriedade não é violada. Estas ferramentas produzem resultados mais significativos quando utilizadas ainda na fase de concepção do produto, detectando possíveis erros de projeto reduzindo os custos de correção e produção de protótipos.

Para aumentar a utilização destas ferramentas por profissionais de diversas áreas da ciência da computação é necessário o desenvolvimento de metodologias simples, generalizando soluções para a representação de aspectos padrões a uma determinada área, abstraindo de detalhes específicos, a fim de diminuir o tempo e a complexidade do aprendizado de tais ferramentas.

Neste cenário, este trabalho descreve uma metodologia para a representação das características básicas presentes no desenvolvimento de protocolos para sistemas de tempo real tolerante a falhas na ferramenta de verificação formal VERUS [7], como a modelagem e a topologia do *cluster*, as mensagens enviadas por cada um dos nodos, além da representação do tempo e das falhas que possam ocorrer no sistema. Tornando possível garantir diversas propriedades e os limites inferiores e superiores de tempo para atingi-las.

A fim de comprovar sua aplicabilidade, utilizou-se a metodologia proposta para verificar e comprovar o limite de tempo de 2n-1 slots de transmissão TDMA para a auto-estabilização do *Group-Membership* para o protocolo TTP/C, conforme a especificação do protocolo.

2. Trabalhos relacionados

2.1 Propriedades de protocolos para sistemas de tempo real tolerante a falhas.

Para realizar a verificação formal de protocolos é necessário determinar quais propriedades devem ser atendidas a fim de garantir o funcionamento correto do protocolo.

K. Birman [8], cita as condições básicas que devem ser atendidas para que sistemas distribuídos permitam uma operação continuada mesmo na presença de falhas. Dentre estas características destaca-se a propriedade de *Group-Membership* [4], que é a necessidade de gerenciar a formação e coordenação de grupos de nodos corretos, tanto na presença de falhas, quanto durante a recuperação do sistema. Este conhecimento, além de ser essencial por motivos de enumeração e endereçamento dos seus membros, é fundamental, sobretudo para a recuperação de uma falha no sistema.

Sistemas *time-triggered* [5] [11] são sistemas distribuídos onde todo o controle de um protocolo é disparado em instantes pré-determinados do tempo. Existem também sistemas *event-triggered* [11], onde o controle ocorre na presença de eventos, como o recebimento de uma mensagem, por exemplo.

Sistemas time-triggered podem detectar uma falha não apenas pelo recebimento de uma mensagem incorreta, mas também pela ausência de uma mensagem após o intervalo de transmissão TDMA de um nodo. Assim, apesar de uma falha ser um evento do sistema, a detecção de uma falha pelos outros nodos de um cluster é controlada por protocolos *time-triggered*.

Recentemente, tem havido um crescente desenvolvimento de protocolos de redundância ativa para aplicações tolerante a falhas em sistemas de tempo real. Como os protocolos TTP/C [11] e TTCAN [12].

A principal estratégia destes protocolos é controlar o acesso de um nodo ao canal de comunicação, a fim de evitar as colisões entre as mensagens e determinar um tempo limite para a entrega de uma mensagem pelo canal de comunicação.

O protocolo TTP/C utiliza uma estratégia TDMA de acesso ao canal de comunicação. Foi desenvolvido especificamente para a área de tolerância a falhas, inclui diversas características que auxiliam o projeto de aplicações, como sincronização de relógio, mecanismos de confirmação do recebimento de mensagens, detecção de falhas e de gestão de grupos.

A verificação formal do protocolo TTCAN quanto à auto-estabilização da propriedade de *group-membership* também foi realizada e encontra-se na versão completa deste artigo.

3. Verificação formal

Ferramentas de verificação formal representam o funcionamento de um circuito ou a execução de um algoritmo em modelos matemáticos, de maneira precisa e não ambígua, realizam operações matemáticas sobre estes modelos a fim de verificar se uma determinada propriedade é satisfeita ou não.

Estas ferramentas são baseadas principalmente em técnicas de *simbolic model checking* [2]. Representando a execução de um algoritmo, seqüencial ou distribuído, em um grafo onde cada estado corresponde a uma possível configuração de valor para as variáveis do modelo, e a transição entre os nodos correspondendo a uma mudança de valor para uma determinada variável.

Por se basear na enumeração de todos os possíveis estados e transições do modelo, estas ferramentas necessitam de uma representação eficiente do grafo que representa o algoritmo. O estado da arte destas ferramentas utiliza grafos de decisão binários - BDDs (*Binary Decision Diagram)* [2] para a representação do modelo. Um BDD é uma representação canônica de fórmulas booleanas. Além disso, possuem um arcabouço de operações sobre estas estruturas a fim de possibilitar uma representação eficiente das transições entre os estados, como operações lógicas booleanas e o caminhamento entre os nodos do grafo[2].

Estas operações permitem realizar uma busca exaustiva aos nodos do grafo e garantir se uma determinada propriedade é satisfeita ou não, garantido a correção de um algoritmo ou, em caso de erro, apresentar o caminho percorrido que provocou a violação de uma propriedade.

Aplicações críticas exigem técnicas de tolerância a falhas tanto em hardware quanto em software, baseadas em redundância, o que aumenta sua complexidade exigindo a utilização de ferramentas para auxiliar na análise do processo. Além disso, gerenciar os recursos redundantes provoca um atraso na execução do sistema aumentando a complexidade da análise dos limites de tempo em sistemas críticos de tempo real.

Para realizar a verificação formal de sistemas de tempo real é necessária a utilização de ferramentas que possuam construções para a representação do tempo. Existem ferramentas desenvolvidas com este propósito como a ferramenta de verificação formal VERUS [7] utilizada neste trabalho. Esta ferramenta possui uma representação discreta do tempo. Esta simplificação é necessária para permitir a modelagem e a verificação de sistemas complexos, e não provoca prejuízos quanto a qualidade da modelagem, visto que em sistemas time-triggered, é necessário representar de forma discreta os intervalos de tempo de controle do protocolo.

4. Metodologia para verificação de protocolos para sistemas de tempo real tolerante a falhas.

Garantir que uma propriedade é satisfeita durante toda a execução de um protocolo, mesmo em caminhos pouco prováveis de execução, além de estabelecer os limites superiores de tempo para atingi-las é fundamental para um projetista durante a elaboração de um protocolo.

Em sistemas de tempo real, estes parâmetros são fundamentais para estabelecer propriedades, como a velocidade de comunicação e número máximo de nodos de um cluster. Entretanto, controlar a simulação de todos os caminhos de execução de um protocolo e todos os instantes possíveis para a ocorrência de falhas pode levar a um enorme número de possibilidades. A utilização de técnicas de *simbolic model checking* possibilita a representação eficiente da estrutura de dados utilizada para modelar todas estas possibilidades.

Este trabalho propõe a utilização da ferramenta VERUS para representar as principais características de um cluster telerante a falhas, o protocolo TDMA executado, a representação da passagem do tempo, necessária para garantir propriedades temporais e as possíveis falhas que possam ocorrer no sistema.

4.1 Modelagem do protocolo

Para representar o algoritmo, a ferramenta VERUS possui uma linguagem muito semelhante a C. Permite a execução em paralelo de diversos processos e a representação discreta do tempo. A tabela 1 representa a modelagem de um protocolo TDMA para a linguagem da ferramenta VERUS.

Tabela 1. Modelagem do cluster tolerante a falhas em Verus

CLUSTER	VERUS
Nodo	Processo
Registradores	Variáveis locais
Slot de tempo TDMA	wait (1);
Mensagens	Variáveis globais
Propriedades a serem verificadas	Lógica CTL
Falhas	Modelo de falhas

Como a verificação é realizada sobre um modelo, e não sobre a implementação do protocolo, a facilidade e a qualidade da modelagem são importantes tanto para ter certeza que o modelo implementa realmente o protocolo, quanto para a análise dos resultados da verificação do modelo, mapeando as informações obtidas da ferramenta para o protocolo em questão. Como a ferramenta VERUS utiliza uma linguagem muito semelhante a C, a tarefa de modelagem do protocolo torna-se bastante simplificada. A modelagem das construções básicas para a representação do sistema distribuído está descrita de forma a fornecer ao projetista de um protocolo uma metodologia simples de representação de seu projeto.

Cada nodo de um sistema distribuído é representado como um processo em VERUS e seus registradores como variáveis locais, inteiras e booleanas.

Os processos trocam informações (mensagens) entre si através da atribuição de valores a variáveis globais. É importante ressaltar a necessidade de representar quando uma determinada mensagem não é mais válida cancelando o valor de uma variável após seu recebimento.

VERUS possui o comando wait(1) que representa a passagem de uma unidade de tempo. Os processos executam em modo de passo, ou seja, os trechos de código de cada processo compreendidos entre comandos wait(1) são executados simultaneamente de maneira atômica. Neste ponto de sincronização, os processos tomam conhecimento de alterações em variáveis globais do sistema. Assim, o trecho do algoritmo que deve ser executado em um intervalo TDMA do protocolo deve ser representado entre dois comandos wait(1). Desta forma, é possível determinar o número de slots e, por conseqüência, o tempo necessário para atingir um determinado estado do sistema.

A ferramenta VERUS representa toda a execução do algoritmo em fórmulas booleanas, utilizando para isto um grafo de decisão binária (BDD). Faz uma busca exaustiva a este grafo para verificar se uma determinada propriedade, descrita em lógica CTL [1], é satisfeita ou não.

É possível determinar também o intervalo de tempo máximo entre dois estados do modelo, utilizando a propriedade MAX (p, q), que determina o número máximo de transições entre o momento em que uma propriedade p se torne verdadeira até que q também seja.

Para representar todas as possíveis execuções de um algoritmo, uma atribuição a uma variável pode ser feita a partir do comando select {0, 1, 2 ...}. Desta forma, é gerado um caminho no grafo para cada possível valor atribuído.

VERUS fornece ainda o caminho que o algoritmo percorreu até atingir uma propriedade ou uma violação, o que favorece o entendimento do algoritmo e sua depuração.

Para aumentar a eficiência da ferramenta e, por conseqüência, o tamanho dos modelos que ela é capaz de verificar é necessário realizar algumas otimizações. Isto ocorre porque o tamanho da estrutura de dados utilizada para representar o modelo é dependente de fatores como a quantidade de bits necessários para representar um inteiro e da ordem em que as variáveis são analisadas.

Uma variável inteira é representada como um vetor de variáveis booleanas, portanto, para reduzir o tamanho do modelo é muito importante utilizar o menor número de bits necessários para representar o maior inteiro do modelo, reduzindo assim o número de variáveis analisadas.

O tamanho de um BDD sofre enormes alterações devido a ordem em que as variáveis são analisadas pela ferramenta. Infelizmente, determinar a ordenação ótima para um determinado modelo é um problema NP-Completo [2]. Entretanto, existem técnicas para aproximar o modelo de uma ordenação ótima.

A primeira medida a ser implementada é analisar inicialmente as variáveis que estão mais relacionadas, por exemplo, analisar as variáveis que representam uma mensagem das variáveis do processo que recebe, ou envia estas mensagens. Após esta etapa é necessário utilizar sobre esta ordenação o recurso de reordenação automática presente na ferramenta.

4.2 Modelo de falhas

Faz parte da especificação de um protocolo para sistemas tolerantes a falhas a descrição de todas as falhas que o sistema deverá suportar. A verificação formal do protocolo deve ser direcionada a garantir que as falhas propostas neste modelo são efetivamente suportadas pelos recursos redundantes, inclusive que os *deadlines* das tarefas não serão violados. É uma tarefa relativamente simples descrever o modelo de falhas de um protocolo em VERUS. A geração e o controle de todas as possíveis falhas em um sistema são realizados a partir da declaração de variáveis de controle e do comando *select* que gera todas as possibilidades necessárias para uma atribuição a estas variáveis.

```
nodofalho = select {0, 1, 2, 3, 4};
instante  = select {1, 2, 3, 4};

if ( instante == 2 && nodofalho == 1)
      Aok = false;
wait(1);
```

Fig 1. Representação de uma falha de omissão em VERUS.

Como a falha em um dispositivo pode ocorrer a qualquer instante do tempo, é necessário representar todos os possíveis instantes para a presença de uma falha. O controle de uma falha de omissão [6], por exemplo, é gerada atribuindo false à variável local que representa a visão de um processo sobre seu próprio funcionamento. Cada passo do algoritmo é executado apenas se seu estado for correto. A figura 1 apresenta um trecho de código em VERUS que representa a geração e o controle de uma falha de omissão.

A variável *nodofalho* controla qual dos nodos de um *cluster* irá falhar. A variável *instante* determina em qual intervalo de transmissão TDMA ocorrerá a falha, e a variável *Aok* representa a visão de um nodo sobre seu funcionamento. Desta forma, atribuições às variáveis: *instante* e *nodofalho*, através do comando select, permite a representação de um possível caminho de execução no modelo. O exemplo da figura 1 representa uma falha de omissão para o nodo A (nodofalho = 1) no intervalo de transmissão do nodo B (instante = 2). A ferramenta gera

todas as possíveis combinações para as variáveis e verifica se as propriedades são satisfeitas em todas as suas execuções.

Para efeito de modelagem de sisemas *time-triggered*, falhas temporais são interpretadas da mesma forma que falhas de omissão. Entretanto, sistemas TDMA devem possuir mecanismos que evitem o envio de mensagens fora do intervalo de transmissão do nodo (*bus guardians*).

O controle sobre a duração de uma falha pode ser realizado através de atribuições sucessivas a uma variável, permitindo a modelagem de falhas transientes.

5. Estudo de caso

5.1 Verificação da propriedade de *Group Membership* para o protocolo TTP/C

5.1.1 Descrição do protocolo

O TTP/C [5] (time-triggered protocol class C) é um protocolo usado para interconexão de micro-controladores através de uma estratégia TDMA de acesso ao canal de comunicação, o C indica que o protocolo atende às características da indústria automobilística. Encontra-se em fase de especificação por um consórcio de empresas chamado TTTech.

Este protocolo conecta diversas centralinas eletrônicas no interior de um veículo, estas centralinas controlam diversas tarefas em um automóvel, como: suspensão eletrônica, direção eletrônica, freios ABS, Air Bag, computador de bordo, injeção eletrônica, vidros elétricos etc.

Como as tarefas executadas são críticas, uma determinada falha de um controlador pode levar a conseqüências terríveis. O principal objetivo do protocolo é fornecer um meio de utilizar os recursos de outras centralinas para realizar as tarefas de um controlador com defeito, tornando o sistema tolerante a falhas. Para isto, deve haver um determinado limite superior de tempo para que todos os controladores corretos detectem a falha, distribua as tarefas e atue sobre o dispositivo em questão, antes que seja impossível evitar um acidente.

Cada controlador correto é denominado um nodo em um cluster. As mensagens são enviadas em broadcast aos outros nodos do cluster. O protocolo tolera apenas uma única falha a cada vez no sistema. Garante um tempo máximo para a detecção de uma falha e que todos os nodos corretos formarão um novo cluster.

5.1.2 Implementação do Group Membership

Para manter uma visão estável dos membros corretos de um cluster o TTP/C utiliza um mecanismo de confirmação implícita de mensagens. Cada nodo de um cluster (digamos A) envia em seu slot de transmissão sua visão do grupo. Seu sucessor (digamos B) é responsável pela confirmação. Se o nodo A estiver na visão de B, então sua mensagem foi enviada corretamente. Senão, algum erro ocorreu; ou A não enviou corretamente seu broadcast, ou B está com problemas de recepção. Para solucionar este impasse, o próximo nodo (digamos C) ao enviar sua visão em broadcast informa qual problema foi detectado. Se A estiver na visão de C, então o problema estava em B, senão, o problema estava em A.

5.1.3 Aplicação da metodologia

A metodologia apresentada foi utilizada para modelar e representar o mecanismo de confirmação implícita do protocolo TTP/C com 4 nodos. A aplicação da metodologia garante apenas as propriedades para esta instância de implementação do protocolo. Entretanto, a partir do pior caso para a ocorrência de uma falha, é possível generalizar os resultados para um número arbitrário de nodos. Na literatura, existem outras modelagens que utilizam esta estratégia como a verificação do protocolo PCI [7]. O código completo da modelagem pode ser obtido sob demanda.

5.1.4 Análise dos resultados

Para facilitar a análise dos resultados foi desenvolvida uma ferramenta responsável capaz de extrair os resultados dos contra-exemplos fornecidos pela ferramenta em formato texto para uma planilha eletrônica, melhorando sua visualização e entendimento. A partir desta visualização é possível, por exemplo, fazer uma análise do pior caso de execução de uma instancia do problema e, se possível, extrair resultados genéricos para um número qualquer de nodos que executam o algoritmo.

A figura 2 apresenta um exemplo de saída da ferramenta para o pior caso de falha para o nodo B, descrito pela propriedade MAX (!pB.Bok, !pA.Bok && !pC.Bok && !pD.Bok). Responsável por determinar o intervalo máximo de tempo desde que haja uma falha no nodo B até que todos os outros nodos de um cluster atualizem sua visão do grupo detectando a falha.

Cada estado é representado pelos valores de todas as variáveis em cada slot TDMA. As variáveis apresentadas no exemplo descrevem as visões de cada nodo sobre o grupo. É importante ressaltar que estas visões permanecem diferentes desde a ocorrência de uma falha até que todos os nodos cheguem a um consenso.

A ferramenta é capaz de verificar não somente propriedades qualitativas, como a diferença entre as visões de cada nodo sobre o grupo, mas também propriedades quantitativas como o tempo necessário para se atingir a auto-estabilização da propriedade de *group-membership*.

		VISAO DO NODO A				VISAO DO NODO B				VISAO DO NODO C				VISAO DO NODO D			
SLOT	WC	pA.Aok	pA.Bok	pA.Cok	pA.Dok	pB.Aok	pB.Bok	pB.Cok	pB.Dok	pC.Aok	pC.Bok	pC.Cok	pD.Dok	pD.Aok	pD.Bok	pD.Cok	pD.Dok
	1	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True
A	2	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True
B	3	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True
C	4	True	True	True	True	True	False	True	True	True	True	True	True	True	True	True	True
D	5	True	True	True	True	True	False	True	True	True	True	True	True	True	True	True	True
A	2	True	True	True	True	True	False	True	True	True	True	True	True	True	True	True	True
B	3	True	True	True	True	True	False	True	True	True	True	True	True	True	True	True	True
C	4	True	True	True	True	True	False	True	True	True	False	True	True	True	True	True	True
D	5	True	True	True	True	True	False	True	True	True	False	True	True	True	False	True	True
A	2	True	False	True	True	True	False	True	True	True	False	True	True	True	False	True	True

Fig 2. Contra-exemplo para a execução do protocolo no pior cenário de ocorrência de uma falha.

Foram geradas falhas em cada um dos processos em todos os slot de tempo, conforme o exemplo da figura 1. Os resultados mostram que o tempo máximo para a detecção de uma falha por todos os outros nodos é de 7 slots de tempo.

A generalização dos resultados obtidos para um número arbitrário de nodos pode ser obtida a partir da análise da saída da ferramenta. No contra-exemplo anterior, nota-se que o pior cenário ocorre quando o nodo B falha logo no slot de transmissão do nodo C. Assim, os outros nodos só podem perceber que existe uma falha em B no próximo ciclo, quando B deixar de enviar seu sinal de vida ao grupo, gastando n slots de tempo. Segundo o protocolo TTP/C cada nodo checa sua caixa de mensagens e atualiza sua visão do grupo imediatamente antes de iniciar seu slot de transmissão, assim, são necessários n-1 slots até que todos os nodos atualizem em sua visão a falha no nodo B, iniciando o protocolo de recuperação de falha e redistribuição de tarefas dependente de aplicação cujo atraso provocado pela troca de mensagens também pode ser verificado pela metodologia proposta neste trabalho.

6. Conclusões

Este trabalho apresenta uma metodologia para representar protocolos TDMA utilizando a ferramenta de verificação formal VERUS, tornando possível verificar diversas propriedades, como características de tempo real e tolerância a falhas. Uma determinada instância do algoritmo é representada através de uma linguagem simples e muito semelhante a sua implementação real, ao invés de utilizar outras estratégias de verificação que criam modelos abstratos para a representação das propriedades exigindo uma grande experiência do usuário em

verificação formal e cujo modelo se torna, apesar de correto, muito diferente de sua implementação.

A metodologia apresentada foi utilizada para verificar diversos protocolos para sitemas de tempo real tolerantes a falhas, como protocolo TTCAN e um novo protocolo para interconexão de centralinas eletrônicas em automóveis, TTP/C, descrito neste trabalho, garantindo que o tempo máximo para a detecção de uma falha de omissão pelo por todos os nodos que executam o protocolo é de 2n - 1 slots de tempo, validando sua especificação.

A passagem do tempo é representada de forma discreta pela ferramenta VERUS. Esta simplificação tem como objetivo a utilização da ferramenta na verificação de sistemas complexos. Entretanto, a representação discreta do tempo não prejudica a qualidade da modelagem sendo possível determinar a quantidade de slots TDMA necessárias para se atingir uma determinada propriedade.

É importante ressaltar que, além de garantir se o sistema funciona ou não, é possível fornecer um resultado quantitativo sobre a execução do protocolo. Garantindo os limites máximos de tempo gastos para o protocolo atingir diversas propriedades. Desta forma, um projetista pode validar ou descartar uma determinada estratégia de tolerância a falhas ainda durante a fase de projeto. Para o caso do TTP/C por exemplo, pode ser necessário diminuir o intervalo de tempo TDMA para que o protocolo atenda as necessidades de uma determinada aplicação, entretanto, será necessário uma rede de comunicação mais rápida e uma maior performance dos nodos para que sejam capazes de realizar suas tarefas dentro de um intervalo menor de tempo.

7. Referências

[1] CLARKE, E. EMERSON, E. Design and synthesis of syncronization skeletons using branching time temporal logic. **Workshop on logics of programs**, v.131, of Lectures Notes in Computer Science, p.52-71. 1981.

[2] BRYANT, R. E. Graph-based algorithms for boolean function manipulation. **IEEE Transactions on Computers**, v.C-35, n.8, 1986.

[3] JOHNSON, B. **Design and Analysis of Fault-Tolerant Digital Systems**. 1.ed. Addison-Wesley series in electrical and computer engineering, 1989. 584p.

[4] CRISTIAN, F. Reaching Agreement on Processor Group Membership in Synchronous Distributed System. **Distributed Computing**, v.4, p.175-187, 1991.

[5] KOPETZ, H. GRUNSTEIDL, G., TTP - A protocol for fault-tolerant real time systems, **IEEE Computer**, v.27#1, p.14-23. Jan 1994.

[6] JALOTE, P., **Fault tolerance in distributed Systems**. Prentice-Hall, 1994. 432p.

[7] CAMPOS, S. et al. Verifying the performance of the PCI local bus using symbolic techniques. **International Conference on Computer Design**, 1995.

[8] BIRMAN, K. Buildin secure and reliable network applications. **Manning Publications Co**, 1996.

[9] CAMPOS, S. CLARK, E. MINEA, M. The Verus tool: a quantitative approach to the formal verification of real-time systems. **Conference on Computer Aided Verification**, 1997.

[10] PASCOE, J. LOADER, R. SUNDERAM, V. Working towards the agreement problem protocol verification environment. **Communications Process Architectures**, 2001.

[11] TTP/C Specification 1.0, Disponível em:<www.tttech.com>, Acesso em 26 mar. 2003.

[12] ISO.Road Vehicles Controller Area Network (CAN) Part 4: TimeTriggered Comunication. Standard ISO/CD 11898-4, International Organization for Standardization, 2001

Proposta de uma abordagem para a verificação formal de Sistemas Distribuídos Baseados em Objetos*

Osmar Marchi dos Santos†, Fernando Luís Dotti

Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul
CEP 90619-900 - Porto Alegre, RS

{osantos, fldotti}@inf.pucrs.br

Resumo. *Este artigo propõe uma abordagem para a verificação formal, através da técnica de verificação de modelos, de Sistemas Distribuídos Baseados em Objetos (SDBOs). A linguagem de descrição utilizada para modelar SDBOs é uma forma restrita de Gramática de Grafos, um formalismo gráfico, declarativo, e que suporta paralelismo implícito. Esta linguagem, chamada Gramática de Grafos Baseada em Objetos (GGBO), é brevemente apresentada. Os modelos descritos na GGBO são mapeados para a linguagem de descrição PROMELA (PROtocol/PROcess MEta LAnguage), usada pelo verificador de modelos SPIN (Simple Promela INterpreter). Este mapeamento é descrito. A partir disso, propriedades sobre modelos descritos na GGBO podem ser especificadas e verificadas através do SPIN. Um exemplo de descrição na GGBO é apresentado e mapeado.*

1. Introdução

O constante crescimento na área de computação é refletido tanto em sistemas de *hardware* (novas tecnologias são lançadas no mercado) como em sistemas de *software* (novos paradigmas de aplicação tal como, por exemplo, mobilidade de código são inseridos). Estas novas funcionalidades quando incorporadas aos sistemas muitas vezes apresentam erros. Erro é uma parte do estado do sistema que pode levar o próprio sistema a um possível defeito. Defeito é um evento que ocorre quando a entrega de um serviço desvia do serviço correto [Avizienis et al., 2001]. Porém, sistemas livres de erros são difíceis de alcançar, senão impossíveis. Caso um sistema complexo, como por exemplo, de controle de aviões, apresente erros não antes verificados e tratados pessoas podem vir a morrer. Este é apenas um exemplo, outros tipos de catástrofes como perda de tempo ou dinheiro podem vir a ocorrer [Clarke et al., 1996].

Como pode ser visto, existe uma forte demanda pela criação de sistemas confiáveis. Em específico, a tarefa de desenvolver sistemas distribuídos é considerada complexa pois o sistema pode apresentar inúmeras configurações possíveis. A partir disto, argumenta-se pela necessidade de um método que permita garantir, ainda na fase de desenvolvimento, que um sistema distribuído mantém certas propriedades desejadas, tornando-o mais confiável.

*Este trabalho é parcialmente financiado pela FAPERGS e pelo CNPq.
†Este autor é parcialmente financiado pelo CNPq.

Um dos principais objetivos do projeto ForMOS (Métodos Formais para Código Móvel em Sistemas Abertos[1]) é o desenvolvimento de um *framework* onde diferentes métodos e ferramentas coexistem para facilitar a construção de sistemas distribuídos: simulação, geração de código e, de maior ênfase neste artigo, a verificação formal. De forma mais concreta, foi desenvolvida [Dotti and Ribeiro, 2000] uma linguagem de especificação formal voltada para sistemas distribuídos assíncronos. Esta linguagem, chamada Gramática de Grafos Baseada em Objetos (GGBO), é utilizada como formalismo integrador entre os diferentes métodos e ferramentas desenvolvidos no projeto. Atualmente o *framework* funciona da seguinte forma: o desenvolvedor define o seu modelo usando a GGBO e pode simular o comportamento deste modelo [Copstein et al., 2000]. Assim que o desenvolvedor estiver satisfeito com o comportamento do modelo ele pode gerar código para execução em um ambiente real [Duarte, 2001]. Além disto, em [Rödel, 2003] foi definida uma forma de inserir certos comportamentos falhos (e.g. modelo de falha *crash*) em uma descrição na GGBO existente. Nesta abordagem um modelo M_1 sem falhas é transformado em um modelo M_2 que apresenta o comportamento falho selecionado. A partir deste modelo M_2 o desenvolvedor pode raciocinar (atualmente via simulação) sobre o seu modelo M_1 na presença de certas falhas, possivelmente incluindo mecanismos de detecção e tolerância a falhas no modelo M_1. Uma vez que o desenvolvedor termina de analisar o comportamento do modelo M_2, i.e. o modelo M_1 com a presença de certas falhas, ele pode gerar, usando o modelo M_1 (o modelo que não contém a adição de comportamento falho), código para execução em um ambiente real.

Neste artigo é apresentada uma proposta para a verificação formal, baseada na técnica de verificação de modelos, de Sistemas Distribuídos Baseados em Objetos (SDBOs). A abordagem usada para a verificação formal consiste em mapear descrições na GGBO para a linguagem de descrição PROMELA (*PROtocol/PROcess MEta LAnguage*), utilizada pelo verificador de modelos SPIN (*Simple Promela INterpreter*). A partir disto, as propriedades desejadas do sistema podem ser especificadas em lógica temporal e verificadas automaticamente. Apesar de se tratar de uma proposta e estar com vários pontos em desenvolvimento espera-se, futuramente, se obter uma forma de integrar esta abordagem no *framework* apresentado acima. Assim o desenvolvedor pode utilizar, além da simulação, a verificação formal para raciocinar sobre o seu modelo. Em especial, a integração deste trabalho com a abordagem definida em [Rödel, 2003] facilitaria bastante a construção de sistemas distribuídos considerando certos tipos de falhas.

Na Seção 2. são comentados os trabalhos relacionados. Já na Seção 3. são apresentadas noções básicas sobre a técnica de verificação de modelos e o verificador de modelos SPIN. Na Seção 4. é apresentado o formalismo GGBO e a modelagem de um algoritmo de eleição em anel, ilustrando o uso da GGBO. A proposta deste artigo é vista na Seção 5., onde o exemplo definido é mapeado para a linguagem de descrição PROMELA. Por fim, na Seção 6. são colocadas as conclusões e definidos os principais trabalhos futuros.

2. Trabalhos Relacionados

Encontra-se na literatura um conjunto de trabalhos recentes voltados à verificação de sistemas distribuídos baseados/orientados em/a objetos. O trabalho proposto em

[1]Suporte FAPERGS e CNPq.

[Leue and Holzmann, 1999] tem como objetivo definir uma linguagem de descrição visual e orientada a objetos que possa ser mapeada para o verificador de modelos SPIN. Já em [Cho et al., 1999] a linguagem de descrição PROMELA é extendida considerando o modelo de concorrência *actors*.

Em [Lilius and Paltor, 1999] é proposta uma ferramenta que tenta disponibilizar a verificação automática de sistemas descritos na linguagem de modelagem UML (*Unified Modelling Language*). Esta abordagem consiste em mapear as descrições na UML para a linguagem de descrição PROMELA. Outro trabalho encontrado na literatura [Winter and Duke, 2002] consiste em integrar a linguagem de especificação formal Object-Z com ASM (*Abstract State Machine*), criando uma notação chamada OZ-ASM. Esta linguagem passa por uma série de conversões, sendo possível verificá-la no verificador de modelos SMV (*Symbolic Model Verifier*).

A GGBO é um formalismo com alto nível de abstração, permitindo a modelagem de atributos básicos, assim como de tipos abstratos de dados. Além disso, a GGBO apresenta as mesmas abstrações de encapsulamento e troca de mensagens dos trabalhos acima apresentados. Ainda, conforme já mencionado, a partir de um sistema descrito na GGBO pode-se aplicar técnicas para simulação e/ou geração de código para execução em um ambiente real, além da abordagem de verificação formal, inicialmente proposta neste artigo.

3. Verificação de Modelos

A verificação de modelos consiste em uma técnica automática, sendo empregada para a verificação de sistemas reativos com estados finitos, tais como projetos de protocolos de comunicação [Clarke et al., 1999]. Na maioria das vezes as propriedades a serem verificadas são expressas através de lógica temporal, um tipo de lógica modal que possibilita descrever certas ocorrências de eventos sobre o tempo.

Os verificadores de modelos trabalham sobre um modelo definido através de uma linguagem de descrição (que deve apresentar uma semântica formal) e certas propriedades especificadas em lógica temporal [Manna and Pnueli, 1992]. As propriedades são verificadas sobre o modelo e, ao final desta verificação, é informado ao usuário se a propriedade é válida (verdadeira), ou não (falsa). Caso não seja válida, o verificador de modelos fornece um contra-exemplo da propriedade que consiste na sequência de execução do modelo até o local onde a propriedade é dada como falsa.

3.1. Verificador de Modelos SPIN

O ambiente SPIN [Holzmann, 1997] possibilita a verificação de modelos descritos utilizando a linguagem de descrição PROMELA. Para a especificação de propriedades o SPIN utiliza a lógica de tempo linear (LTL - *Linear Temporal Logic*).

A linguagem de descrição PROMELA apresenta uma sintaxe *C-like*, com características da linguagem de especificação formal CSP (*Communicating Sequential Processes*). Descrições em PROMELA consistem de processos, que podem trocar informações através de canais de mensagem e/ou variáveis globais. O não-determinismo em PROMELA é modelado nas estruturas de repetição/condição. Além disto é possível definir sequências atômicas na linguagem, ou seja, a execução de um conjunto de declarações em um único passo.

4. Gramática de Grafos Baseada em Objetos

Uma Gramática de Grafos [Ehrig, 1979] é composta por um Grafo de Tipos (representa os tipos dos vértices e arestas permitidas na descrição), um Grafo Inicial (representa o estado inicial da descrição), e um conjunto de Regras (descrevem as possíveis mudanças de estado que podem ocorrer numa descrição).

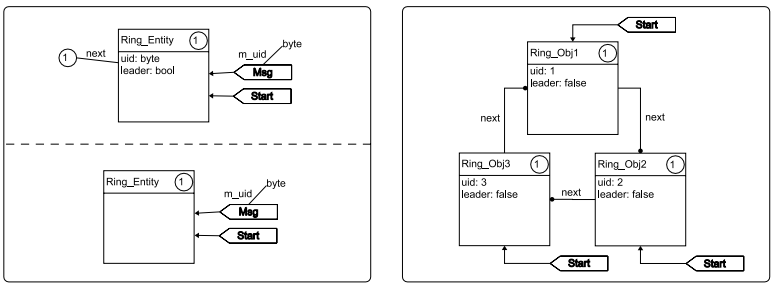
Em [Dotti and Ribeiro, 2000] é proposta a GGBO, uma restrição de Gramática de Grafos para descrever SBOs (Sistemas Baseados em Objetos). Na GGBO uma descrição consiste de objetos que possuem um estado interno e se comunicam através da troca de mensagens. Um objeto é definido segundo um Grafo de Tipos do objeto. O comportamento do objeto corresponde às reações executadas por ele (Regras do objeto) ao receber uma mensagem. Estas reações podem vir a mudar o estado interno do objeto e/ou causar o envio de mensagens para outros objetos e/ou para si mesmo. A instanciação dos objetos que definem uma descrição ocorrem no Grafo Inicial.

Algumas características importantes da GGBO são: apenas uma única mensagem é consumida na aplicação de uma Regra; o não-determinismo é modelado através da escolha implícita de uma Regra, i.e. quando mais de uma Regra se aplica a mesma mensagem, uma destas Regras é escolhida não-deterministicamente para executar; mais de uma Regra pode ser aplicada em paralelo quando as mensagens que as disparam estão presentes no atual estado do sistema (o Grafo) e as Regras não são conflitantes[2].

4.1. Algoritmo de eleição em anel

Nesta Seção é modelado o algoritmo de eleição em anel definido em [Lynch, 1996]. Neste algoritmo os objetos (do tipo *Ring_Entity*, Figura 1 (a)) que compõem a descrição são compostos por três atributos: *next* (referência[3] para o próximo nodo no anel), *uid* (número de identificação do objeto), e *leader* (indica se o objeto é o líder ou não).

O Grafo Inicial para esta descrição é apresentado na Figura 1 (b), onde são instanciados três objetos do tipo *Ring_Entity* e estes iniciam o seu funcionamento devido a recepção das mensagens *Start* definidas. Ao final da execução deste cenário, o objeto *Ring_Obj3* torna-se o líder pois apresenta o maior número de identificação. Devido a restrições de espaço, neste artigo não são definidas e verificadas propriedades sobre este exemplo.



(a) (b)

Figura 1: Grafo de Tipos *Ring_Entity* (a). Grafo Inicial (b).

[2]Neste caso, uma Regra R_1 é conflitante com uma Regra R_2 se consomem (apagam) mesmos ítens.
[3]Por uma questão de sintaxe as referências na GGBO não são definidas dentro do objeto.

As Regras que definem o comportamento dos objetos do tipo *Ring_Entity* são definidos na Figura 2. Como pode ser visto (Figura 2), um objeto do tipo *Ring_Entity* ao receber uma mensagem *Start* (Regra *RuleStart*) inicia o seu funcionamento, enviando uma mensagem *Msg* (carregando o seu número de identificação) ao seu vizinho. Quando um objeto do tipo *Ring_Entity* recebe uma mensagem *Msg* ele pode: reenviar a mensagem ao seu vizinho (Regra *RuleMsg_1*, se o número de identificação da mensagem for maior que o seu), não fazer nada (Regra *RuleMsg_3*, se o número de identificação da mensagem for menor que o seu), ou se tornar líder (Regra *RuleMsg_2*, se o número de identificação da mensagem for igual ao seu).

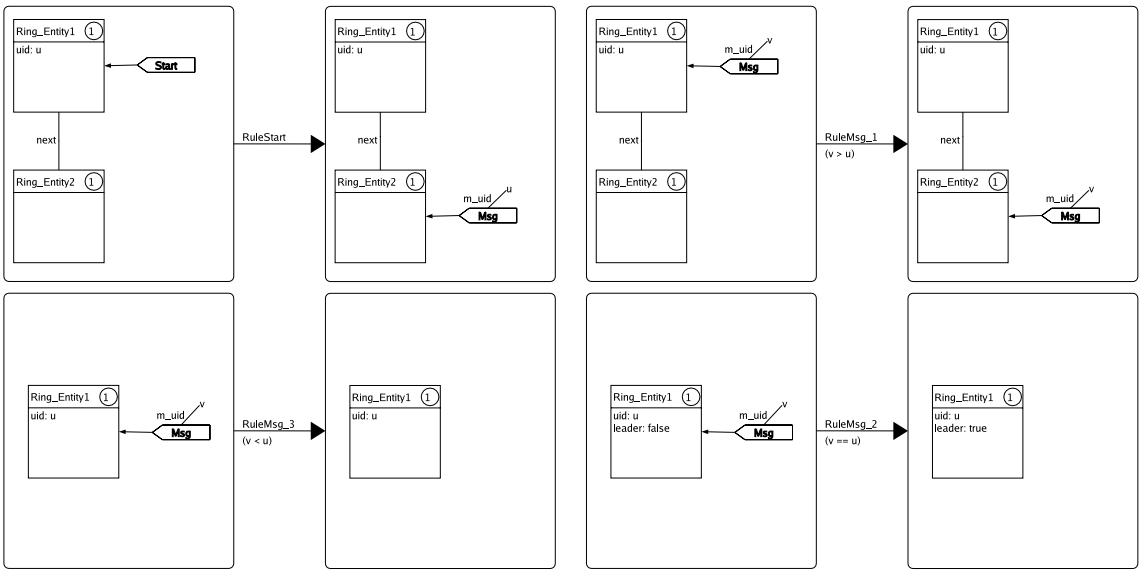


Figura 2: Regras para o tipo de objeto *Ring_Entity*.

5. Proposta para a verificação formal de Sistemas Distribuídos Baseados em Objetos

Esta Seção tem por objetivo apresentar o mapeamento de descrições na GGBO para a linguagem de descrição PROMELA. O exemplo de algoritmo de eleição definido anteriormente é usado para ilustrar este mapeamento.

Na Figura 3 é apresentado parte do tipo de objeto *Ring_Entity* mapeado, assim como parte do Grafo Inicial do exemplo definido. O código visto na Figura 3 contém inúmeros comentários (“/* ... */”), tornando mais fácil a compreensão do mapeamento realizado.

Os nomes de mensagens que compõem a descrição são definidas na Linha 3, estes nomes são usados na escolha de Regras a serem aplicadas. Um tipo de objeto na GGBO é mapeado para um processo em PROMELA (Linhas 26 a 36). O processo em PROMELA tem adicionado a ele um canal de comunicação assíncrono (Linha 26), definido a partir de todos os parâmetros de mensagens que este objeto pode receber (Linha 39). A partir da recepção de uma mensagem (Linha 33) o objeto escolhe (Linhas 5 a 17) uma mensagem que possa ser consumida, aplicando a Regra correspondente.

```
1  #define SIZE 3                  /* Tamanho dos canais de comunicação, definido pelo usuário */
2
3  mtype = { Msg, Start };                                /* Mensagens pertencentes a descrição */
4
5  inline rules_Ring_Entity() {                           /* Escolha de um Regra para aplicação */
6    if
7    ::(msg_name == Msg) && (par_m_uid > atr_uid);              /* Escolha da Regra RuleMsg_1 */
8        run Ring_Entity_RuleMsg_1(atr_next, atr_uid, par_m_uid, ver_id);
9    ::(msg_name == Msg) && (par_m_uid == atr_uid);             /* Escolha da Regra RuleMsg_2 */
10       atr_leader = true;                                  /* Aplicação da Regra RuleMsg_2 */
11
12   ...                                          /* Escolha das Regras RuleMsg_3 e RuleStart */
13
14   ::else;
15     this_chan!msg_name, par_m_uid;                  /* Reenvio de mensagens não aplicáveis */
16   fi;
17 }
18
19 proctype Ring_Entity_RuleMsg_1(chan atr_next; byte atr_uid;
20                                byte par_m_uid; byte ver_id) {
21   atr_next!Msg, par_m_uid;                                /* Aplicação da Regra RuleMsg_1 */
22 }
23
24 ...                                         /* Aplicação das Regras RuleMsg_3 e RuleStart */
25
26 proctype Ring_Entity(chan this_init; chan this_chan) {       /* Tipo de objeto Ring_Entity */
27   mtype msg_name;                                  /* Contém o nome das mensagens recebidas */
28   byte ver_id = _pid;                           /* Identificação usada na verificação formal */
29   bool atr_leader; chan atr_next; byte atr_uid;                    /* Atributos do objeto */
30   byte par_m_uid;                                  /* Parâmetros das mensagens recebidas */
31   this_init?atr_leader, atr_next, atr_uid;      /* Inicialização dos atributos dos objetos */
32   do
33   ::this_chan?msg_name, par_m_uid;                              /* Recepção de mensagens */
34     rules_Ring_Entity();                           /* Seleção para aplicação de um Regra */
35   od;
36 }
37
38 init {                                                  /* Mapeamento do Grafo Inicial */
39   chan chan_Ring_Entity_1 = [SIZE] of {mtype, byte};          /* Canal do objeto Ring_Obj1 */
40   chan chan_Ring_Entity_1_init = [0] of {bool, chan, byte};    /* Canal para inicialização */
41
42   ...                     /* Criação de outros canais, pertencentes a Ring_Obj2 e Ring_Obj3 */
43
44   atomic {
45     run Ring_Entity(chan_Ring_Entity_1_init, chan_Ring_Entity_1);  /* Criação de Ring_Obj1 */
46     chan_Ring_Entity_1_init!false, chan_Ring_Entity_2, 1;    /* Inicialização de Ring_Obj1 */
47
48     ...                                 /* Criação e inicialização de Ring_Obj2 e Ring_Obj3 */
49
50     chan_Ring_Entity_1!Start, 0;                 /* Envio da mensagem Start para Ring_Obj1 */
51
52     ...                       /* Envio das outras mensagens Start para Ring_Obj2 e Ring_Obj3 */
53   }
54 }
```

Figura 3: Mapeamento do exemplo na GGBO definido anteriormente.

Neste mapeamento, a semântica de paralelismo implícito para a aplicação de Regras na GGBO não é completamente respeitado, sendo que as Regras que modificam o estado interno do objeto são aplicadas de forma serial (Linha 10). Já as Regras que não modificam o estado interno do objeto podem ser aplicadas de forma paralela, sendo estas criadas na forma de processos (Linhas 19 a 22 e 24).

O Grafo Inicial da GGBO é mapeado para um processo inicial em PROMELA (Linhas 38 a 54). Este processo tem como objetivo definir canais usados pelos objetos e canais para a inicialização dos atributos dos objetos (Linhas 39 a 42). Além disto, são criados os processos que correspondem aos objetos definidos no Grafo Inicial (Linhas 45 a 48) e enviadas as mensagens *Start* definidas no Grafo Inicial (Linhas 50 a 52).

6. Conclusão

Este artigo apresentou uma proposta de abordagem para a verificação formal, baseada na verificação de modelos, de SDBOs. A verificação é viabilizada a partir do mapeamento de um sistema descrito na linguagem de especificação formal GGBO (usada como linguagem de descrição) para um modelo na linguagem de descrição PROMELA.

A partir deste mapeamento o desenvolvedor tem de especificar as propriedades desejadas para o modelo e verificá-las usando o SPIN. Assim, o desenvolvedor inicia um processo de especificação $\Rightarrow$ verificação $\Rightarrow$ correção. Quando uma propriedade desejada é verificada e retorna o valor falso, o desenvolvedor, com base no contra-exemplo gerado, corrige o modelo para que a propriedade venha a ser verdadeira numa próxima verificação. No momento em que as propriedades desejadas são verdadeiras sobre o modelo, diz-se que o modelo é mais confiável. Isto se deve a garantia formal de que as propriedades desejadas são mantidas.

Devido a restrições de espaço não foi possível apresentar uma metodologia para a especificação e verificação de propriedades. Este tipo de metodologia existe no contexto do trabalho, porém, ainda exige que o usuário conheça o mapeamento de GGBO para PROMELA apresentado neste artigo. No entanto, como pode ser visto na Seção 5., as construções da GGBO puderam ser mapeadas de maneira bastante direta para as construções em PROMELA, o que facilita a compreensão da descrição mapeada.

Além disto, o mapeamento aqui proposto ainda não apresenta uma prova formal. Esta prova formal está sendo estudada e é vista como um importante trabalho futuro. Entretanto, pode-se observar que, apesar de não existir uma prova formal o mapeamento mantém (de acordo com a restrição de paralelismo evidenciada na Seção 5.) a semântica da GGBO.

Com relação a ferramentas computacionais, existe uma interface básica para a definição, armazenamento e recuperação de modelos na GGBO. A partir de descrições armazenadas é possível gerar: **i)** código para simulação, **ii)** código para execução em um ambiente real, ou **iii)** código para verificação formal, usando a abordagem proposta neste artigo.

Referências

Avizienis, A., Laprie, J.-C., and Randell, B. (2001). Fundamental concepts of dependability. Technical Report 01-145, LAAS (Laboratory for Analysis and Architecture of Systems), Taulosse, France.

Cho, S. M., Bae, D. H., Cha, S. D., Kim, Y. G., Yoo, B., and Kim, S. (1999). Applying Model Checking to Concurrent Object-oriented Software. In *4th International Symposium on Autonomous Decentralized Systems*, pages 380–383, Tokyo, Japan. IEEE Computer Society Press.

Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). *Model Checking*. MIT Press, Cambridge, MA, USA.

Clarke, E. M., Wing, J. M., Alur, R., Cleaveland, R., and et al (1996). Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643.

Copstein, B., da Costa Móra, M., and Ribeiro, L. (2000). An Environment for Formal Modeling and Simulation of Control Systems. In *33rd Annual Simulation Symposium*, pages 74–82, Washington, USA. IEEE Computer Society Press.

Dotti, F. L. and Ribeiro, L. (2000). Specification of Mobile Code Systems Using Graph Grammars. In *4th International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 177, pages 45–63, Stanford, CA, USA. Kluwer - IFIP Conference Proceedings.

Duarte, L. M. (2001). Desenvolvimento de Sistemas Distribuídos com Código Móvel a partir de Especificação Formal. Dissertação de mestrado, Pontifícia Universidade Católica do Rio Grande do Sul, Faculdade de Informática - Programa de Pós-Graduação em Ciência da Computação, Porto Alegre, RS, Brasil.

Ehrig, H. (1979). Introduction to the Algebraic Theory of Graph Grammars. In *1st International Workshop on Graph Grammars and Their Application to Computer Science and Biology*, volume 73, pages 1–69, Berlin, Germany. Springer-Verlag - Lecture Notes in Computer Science.

Holzmann, G. J. (1997). The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295.

Leue, S. and Holzmann, G. (1999). v-Promela: a visual, object oriented language for SPIN. In *2nd International Symposium on Object-Oriented Real-Time Distributed Computing*, Saint-Malo, France. IEEE Computer Society Press.

Lilius, J. and Paltor, I. P. (1999). vUML: A Tool for Verifying UML Models. In *14th International Conference on Automated Software Engineering*, pages 255–258, Cocoa Beach, Florida, USA. IEEE Computer Society Press.

Lynch, N. A. (1996). *Distributed Algorithms*, pages 476–482. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Manna, Z. and Pnueli, A. (1992). *The Temporal Logic of Reactive and Concurrent Systems - Specification*. Springer-Verlag, New York, NY, USA.

Rödel, E. T. (2003). Modelagem Formal de Falhas em Sistemas Distribuídos envolvendo Mobilidade. Dissertação de mestrado, Pontifícia Universidade Católica do Rio Grande do Sul, Faculdade de Informática - Programa de Pós-Graduação em Ciência da Computação, Porto Alegre, RS, Brasil.

Winter, K. and Duke, R. (2002). Model Checking Object-Z using ASM. In *3rd International Conference on Integrated Formal Methods*, volume 2335, pages 165–184. Lecture Notes in Computer Science, Spinger-Verlag.

The MARES Platform: Support for Transactional and Fault-tolerant Execution of Mobile Agent-based Applications

Flávio M. Assis Silva, Raimundo J.A. Macêdo, Ana Vitoria Piaggio Freitas

LaSiD – Distributed Systems Laboratory
Federal University of Bahia (UFBA)
Av. Adhemar de Barros, S/N
Bairro de Ondina, Salvador/BA – CEP 40170-110

{fassis, macedo, piaggio}@ufba.br

Abstract

Support for transactional behaviour and fault tolerant executions of mobile agent-based applications is a fundamental issue in the development of mobile agent systems. We are developing the MARES platform, which supports the modelling of mobile agent-based applications as distributed transactions. The executions of mobile agent-based transactions on top of the MARES platform are fault-tolerant, in the sense that if the location in the distributed environment where a part of a global transaction is being executed becomes faulty for a long time, the system performs a recovery procedure to resume the execution of that part of the transaction at another location. In this paper we present the transaction model used in the MARES platform, we outline the interface for modelling mobile agent-based transactions, and we discuss alternatives for implementing mobile agent fault tolerance that increase the level of flexibility for modelling reliable mobile agent-based applications when compared with previously proposed approaches.

1. Introduction

A *mobile agent* (or simply *agent*) is a self-contained software element responsible for executing a programmatic process, which is capable of *autonomously migrating* through a network. An agent migrates in a distributed environment between logical ”places“ referred here to as *agencies*. When an agent migrates, its execution is suspended at the original agency, the agent is transported (i.e. program code, data, execution state and control information) to the destination agency, and, after being re-instantiated, it resumes execution. The mobile agent concept is being proposed to support different types of applications, including electronic commerce, workflow management systems and network management due to the potential benefits that might be achieved by exploring its asynchronous way of execution [1, 3].

For at least some types of mobile agent-based applications (such as electronic commerce or workflow applications) it is fundamental that the executions of these applications exhibit strong reliability properties. Among the reliability requirements are the need for the executions of the applications to be fault tolerant and to exhibit some transactional semantics, and that groups of mobile agents can coordinate their activites by using a mechanism of reliable group communication [5].

We are developing and implementing a platform called MARES, which provides solutions for these reliability requirements, at LaSiD/UFBA (Distributed Systems Laboratory / Federal University of Bahia). This platform provides for its applications an interface through

which *mobile agent-based transactions* can be modelled. A mobile agent-based transaction is a distributed transaction that is carried out by a set of mobile agents. The control of the execution of such a transaction becomes thus decentralized and the components that execute the transaction (the agents) might change their location in the distributed environment. In the MARES platform the executions of mobile agent-based transactions are fault tolerant. The execution of a part of a transaction at an agency that becomes faulty is recovered locally when that agency restarts normal execution. Additionally the execution of that part of the transaction is resumed at *another* agency, if the failure lasts long.

In this paper we present the transaction model provided by the MARES platform and we discuss alternatives for implementing mobile agent fault tolerance. Approaches for mobile agent fault tolerance and the modelling of transactions on top of such approaches have been proposed in the literature, e.g. [1, 4, 7]. The transaction model implemented in the MARES platform is based on the model presented in [1]. The alternatives for implementing mobile agent fault tolerance presented in this paper provide more flexibility for modelling reliable mobile agent-based applications than the existing approaches.

This paper is structured as follows. In section 2 we present the architecture of the MARES platform. In section 3 we present the transaction model used in the platform and the interface for specifying these transactions. In section 4 we discuss alternatives for implementing mobile agent fault tolerance. Section 5 describes a concept upon which these alternatives might be implemented, the mobile groups. Finally, section 6 concludes the paper.

2. The MARES Platform

The MARES Platform is structured in three layers, as shown on Figure 1.

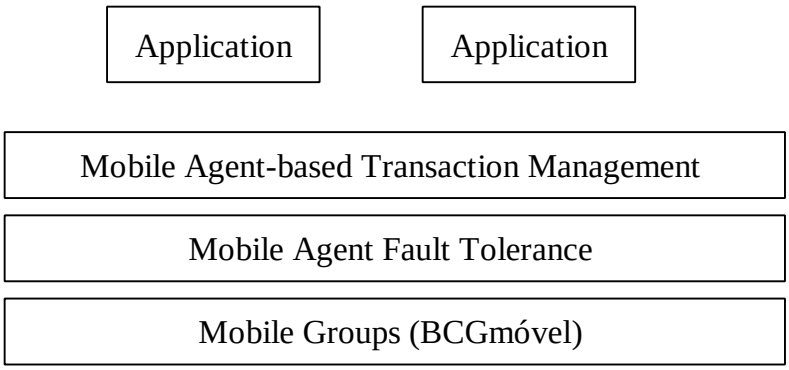


Figure 1: The Layers of the MARES Platform

The platform is based on a group communication system that implements the concept of *mobile groups* (lowest layer). The concept of mobile group was introduced by Macêdo and Assis Silva [6] and is an extension of the traditional concept of a group of cooperating processes in which a member process can move from a location in a distributed system to another while continuing to be a member of the group. Analogously to traditional group systems, mobile groups also provide message delivery guarantees and virtual synchrony. Traditionally, mobile group systems (such as Horus, Transis, NewTop, among others) do not inherently support process migration. Making process migration an integrated functionality of a process group system makes the implementation of migration of a group member more efficient and allows the synchronization of movement with other group events (such as joins

and crashes of processes). The group communication system being developed at LaSiD that implements the mobile groups concept is called BCGmóvel (*Reliable Group Communication System with Mobility Support*). As discussed in [5], a mobile group can be considered to be an underlying functionality in a mobile agent system to fulfill some of the reliability requirements of mobile agent-based applications.

On top of the layer providing mobile group support is a layer implementing a protocol for mobile agent fault tolerance. If the agency where an agent is running fails, the execution of that agent remains blocked while the agency is faulty. Long unavailability periods of that agency have the obvious undesirable effect of delaying the execution of an agent-based application. It is desirable to have a mechanism that makes it possible to recover the execution of a computation being performed by an agent at an agency that has remained faulty for a long time from some other agency in the distributed environment (for possibly following an alternative execution path, restarting from a previous consistent execution state).

The protocolos proposed so far for mobile agent fault tolerance are based on mobile agent replication, e.g. [1, 4, 7]. When a migration is performed, the code and state of a mobile agent-based application is sent to a set of locations, instead of just to one. This set of agents executes as a group. If one of them fails, the other agents might recover the execution of the group and continue its execution from a previous consistent state. The coordination of this set of agents is the goal of the *Mobile Agent Fault Tolerance* layer of the MARES Platform.

On top of the Mobile Agent Fault Tolerance layer is the *Mobile Agent-based Transaction Management* layer. This layer provides the needed functionality for enabling mobile agent-based applications to run as global distributed transactions. The functionality implemented at this layer guarantees that the whole system will remain in a consistent state independently of concurrent executions of global transactions. One or more mobile agents can be used to execute a single global transaction.

As examples of applications to run on top of the MARES platform are electronic commerce and workflow applications. At LaSiD we are developing a workflow management system based on mobile agents, called MAB*flow* (*Mobile Agent-based Workflow Management System*). In this system, workflows are specified in UML (*Unified Modelling Language*). Tools are provided that generate an XML (*Extended Markup Language*) document that describes the structure of the workflow specified in UML. From this XML document, Java code is automatically generated that models the workflow as a global transaction using the interface provided by the MARES platform.

The platform will include also tools for supporting the analysis of correctness of mobile agent-based applications. The goal of these tools will not be discussed here.

3. The Mobile Agent-based Transaction Management Layer

The application program interface (API) of the MARES platform is the interface provided by the *Mobile Agent-based Transactional Management Layer*. This API consists of a set of Java classes and interfaces. By using this API, applications can be modelled as mobile agent-based transactions.

3.1. The Transaction Model

A transaction is a concept used to guarantee correct and reliable executions of programs accessing resources concurrently in an environment subject to faults. The most simple transaction model is the *atomic (flat) transaction* model. Atomic transactions satisfy a set of properties commonly referred to as ACID (*Atomicity, Consistency, Isolation and Durability*) properties [2]. In this model, intermediate results of a transaction are not visible to other transactions (the *isolation property*) and all of the effects produced by the transacion are made

durable in the system or none of them will be (*atomicity* and *durable* properties). Although atomic transactions represent a suitable abstraction for modelling concurrent accesses to shared data, it is well known that they do not fulfill requirements on transaction processing of complex applications, for example, due to the need for modelling long duration activities [8].

A set of so-called *extended transaction models* were then proposed to fulfill the requirements of such applications [8]. In the models that do not provide the isolation property, the recovery of transactions is usually performed by using *compensation*. A transaction the effects of which must be cancelled (the *compensated-for* transaction) has an associated *compensating transaction*. The compensating transaction is able to cancel the effects produced by the compensated-for transaction *from a semantic point-of-view.* That is, the resulting state produced in the system by the compensating transaction is not necessarily equal to the state read by the compensated-for transaction before its execution. The resulting state is only *related* to the original one according to the semantics of the applications.

The transaction model implemented in the MARES platform is an *open nested transaction model*. An open nested transaction can be composed of other transactions, called its *subtransactions*. Each subtransaction might have its own subtransactions, resulting this way in a transaction tree. The root of the transaction tree is an open nested transaction. The subtransactions of an open nested transaction can be *(flat) atomic* or also *open nested*. Differently from atomic transactions, the intermediate states of an open transaction are made visible to other transactions and its recovery is based on compensation.

Atomic and open transactions are used to model different types of activities in the MARES platform. A flat transaction executes on a single agency and its code consists typically of a set of invocations of methods of objects providing services in the distributed environment (for example, an application database). These methods are to be invoked *locally*, i.e., with the agent performing the transaction and the object providing the service in an environment where the invocations can be made with a good quality of service (for example, with both the agent and the service provider object at the same host or at different hosts but at the same local area network). The control flow of an open transaction, on the other hand, defines how its subtransactions are to be executed. They do not access directly services at the interface of objects. Subtransactions of an open transaction can be executed sequentially or in parallel. Sets of alternative transactions can also be defined. These sets are composed by transactions which provide equivalent results to the application, but at most one of the transactions in the set can commit.

Compensating transactions are defined as follows. Each transaction, with exception of the root transaction, has associated with it a compensating transaction. The compensating transaction of an atomic transaction is another atomic transaction, written by the application programmer. The compensating transaction of an open transaction is another open transaction, but defined automatically during runtime. The compensating transaction of an open transaction cancels the effects of the subtransactions that were committed, by executing the compensating transactions of its subtransactions, in the reverse order of their executions.

A transaction can also be *vital* or *non-vital*, depending on the impact that a failure of this transaction has on its parent transaction. A transaction is vital if a failure during its execution implies that the effects of its parent transaction must also be cancelled. If the execution of a non-vital transaction fails, the execution of its parent transaction can continue.

Transactions can be grouped into *alternative sets*. Alternative sets model sets of transactions that provide equivalent effects on the system (according to application semantics). Transactions of an alternative set have priorities. The transaction with the highest priority is the one which is executed first. If this transaction fails, the transaction with the next

priority is executed. The process is repeated until either one of the transactions in the set is executed successfully or there is no more transactions to execute in the set.

3.2. The Transaction Support Interface

Two general classes were defined, that represent the basic types of transactions:

- *FlatTransaction*: used to model leaf (atomic) transactions in a transaction tree;
- *OpenTransaction*: corresponds to non-leaf nodes in a transaction tree (open transactions).

The class *OpenTransaction* has three specializations: *SequentialOpenTransaction*, *ParallelOpenTransaction* and *AlternativeOpenTransaction*. Each of these specializations has associated with it a set of transactions, each of which might be, on its turn, either a *FlatTransaction* or one of the specializations of *OpenTransaction*.

The different specializations of *OpenTransaction* define specific ways of executing the transactions in the associated set as follows.

In the case of *SequentialOpenTransaction*, each transaction in the set is associated with a set of conditions that must be satisfied in order for that transaction to be executed. The conditions are boolean expressions that can involve values of data items processed in the transaction or the outcome of previously executed transactions. When a *SequentialOpenTransaction* is to be executed, the sets of conditions associated with the transactions in the set are repeatedly evaluated. If all the conditions in the set associated with a transaction are evaluated to true, that transaction is executed. If that happens to more than one transaction in the set, one of them is randomly chosen.

In the case of *ParallelOpenTransaction*, all transactions in the associated set are executed in parallel.

In the case of *AlternativeOpenTransaction*, each transaction in the associated set has a different priority. The execution of an *AlternativeOpenTransaction* begins with the execution of the transaction with the highest priority. If that transaction fails, the transaction with the next highest priority is executed. This process is repeated until some of the transactions is executed successfully or until there is no more transaction in the set to be executed.

A mobile agent-based transaction is thus formed by choosing the type of transaction that apropriately models the type of control flow desired. If the chosen transaction is an open transaction, then transactions of appropriate classes are inserted in the set. The whole transaction tree is thus built by recursively repeating this procedure.

```
1 Class ExampleTransaction extends SequentialOpenTransaction {
2       ...
3  protected boolean controlFlow() {
4       ExampleFlatTransaction t1 = new ExampleFlatTransaction( “t1”, true, “ag1”);
5       ExampleFlatTransaction t2 = new ExampleFlatTransaction(“t2”, true,”ag2”);
6       this.addSubTransaction(t1);
7       this.addSubTransaction(t2);
8       this.addCondition(“t2”, newCommitCondition(t1));
9       ...
10 }
```

Figure 2: Example of part of a mobile agent-based transaction

An example of part of the specification of a mobile agent-based transaction is shown on Figure 2. In this example, the code of a sequential open transaction is defined. In lines 4 and 5 two atomic transactions were defined, *t1* and *t2*. In lines 6 and 7 these transactions are defined as subtransactions of the *SequentialOpenTransaction* being specified. In line 8 the condition for the execution of transaction *t2* is defined. Transaction *t2* will only execute if transaction *t1* commits. Since there is no condition associated with *t1* it is eligible to be executed when its parent transaction begins executing.

At any time, if the execution of a vital transaction fails, the compensation process starts. This process executes the compensating transaction for each of the committed transactions, in the reverse order of their execution.

4. The Mobile Agent Fault Tolerance Layer

4.1. Executing Mobile Agent based Transactions

Agent-based transactions are executed by a set of one or more mobile agents. As previously highlighted, mobile agent fault tolerance is achieved by replicating agents. With replication, an agent execution is seen as being performed in a sequence of *stages*. The first stage begins when the application execution starts. A new stage then begins when the mobile agent execution reaches a move operation, to continue at a new agency. Whenever a new stage of execution is achieved, new replicas of the agent with the current execution state are created at a new set of agencies. The agent copies at the terminating stage are destroyed. Stages are created and terminated in this way until the end of the execution of the application.

In order to illustrate how the management of the agent replicas is done, consider the case of the execution of an *AlternativeOpenTransaction* which, for example, has three atomic transactions in its corresponding transaction set, each of them to be executed at a different agency. This set of transactions will be executed by a set of mobile agents, all with the same code and initially with the same execution state information (for instance, the current state of the transaction which includes this *AlternativeOpenTransaction* as one of its subtransactions). One agent is sent to each of the agencies where transactions in the set are to be executed. The agent at the agency where the transaction with the highest priority is to be executed starts executing. If that agent fails, the other agents in the group *elect* the agent in the group with the transaction with the next highest priority to resume the execution of the *AlternativeOpenTransaction*. If the agency where that agent is fails, the process is repeated. If one of the transactions in the set executes successfully or all of the transactions in the set has been unsuccessfully executed, the execution of the *AlternativeOpenTransaction* will have been terminated. A new configuration of agents at agencies will thus be created to continue the execution of that *AlternativeOpenTransactions*'s parent transaction.

4.2. Alternatives for Managing the Execution of Agent-based Applications by Sets of Agents

There are two issues involved in the management of replicated agents: preserving the computation and data integrity while executing a stage; and managing the creation and destruction of stages.

Preserving computation and data integrity relates to managing how the members of a stage will execute their actions in such a way that the consistency of the whole system is guaranteed. For example, if an agent executes some action at an agency, this agency fails and some other member of the group resumes the execution of the transaction, specific actions must be carried out when that agency recovers from the failure in order to cancel the effects performed there before the failure.

Managing the creation and destruction of stages relates to coordinating the creation of the configuration of agent copies that will participate in a stage and destroying the configuration that was used to execute the previous stage. An important issue involved in that is the guarantee of *exactly-once semantics*. Informally, exactly-once semantics means in this context that each step of an application executed by mobile agents must be executed exactly once. Exactly-once semantics is achieved by guaranteeing that during a stage the results produced by only one of the replicas can be committed (and used to form the replicas forming a new stage). For example, if more than an agent tries to create a new stage, only one of them can succeed.

The approaches for mobile agent fault tolerance proposed so far (e.g. [1, 4, 7]) consider alternatives for preserving the computation and data integrity, but all of them consider exactly-once semantics. For example, as described above, copies of an agent could be the members of a stage to execute the set of transactions of an *AlternativeOpenTransaction*. We could alternatively have a similar set of transactions all with the same priority. In this case, a set of agents could be used to execute these transactions, but now all the transactions *can be executed in parallel*. However, only one of them must commit. In this case the exactly-once must still be enforced.

Exactly-once property is not, however, necessary in all cases. With exactly-once semantics, at any point in the execution, the results of the execution of only one of the agents in the group can commit. We can decouple the management of stages from the need for having fault tolerance of the mobile agent-based application. For example, consider that at a certain point in the execution of an application, three independent execution threads exist (for example, three sequences of atomic transactions that can be executed in parallel). In this case, we could have three agents, each executing one of the threads. The group of three agents is still used to provide fault tolerance for the application, but now they can migrate independently one from the others. The agents might exchange messages during execution in order to store information about the execution of their threads (in case of failure of one of the agents, a recovery procedure can use that information to resume execution from a more recent previous consistent state).

5. The Mobile Groups Layer

Mobile groups [6] provide the necessary mechanism to implement the alternatives discussed in the previous section. The virtual synchrony of Mobile Groups allows all replicas of an agent to identify agent failure (e.g., the leader failure) in a mutually consistent manner. Atomicity and FIFO message delivery properties can be used to assert that all replicas perceive the same set of actions executed by other agents (e.g., the leader). The exactly-once semantics is a direct benefit of using Mobile Groups and its total order of views. In general terms, it can be achieved by having just one leader at any instant and allowing only the leader to create a new stage. A leader can be uniquely chosen by applying a function on the identifiers of the view members. When a leader decides to move and the movement succeeds, a new view is generated and a new stage begins. The new view will describe the new stage configuration. When the current leader is suspected to have failed, it will leave the group and its actions will not have effects on the group any longer (if it was trying to create a new stage, for example, this new stage will not be created). A new leader is then elected to continue the stage execution. The current execution state information can be transmitted by the leader to the other group members by multicasting it before moving. Other actions, such as the creation of a new group of agents, can be achieved similarly.

6. Conclusion

With the development of the MARES platform we are analysing a set of alternatives for the development of reliable mobile agent-based applications. Although a set of approaches for addressing reliability requirements of such applications has been proposed in the literature, there is still many aspects to be analysed in more depth yet. In particular, we are more deeply interested in decreasing the cost of solutions for mobile agent-based applications reliability, in identifying fundamental components of a platform that implements such solutions, and in relaxing the restrictions that are imposed by the existing proposals. The platform described in this paper corresponds to some of the results we achieved in these directions so far.

Currently the transaction support as described here is already implemented. It runs, however, on top of a simulation of the *Mobile Agent Fault Tolerance* layer. Although a prototype of the mobile groups concept is already implemented, there are still some componentes to be implemented yet to turn it into a completely functional tool.

We are currently porting a workflow management system, the MAB*flow*, in development at LaSiD to run on top of the MARES platform. The current version of *MABflow* is operational at LaSiD, but without fault tolerance and transactional support. A mapping from the XML description of a workflow to a corresponding modelling of the workflow in terms of a mobile agent-based transaction is under development. We are currently investigating how the support for disconnected operation that is implemented in the current version of MABflow can be best integrated when MARES is used to execute the workflows.

Referências

[1] Flávio M. Assis Silva. *A Transaction Model based on Mobile Agents.* PhD thesis, Universidade Técnica de Berlim, Berlim, Alemanha, 1999.

[2] J.Gray, A.Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers. California, USA. 1993

[3] A. Fuggeta, G. Picco and G. Vigna. Understanding Code Mobility. IEEE Transactions of Software Engineering, 24(5), Maio 1998.

[4] K.Rothermel, M.Strasser. A Fault-Tolerant Protocol for Providing the Exactly-Once Property of Mobile Agents. *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS´98)*. West Lafayette, USA. Outubro, 1998

[5] F.M.Assis Silva, R.J.A.Macêdo. Reliability Requirements in Mobile Agent Systems. *Proceedings of the Second Workshop on Tests and Fault Tolerance (II WTF 2000)*. Curitiba, Brazil. July, 2000.

[6] R.J.Araújo Macêdo, F.M.Assis Silva. Coordination of Mobile Processes with Mobile Groups. *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN2002)*. Washington, DC, USA. June 2002. Pp. 177-186

[7] S.Pleisch, A.Schiper. Modeling Fault Tolerant Mobile Agent Execution as a Sequence of Agreement Problems. *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*. Nuremberg, Germany. Oct. 2000

[8] A.K.Elmagarmid (ed.). *Database Transaction Models for Advanced Applications*. Morgan-Kaufmann Publishers. USA. 1992

Uma Arquitetura Altamente Disponível Aplicada a Sistemas de Controle Embutidos de Tempo Real[1]

Cesar Ossamu Ida, **Taisy Silva Weber**

Instituto de Informática - Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15064 – 91501-970 – Porto Alegre – RS

{cesarida, taisy}@inf.ufrgs.br

Resumo. *Uma arquitetura baseada em redundância de controladores para aumentar a segurança (safety) e disponibilidade de sistemas embutidos de tempo real é apresentada. Dois controladores (de prateleira) processam os mesmos dados de entrada e os dados computados são comparados, como forma de detecção de erros. Quando um erro é detectado, uma rotina de diagnóstico tenta identificar sua localidade. Isto pode demorar alguns ciclos de controle, o que viola a propriedade de tempo real da aplicação controlada; entretanto, a violação pode ocorrer na classe de aplicações embutidas sendo considerada, desde que não cause o colapso do processo controlado.*

Abstract*. An architecture based on controller redundancy for increased embedded real-time systems safety and availability is presented. Two COTS controllers process the same inputs and the computed data is compared as a means to detect errors. When an error is detected, a diagnosis routine tries to identify its locality. This may take some control cycles, which violates the real-time property of the controlled application; however, this situation is allowed to happen for the embedded systems we consider as long as it does not cause the failure of the controlled process.*

1. Introdução

Um dos maiores obstáculos à ampla utilização de sistemas tolerantes a falhas em computadores embutidos de tempo real é o alto custo inerente à redundância empregada, seja ela temporal ou espacial. Durante alguns anos, sua aplicação foi restrita a sistemas críticos de custos altíssimos, justificados pelos potenciais prejuízos causados por falhas nestes sistemas. Entretanto, fatores como a redução nos preços dos componentes de hardware para computadores, a redução do próprio tamanho desses componentes e produção em larga escala, contribuíram para sua maior utilização em diferentes aplicações comerciais e industriais. Atualmente, técnicas de tolerância a falhas têm sido utilizadas em sistemas de controle como plantas industriais, automóveis, usinas de eletricidade, plataformas de petróleo, controle de tráfego, navegação, controle de trens, sistemas autônomos, aplicações militares, edifícios inteligentes, etc..

Em algumas destas aplicações, um defeito no sistema pode colocar em risco a vida de pessoas, danificar equipamentos ou levar a prejuízos monetários bastante elevados [Laprie 1998], o que justifica altos investimentos em técnicas de tolerância a falhas para manter estes sistemas funcionando, mesmo na presença de falhas. Estas aplicações envolvem os sistemas

[1] Este trabalho é parcialmente financiado pelo projeto CT-Petro, desenvolvido em cooperação entre a Universidade Federal do Rio Grande do Sul, UNISINOS e a empresa Altus Sistemas de Informática.

chamados de "ultra-seguros" (*ultra-dependable*), bem como os sistemas "altamente-seguros" (*highly-dependable*) [Suri 1995]. Por outro lado, algumas destas aplicações possuem requisitos de confiabilidade, disponibilidade e segurança mais baixos que as primeiras e não são consideradas críticas. Estas, são as aplicações "altamente disponíveis" (*highly-available*). Este tipo de aplicação criou um novo nicho para o emprego de sistemas tolerantes a falhas, e se caracteriza pelo custo mais baixo com relação aos sistemas "ultra-seguros", requisitos temporais de resposta da ordem de centenas de milissegundos, execução cíclica (a maioria), computadores embutidos, e normalmente possuem um operador remoto que monitora o sistema. Nestas aplicações, a parada do sistema durante intervalos de tempo curtos são aceitáveis, desde que o mesmo seja colocado em um estado seguro em situações de falha. As aplicações "altamente-disponíveis" controladas por CLPs (Controladores Lógico-Programáveis)[2] são o foco deste trabalho.

Ao mesmo tempo em que se busca maior disponibilidade e segurança, o trabalho desenvolvido objetiva, também, manter os custos do sistema baixos, visto que um dos maiores obstáculos à ampla utilização comercial de sistemas tolerantes a falhas é o seu elevado custo. Por este motivo, a duplicação de CLPs é bastante utilizada comercialmente quando comparada a outros esquemas de redundância, como a triplicação e a quadruplicação. Outra maneira de se manter os custos do sistema baixos - além da utilização de apenas um CLP redundante - é através da utilização de componentes de prateleira (COTS). No sistema desenvolvido, todos os componentes de hardware (UCPs, redes de comunicação, interfaces de comunicação, etc..) e software (sistema operacional, drivers) utilizados são COTS. Este trabalho está sendo desenvolvido dentro do projeto CT-Petro, o qual visa o desenvolvimento de um sistema embutido de tempo real para o controle de plataformas de petróleo *off-shore*, que atenda a requisitos de confiabilidade e disponibilidade das licitações para plataformas da Petrobrás.

Este artigo trata da especificação e desenvolvimento de uma arquitetura de hardware baseada na duplicação de CLPs, que tem por objetivo adicionar características de tolerância a falhas a um sistema de controle embutido de tempo real. É apresentada, também, uma estratégia de tolerância a falhas implementada em software, complementar à arquitetura de hardware redundante. Esta, é baseada na técnica de redundância dinâmica [Pradhan e Banerjee 1996] e detecção de erros por comparação. A seção 2 do artigo descreve a arquitetura geral do sistema e do CLP empregado; a seção 3 o modelo de falhas; a seção 4 a estratégia de tolerância a falhas; a seção 5 a plataforma utilizada na implementação; e na última seção é feita uma análise geral do sistema.

2. Arquitetura de Hardware do Sistema

O sistema desenvolvido pelos autores, denominado HARTS (Highly Available Real Time System) é dividido em 3 subsistemas: supervisão, controle e campo (fig. 1).

O subsistema de supervisão liga o CLP aos computadores dos usuários através de uma rede de supervisão padrão Ethernet, e permite que os operadores monitorem o sistema. O subsistema de campo liga o CLP aos dispositivos de E/S, através de uma rede de campo padrão Profibus e módulos intermediários de E/S. O subsistema de controle é composto pelos próprios CLPs, que são interligados através de um canal de sincronização padrão Ethernet,

[2] O CLP é uma UCP, convencional ou não, somada a interfaces para periféricos e redes específicos para ambiente industrial [Jones 1983]

utilizado para troca de dados referentes ao controle de redundância. Se ocorrer falha nesta rede, os CLPs utilizam a rede de supervisão como um canal alternativo para troca de informações referentes ao diagnóstico da falha. Elimina-se, deste modo, a necessidade de duplicação da rede de sincronização.

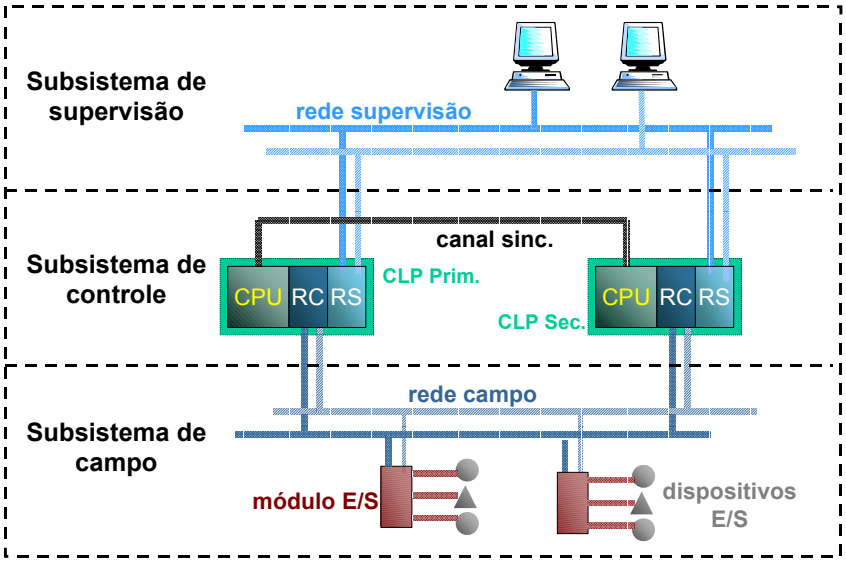


Figura 1. Arquitetura de hardware do sistema HARTS

Os CLPs são nomeados primário e secundário apenas como forma de referência. Ambos executam todas as tarefas em paralelo e possuem programas idênticos. Realizam a leitura dos dados de entrada (adquiridos através da rede de campo), os processam e geram novos valores que são enviados para os equipamentos de campo. Os CLPs comparam seus resultados antes de enviá-los para os equipamentos de campo, como forma de detecção de erros. Uma vez que um erro é detectado, rotinas de diagnóstico, recuperação e reintegração são executadas e, se necessário, uma reconfiguração automática de recursos é realizada para manter o sistema em funcionamento, sem afetar sua disponibilidade.

2.1 Arquitetura do CLP

Cada CLP do sistema HARTS é composto por 3 módulos: Módulo UCP, Módulo de Campo (MC) e Módulo de Supervisão (MS). Estes executam tarefas específicas e possuem baixa capacidade de processamento, sendo compostos por um microcontrolador, fonte de energia elétrica e ligações com suas respectivas redes de comunicação. Todos os módulos (de cada CLP) são interligados através de um barramento, através do qual comunicam-se entre si - o protocolo de comunicação entre módulos utilizado no projeto CT-Petro é o GBL, um protocolo mestre/escravo desenvolvido por, e de propriedade da Altus Sistemas de Informática. Por questões de simplificação, este barramento é assumido como livre de falhas. No HARTS, a falha permanente de qualquer um destes módulos causa a desativação de todo o CLP; neste caso, o sistema pode ser desligado ou continuar funcionando com apenas um CLP.

O Módulo UCP é o principal, controla as atividades do CLP, executa o programa de controle e gerencia as atividades relacionadas à tolerância a falhas. Possui ligação a um canal de sincronização padrão Ethernet (que a interliga a outro CLP), além de ligação ao barramento (GBL) que a interliga aos outros módulos.

O Módulo de Supervisão realiza comunicação com os usuários do sistema através de computadores e rede de supervisão, recebendo as requisições, numerando-as e repassando-as para a UCP. Possui ligação ao barramento GBL e à rede de supervisão (padrão Ethernet), que pode ser redundante (duplicada). As requisições de usuário, geradas através dos computadores de supervisão, fogem à execução cíclica do sistema, pois não se pode prever o momento em que um usuário gera uma requisição. Por este motivo, estas requisições são consideradas dinâmicas e podem causar inconsistências entre CLPs se atendidas em qualquer momento, ou em apenas um dos CLPs. No HARTS, estas requisições são devidamente tratadas pelos módulos de supervisão de maneira consistente e determinística, como é apresentado na seção 3 deste artigo.

O Módulo de Campo perfaz o papel de mestre de rede de campo e realiza comunicação e varredura dos módulos de E/S. Possui ligação à rede de campo (padrão Profibus), além de ligação ao barramento GBL. O HARTS permite que a ligação à rede de campo seja duplicada.

3. Modelo de Falhas

O HARTS suporta falhas de hardware, partindo da premissa que as mesmas sejam independentes e simples. Independente significa que a falha não afeta os dois CLPs[3] ao mesmo tempo (falhas correlacionadas); simples significa que nenhuma outra falha ocorre até que a primeira seja recuperada.

Os tipos de falhas toleradas, quanto à sua duração, são as falhas temporárias e permanentes ocorridas na UCP (Unidade Central de Processamento). As falhas temporárias são aquelas que ocorrem por curtos instantes de tempo, como um eventual ruído. Se não houver como tratar o erro sem parar o sistema, a simples reinicialização do mesmo pode ser suficiente para resolvê-lo. Algumas falhas temporárias são periodicamente recorrentes e, neste trabalho, são consideradas permanentes. Falhas permanentes são falhas de hardware. Não há como recuperar o componente que sofreu falha permanente sem a substituição do mesmo.

Assume-se que falhas no canal de comunicação (sincronização) entre os CLPs (interface ou *link* de comunicação) sejam falhas do tipo colapso (*crash*). Falhas de projeto, implementação e de software não são tratadas, pois necessitam de mecanismos de detecção e tratamento diferentes dos utilizados neste trabalho. Falhas nos dispositivos de E/S não são consideradas, assume-se que os mesmos sempre funcionam corretamente. Os códigos executáveis dos programas são armazenados em memória ROM (Read Only Memory) e, portanto, imunes a interferências e corrupção de dados.

4. Estratégia de Tolerância a Falhas

O CLP, sob uma visão bastante simplificada, lê um conjunto de dados de entrada do ambiente (através dos módulos de E/S e sensores), processa esses dados e gera um conjunto de resultados que são utilizados neste ambiente (através dos módulos de E/S e atuadores). Estas atividades devem ser executadas em um tempo finito, predefinido, porém, variável de acordo com cada aplicação.

[3] Os CLPs são fisicamente e eletricamente isolados um do outro e, portanto, apresentam falhas independentes, dentro dos limites cabíveis

No HARTS, as UCPs dos dois CLPs executam os mesmos programas em paralelo: o sistema operacional e o programa de controle devem ser idênticos e determinísticos. Os resultados dos programas de aplicação (controle) são comparados bit-a-bit a cada ciclo, como forma de detecção de erros. O método de comparação permite a detecção rápida de erros (pequeno atraso), fator importante para contenção de seus efeitos nocivos na UCP, de modo que não se espalhem pelo sistema.

Para evitar que seja um ponto único de falha, o comparador é distribuído em ambas UCPs. Estas, comunicam-se através de troca de mensagens. Após a comparação bem sucedida de resultados, ambas UCPs os enviam para os mestres de rede de campo, que por sua vez os repassam para os módulos de E/S. Os equipamentos de rede de campo também poderiam comparar os resultados antes de utilizá-los, para garantir que nenhum erro tenha ocorrido entre o momento da comparação nas UCPs, transmissão através da rede de campo e sua recepção nos equipamentos. A adoção da solução de comparar resultados pelos equipamentos de campo depende do método de sincronização empregado nos mesmos, mas não é essencial para o funcionamento da estratégia de tolerância a falhas. No HARTS a comparação é realizada na UCP antes do envio dos dados, mas não é realizada pelos equipamentos de campo no momento de sua recepção.

Após a inicialização do sistema, as UCPs fazem a leitura dos dados de entrada, contidos nos respectivos mestres de rede de campo - assume-se que o sistema inicia seu funcionamento livre de falhas. Em seguida, o programa de aplicação é executado, o qual gera um conjunto de dados de saída, que não são imediatamente enviados para os equipamentos de rede de campo. Ao contrário, o procedimento de comparação de resultados é acionado. Se os valores gerados pelos programas de aplicação das duas UCPs forem iguais, o sistema continua seu fluxo de execução normalmente, atendendo às requisições de usuários e iniciando o ciclo seguinte. Se os valores não forem idênticos, significa que ocorreu alguma falha e os procedimentos relacionados à tolerância a falhas são executados. Estes procedimentos são descritos na seção seguinte.

4.1 Tarefas de Tolerância a Falhas

Após a execução do programa de aplicação, são executadas as tarefas relacionadas à tolerância a falhas, quais sejam a comparação, diagnóstico, recuperação e reintegração.

A primeira delas a ser executada é a comparação, para detecção de erros. Se nenhum erro for detectado, os valores de saída são enviados aos dispositivos de saída (atuadores), as requisições de usuário (se existirem) são processadas e o ciclo inicia-se novamente. Por outro lado, se algum erro for detectado no momento da comparação, os dados de saída do ciclo anterior são mantidos nos módulos de Rede de Campo e de E/S, e o fluxo de execução da UCP é desviado para a rotina de diagnóstico. O diagnóstico deve apontar em qual das UCPs a falha ocorreu; dependendo de seu resultado, a rotina de recuperação é invocada na UCP em falha, enquanto a UCP livre de falhas inicia o ciclo seguinte e mantém o sistema operacional. Se a UCP em falha for recuperada, é reintegrada ao sistema e seu estado é sincronizado novamente com o estado da UCP não falha – a reintegração também é realizada quando a UCP é substituída. Se não for possível recuperar o componente em falha, ele é desligado do sistema, que continua seu funcionamento com apenas um CLP controlando o processo; neste caso, não é mais possível realizar a detecção de erros através da comparação, mas mecanismos auxiliares de detecção podem ser empregados.

Se o diagnóstico das UCPs não for satisfatório, ou seja, se não for possível determinar em qual das UCPs ela ocorreu, então não é possível manter o sistema em funcionamento: ambas UCPs são suspeitas e devem ser desligadas para que não executem algum comportamento errôneo. O sistema entra em um estado seguro, livre de falhas. Este processo pode ser melhor visualizado na fig. 2:

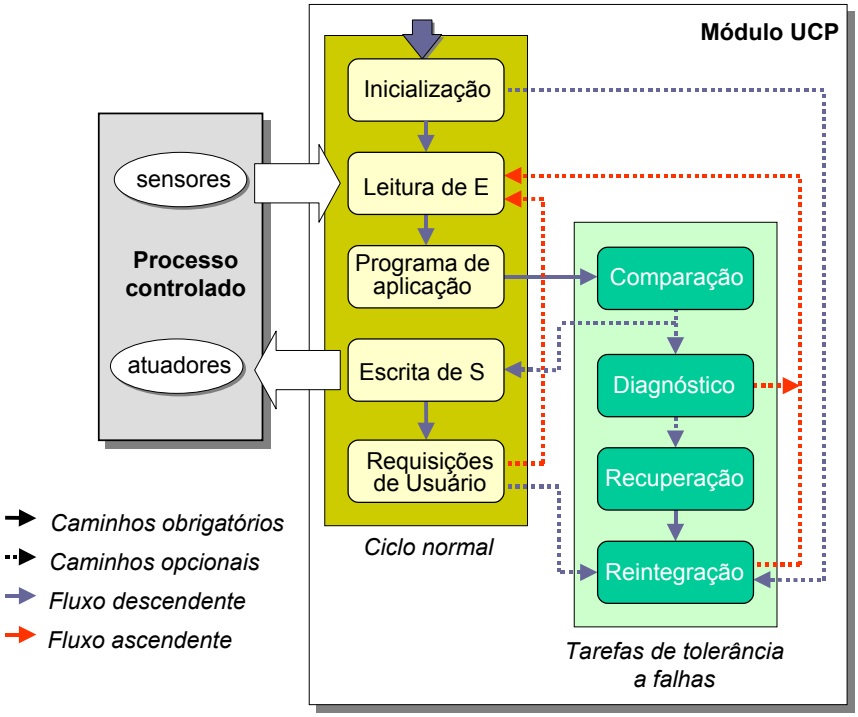


Figura 2. Seqüência de tarefas da UCP

4.2 Requisições de Usuário

As requisições de usuário são fonte potencial de inconsistência entre UCPs se processadas em momentos diferentes. No HARTS, o esquema utilizado para sincronização de módulos de supervisão é similar ao líder seguidor [Brasileiro 1996]. Uma requisição de usuário é enviada pelo computador de supervisão para os dois módulos de supervisão. Assim que o Módulo de Supervisão Líder (MSL) recebe a mensagem, este a ordena e a repassa, juntamente com sua ordem, para o Módulo de Supervisão Seguidor (MSS). Este, responde com uma mensagem de confirmação (*ack*). Desta forma, os dois módulos possuem filas idênticas de requisições ordenadas, prontas para serem enviadas para a UCP. Este processo é ilustrado na fig. 3.

As UCPs, após executarem o programa de controle e a comparação de resultados, verificam se existem requisições de usuários nos módulos de supervisão. Estes, enviam suas respectivas filas de requisições ordenadas para suas respectivas UCPs, as quais verificam a igualdade das mesmas (comunicam-se entre si) e processam as requisições. As respostas das UCPs são enviadas para seus respectivos Módulos de Supervisão (módulo contido no mesmo CLP da UCP). O MSL e MSS comparam as respostas recebidas e, se forem iguais, apenas o MSL as repassa para o computador de supervisão.

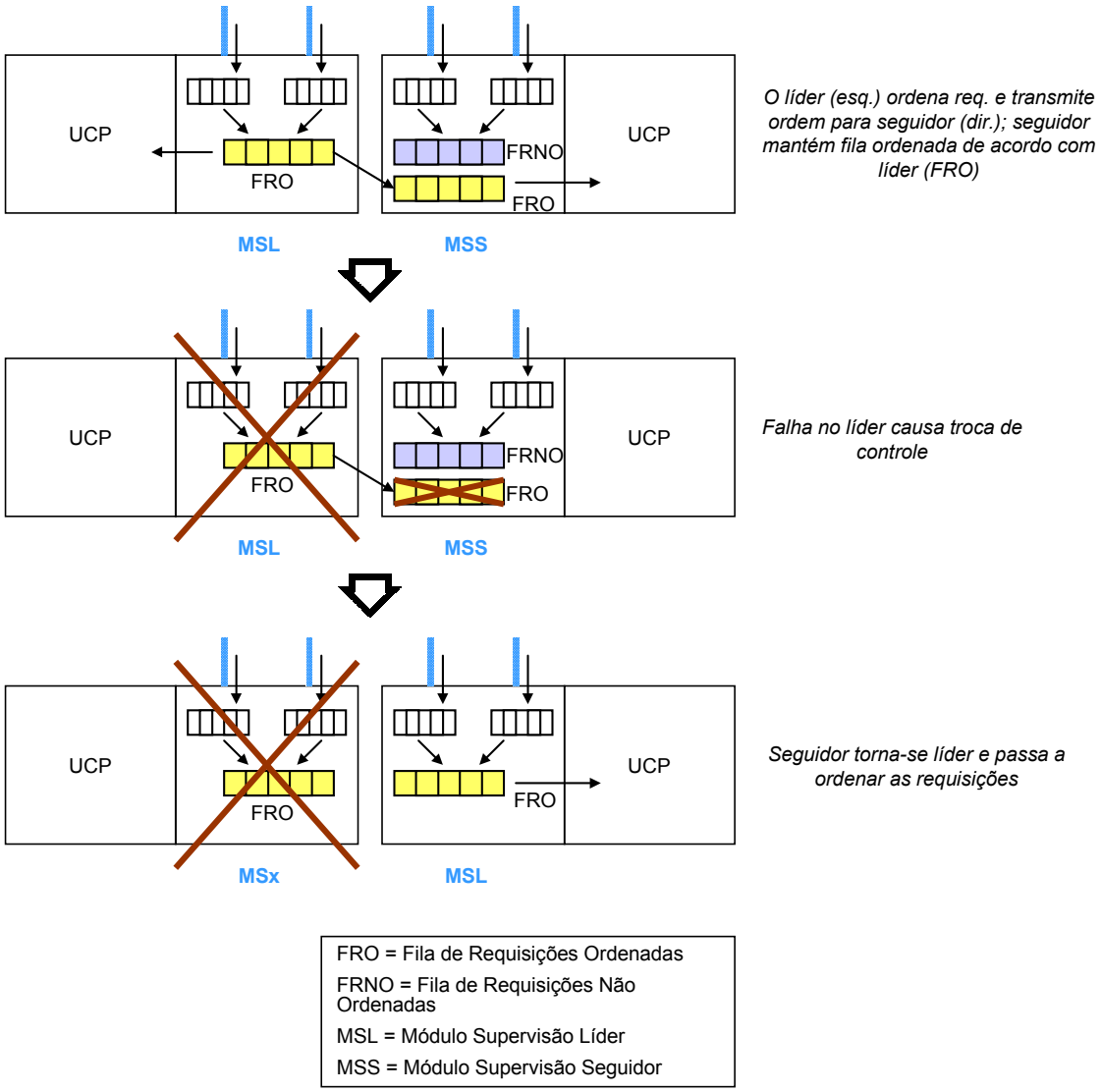


Figura 3. Troca de controle entre MS Líder e MS Seguidor

5. Plataforma

A plataforma do HARTS é composta por equipamentos da série Ponto da empresa Altus Sistemas de Informática, tais como UCPs, Módulos de Campo e Supervisão, mestres e escravos Profibus, FPGAs mestres e escravos de controle do barramento GBL, fontes de alimentação e módulos de E/S. Para implementação da estratégia de tolerância a falhas por software, foi utilizado o sistema operacional de tempo real QNX.

6. Conclusões

Este artigo apresentou o sistema HARTS, composto de uma arquitetura de hardware baseada na duplicação de CLPs e uma estratégia de tolerância a falhas implementada por software sobre o hardware redundante. O HARTS visa controlar processos industriais e plataformas de petróleo com restrições temporais de execução e requisitos de alta disponibilidade e segurança e que empregam computadores embutidos.

A tolerância a falhas do HARTS é alcançada através da detecção de erros nos CLPs por comparação de resultados, diagnóstico e recuperação de falhas, reintegração de CLP e troca de controle. Para o programador da aplicação, os recursos relacionados à tolerância a falhas são transparentes, inclusive a arquitetura com CLPs redundantes. Desta forma, O HARTS oferece altos níveis de disponibilidade e segurança no controle de processos, sem adicionar complexidade à sua programação e utilização, ao mesmo tempo em que permite configuração de acordo com os requisitos de funcionamento da aplicação.

A execução do diagnóstico ocupa alguns ciclos de controle, durante os quais as UCPs tornam-se indisponíveis para o resto do sistema. Nesta situação, os módulos de campo repetem os valores de saída dos ciclos anteriores nos dispositivos de E/S, até que uma ou as duas UCPs tornem-se disponíveis novamente ou que ocorra um *time-out*. Isto pode ocasionar violação da propriedade de tempo real da aplicação, mas não deve causar seu colapso. De acordo com experiência prática [Cunha 2001], eventuais erros nos valores de saída não são dramáticos, desde que os valores subseqüentes sejam corrigidos, observação válida inclusive para alguns sistemas de tempo real rígido (*hard real-time*).

Através deste trabalho, foi possível constatar a viabilidade de utilização de componentes de hardware e software do tipo COTS na implementação de sistemas de controle embutidos de tempo real com requisitos de alta disponibilidade e segurança, desde que devidamente adaptados. O HARTS é voltado para plataformas de petróleo, mas pode ser empregado em aplicações cíclicas que utilizem CLPs e que possam ter algumas saídas repetidas durante o seu funcionamento.

Referências Bibliográficas

[Brasileiro 1996] Braileiro, F.V. et al. Implementing fail-silent nodes for distributed systems. **IEEE Transactions on Computers**, Los Alamitos, v.45, n.11, p.1226-1238, Nov. 1996.

[Cunha 2001] Cunha, J.C. et al. A study of failure models in feedback control systems. In: International Conference on Dependable Systems and Networks, Goteburg, 2001. **Proceedings.** Los Alamitos: IEEE Computer Society, Jul. 2001.

[Jones 1983] Jones, C.T.; Bryan, L.A. **Programmable controllers – concepts and applications**. USA: International Programmable Controls, 1983. 329p.

[Laprie 1998] Laprie, J.C. Dependability of computer systems: from concepts to limits. In: IFIP International Workshop on Dependable Computing and its Applications. Johannesburg, South Africa, 1998. **Proceedings.** [S.l.: s.n.], Jan. 1998.

[Pradhan e Banerjee 1996] Pradhan, D.K.; Banerjee, P. Fault-tolerant multiprocessor and distributed systems: principles. In: PRADHAN, D. K., **Fault-tolerant computer system design**. Upper Saddle River: Prentice Hall, 1996. 550p. cap.3, p.135-235.

[QNX 2003] QNX Developer's Network. Disponível em <http://www.qnx.com/developer/docs> Acesso em: abril de 2003.

[Suri 1995] Suri, N.; Walter, C.J e Hugue, M.M. **Advances in Ultra-Dependable distributed systems**. cap.1. Los Alamitos: IEEE Computer Society, 1995. 467p.

Middleware para Redes de Sensores Sem Fio*

Antonio A.F. Loureiro, Linnyer Beatrys Ruiz, Fernanda Paixão Franciscani, Rainer Ronnie Pereira Couto, José Marcos S. Nogueira

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais

{loureiro, linnyer, fepaixao, rainerpc, jmarcos}@dcc.ufmg.br

Resumo

Na última década, houve um grande avanço tecnológico nas áreas de sensores, circuitos integrados e comunicação sem fio, que levou a criação das redes de sensores sem fio. As novas tecnologias de fabricação e integração reduziram o custo e o tamanho dos nodos sensores que formam estas redes.

As redes de sensores definem novos conceitos e problemas. Alguns, tais como gerenciamento de localização e energia, são fundamentais, independentes da aplicação. Muitas características das redes de sensores, tais como auto-organização, localização, mecanismos de endereçamento e serviços de *binding*, coleta de dados envolvendo problemas de cobertura de área e exposição, topologia dinâmica, fusão de dados, arquitetura da aplicação, mecanismos de segurança e tráfego são desafios em relação aos sistemas distribuídos tradicionais, mas também representam novas oportunidades de pesquisa.

Um *middleware* para RSSFs precisam ser leves, eficientes em energia, prover comunicação assíncrona entre os componentes, prover informações sobre o contexto, permitir escalabilidade, robustez, tratar do tempo e da localização.

Neste tutorial, apresentamos o objetivo, a funcionalidade e as características de um *middleware* construído para redes de sensores sem fio. Também abordamos o uso de mecanismos de tolerância a falhas neste tipo de rede. Com este intuito, na primeira parte do trabalho fazemos uma revisão dos tipos de sistemas distribuídos e os requisitos para construção de um *middleware* para cada um deles. Na segunda parte, apresentamos as redes de sensores sem fio e discutimos o uso de *middleware* e mecanismos de tolerância falhas em tais redes.

*Este trabalho foi financiado parcialmente pelo CNPq.

1 Introdução

Um *middleware* é uma camada de software entre o sistema operacional – incluindo os protocolos de comunicação –, e aplicações distribuídas que interagem entre si através de um suporte à comunicação. *Middleware* é um termo amplamente utilizado para denotar um conjunto de serviços genéricos. De uma forma geral, um *middleware* é uma classe de tecnologias projetada para ajudar a gerenciar a complexidade e heterogeneidade inerentes aos sistemas distribuídos e redes. Pode-se dizer que um *middleware* provê uma abstração de programação em um sistema distribuído. Ao fazer isto, um *middleware* provê aos programadores componentes básicos de mais alto nível que soquetes e outras APIs (*Application Programming Interfaces*), providos pelo sistema operacional. Ou seja, um *middleware* tem como objetivo facilitar a interação entre módulos de software distribuídos. Isto reduz significativamente a carga sobre os programadores de aplicações além de diminuir a chance de erros de programação. Algumas vezes, um *middleware* é informalmente chamado de "encanamento" porque conecta aplicações distribuídas através de canais de comunicação para troca de dados. Dependendo do tipo de aplicação, diferentes componentes e funcionalidades estão presentes num *middleware*.

Um *middleware* pode ser visto como o software que torna um sistema distribuído mais facilmente programável. Assim como é mais difícil programar um computador sem um sistema operacional, também é mais difícil programar um sistema distribuído sem um *middleware*, especialmente quando uma operação que envolve sistemas heterogêneos é necessária [1].

Um *middleware* é projetado para tornar transparente para as aplicações diferentes tipos de heterogeneidade que programadores de sistemas distribuídos devem lidar, como, por exemplo, a heterogeneidade das plataformas computacionais (arquiteturas de computadores, sistemas operacionais, e linguagens de programação) e tecnologias de comunicação de dados. O projeto de um *middleware* também inclui aspectos relacionados com o gerenciamento da qualidade de serviço (QoS), segurança da informação e abstrações de programação para prover transparência com relação à distribuição e heterogeneidade do dado e processamento em relação a aspectos como concorrência, replicação, falhas, mobilidade e localização de entidades usuárias e provedoras de serviços.

Como qualquer outro sistema, o projeto de um *middleware* deve considerar as características do sistema distribuído para o qual está sendo desenvolvido. Neste tutorial, tratamos de questões relacionadas de um *middleware* para RSSFs – Redes de Sensores Sem Fio. Uma RSSF é formada por um conjunto de dispositivos compactos e autônomos, chamados nodos sensores. Os nodos da rede de sensores são distribuídos por uma área, despertam, realizam auto-teste, descobrem sua localização, se organizam, comunicam entre si formando uma rede sem fio *ad hoc* e interagem com o ambiente com o objetivo de coletar dados. Uma rede *ad hoc* é uma rede onde os nodos comunicam diretamente entre si, não dependendo de uma infra-estrutura fixa de comunicação. Os nodos de uma RSSF podem processar localmente os dados coletados e/ou enviar esses dados a um ou mais pontos de acesso da RSSF com uma outra rede, como a Internet. É possível também haver nodos atuadores na rede para controlar variáveis do ambiente.

As redes de sensores sem fio são um tipo especial de sistema distribuído *ad hoc* onde, geralmente, os nodos executam uma tarefa colaborativa e, todo o projeto da rede é guiado fundamentalmente pela aplicação de sensoriamento. As RSSFs podem ser formadas por centenas a milhares de nodos sensores e projetadas para executar suas tarefas sem intervenção humana, isto é, após a deposição dos nodos no ambiente de interesse, a rede não é assistida por operadores.

As redes de sensores possuem outras características que as diferenciam das demais redes *ad hoc*. Em geral, os nodos são descartáveis e a fonte de energia não é recarregável [2]. As falhas não são exceções mas acontecimentos freqüentes já que a energia é limitada e nodos podem ser destruídos. A conectividade também pode não ser permanente em razão das falhas nos nodos, da natureza *ad hoc* da rede e das condições ambientais. Para que o sistema continue operando, mesmo de forma degradada, é necessário empregar mecanismos de tolerância a falhas.

Neste tutorial apresentamos uma revisão sobre sistemas distribuídos e *middleware* e discutimos o objetivo, a funcionalidade e as características de um *middleware* para uma RSSF. Também discutimos a implementação de mecanismos de tolerância a falhas neste tipo de rede. O texto está organizado como segue. A seção 2 trata dos tipos de sistemas distribuídos e seus requisitos não funcionais. Na seção 3 apresentamos alguns projetos de *middleware* desenvolvidos para sistemas distribuídos tradicionais e na seção 4 para ambientes móveis. Na seção 5 apresentamos uma visão geral de rede de sensores e como um *middleware* pode auxiliar na sua configuração, manutenção e comunicação em tais redes. A seção 6 discute esquemas de tolerância a falhas para RSSFs. Finalmente, a seção 7 apresenta algumas conclusões sobre este tutorial.

2 Sistemas Distribuídos

Um sistema distribuído é composto por uma coleção de componentes distribuídos sobre um conjunto de computadores (denominados *hosts*), os quais estão conectados por uma rede de comunicação de dados. Os componentes interagem entre si com o objetivo de trocar informação e/ou utilizar serviços disponíveis no sistema. O objetivo primordial de um *middleware* é facilitar essa interação.

Algumas das diferenças entre um sistema distribuído para ambientes fixos e móveis são os dispositivos, conexões e contexto de execução.

- ***Tipos de dispositivos***: dispositivos para uma rede tradicional são normalmente fixos, ou seja, suas localizações não variam ao longo do tempo. Alguns dos dispositivos usados nessas redes são computadores pessoais (PCs), estações de trabalho e *mainframes*. Dispositivos típicos na rede móvel, como PDAs – Personal Digital Assistants, celulares e *smartcards*, normalmente são pequenos, leves, portáteis, usam energia de bateria e possuem uma interface de comunicação sem fio. Todas estas características permitem a mobilidade desses dispositivos. Outra diferença importante é o poder computacional típico dos dispositivos computacionais das redes fixa

e móvel: os primeiros possuem uma capacidade de processamento, memória, armazenamento, e periféricos bem maior que os segundos, além da restrição da energia dos dispositivos móveis.

- ***Tipos de conexões***: apesar da conexão contínua (permanente) não ser uma realidade em nenhum dos dois ambientes, as quebras de conexões são mais freqüentes no ambiente móvel. No ambiente fixo as eventuais quebras de conexão são ocasionadas por falhas imprevistas ou por razões administrativas. No ambiente móvel, entretanto, quebras de conexão não são exceções; ao contrário, fazem parte do funcionamento normal do sistema. Um usuário móvel pode, a qualquer instante, sair do alcance de comunicação ou experimentar uma queda repentina na largura de banda. Assim, conexões em ambiente móveis são ditas *intermitentes*.

- ***Tipos de contexto de execução***: denomina-se contexto o conjunto de fatores que podem influenciar o comportamento da aplicação, o que inclui tanto características internas do dispositivo quanto do ambiente externo. Em ambiente fixos diz-se que o contexto é estático, já que alterações como mudanças de localização e variação de largura de banda são infreqüentes. No extremo oposto, temos o contexto dinâmico dos ambiente móveis: dispositivos são desconectados freqüentemente, localizações não são constantes e tanto a largura de banda de transmissão quanto a qualidade do sinal podem sofrer alterações abruptas.

Os sistemas distribuídos podem ser divididos em tradicionais, textslad hoc e nômades, cada um com suas necessidades próprias. A seguir, são apresentados a definição e os requisitos básicos para a construção de tais sistemas distribuídos.

2.1 Sistemas Distribuídos Tradicionais

Sistemas distribuídos tradicionais, como mostrados na figura 1, são conjuntos de *hosts* fixos, conectados permanentemente através de canais de comunicação estáticos. Além de uma rede fixa com uma alta largura de banda e disponibilidade (quebras de conexões são eventos raros), assume-se que os *hosts* possuem processadores rápidos e uma grande quantidade de memória.

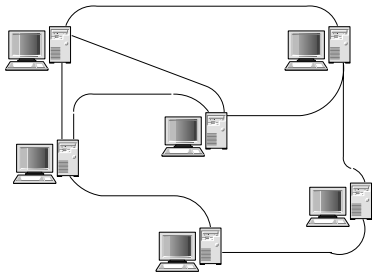


Figura 1: Exemplo de um sistema distribuído tradicional.

Ao projetar aplicações para este tipo de sistema, o projetista deve lidar com os seguintes requisitos não-funcionais:

- ***Escalabilidade***, ou a habilidade do sistema suportar uma carga crescente de trabalho. Em sistemas centralizados (cliente-servidor) a escalabiliade do sistema é ligada diretamente à carga suportada pelo servidor. Sistemas distribuídos podem obter melhor escalabilidade através da distribuição de carga entre diversos *hosts* ou através da replicação de dados. Nas duas técnicas, a escalabilidade é obtida através da disponibilização de cópias de serviços ou de dados. Nesse caso, o desafio é atingir o grau de transparência necessário para que alterações na arquitetura não exijam mudanças radicais no código e nos componentes do sistema.

- ***Extensibilidade***, ou seja, facilidade de inserção de novos módulos ou propriedades no sistema. Em geral, um sistema distribuído sofre alterações durante o seu tempo de vida, seja através da inserção de plataformas mais novas e rápidas ou através da remoção de componentes obsoletos, o que demanda uma forma simples e eficiente de se alterar o sistema.

- ***Heterogeneidade***. Sistemas distribuídos devem lidar com uma diferentes sistemas operacionais, hardware e linguagens de programação. Homogeneidade em sistemas distribuídos pode resultar em uma maior susceptibilidade a falhas e ataques. Um sistema heterogêneo atenua esse fato, além de proporcionar uma melhor escalabilidade, pois tem-se a liberdade de utilizar componentes diferentes para serviços críticos.

- ***Compartilhamento de recursos***. Alguns dispositivos do sistema distribuído são compartilhados entre diversos usuários e aplicações. Alguma forma de gerenciamente deve ser estabelecida para garantir o acesso seguro a esses elementos.

- ***Tolerância a falhas***. Um sistema distribuído deve ser capaz de tratar falhas isoladas sem que isto comprometa o seu funcionamento global. Existem três tipos de falhas: hardware, software e sistema – cada uma exige um tratamento adequado. Falhas de sistema assumem uma importância maior em sistemas distribuídos devido à multiplicidade de componentes e *hosts*. Este assunto será discutido na seção 6.

2.2 Sistemas Distribuídos *Ad hoc*

Um sistema distribuído *ad hoc* é composto por um conjunto de elementos computacionais móveis, conectados através de uma rede sem fio intermitente, cujos canais de comunicação apresentam qualidade váriavel (figura 2). Em redes móveis *ad hoc*, ou MANETs – Mobile Ad hoc Networks –, a comunicação é ponto-a-ponto, o que significa dizer que os elementos computacionais atuam como clientes, servidores e roteadores de mensagens. A diferença básica para um ambiente tradicional é a falta de uma infra-estrutura fixa, ou seja, os *hosts* possuem total liberdade de movimentação. Essa dinâmica permite entrada e saída de nodos na rede de forma simples, porém exige a criação de algoritmos de roteamentos complexos e caros.

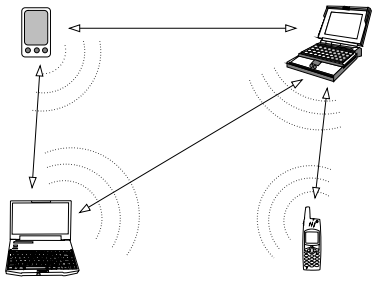


Figura 2: Exemplo de um sistema distribuído *ad hoc*.

Os requisitos não-funcionais para sistemas distribuídos tradicionais ainda são válidos no ambiente móvel *ad hoc*. A escalabilidade se torna um fator crítico: o canal de comunicação é compartilhado entre vários *hosts* e não é confiável. Além disso, os avanços da tecnologia irão permitir uma presença massiva de elementos computacionais móveis, o que agrava ainda mais o problema da escalabilidade. Em ambientes móveis *ad hoc*, a variedade de dispositivos e modos de comunicação é ainda maior que em sistemas tradicionais. A tolerância a falhas é outro fator importante, uma vez que as desconexões e erros na comunicação são eventos comuns em tais redes. O compartilhamento de recursos exige uma atenção especial, pois os canais sem fio são inerentemente inseguros.

Um tipo especial de sistema distribuído *ad hoc* é a rede de sensores sem fio. Uma RSSF tende a ser autônoma e requer um alto grau de cooperação para executar as suas tarefas. Os nodos combinam tecnologia de micro-sensores, computação de baixa potência e comunicação sem fio em um sistema compacto [3]. Um nodo sensor é composto de memória, bateria, processador, transceptor (inclui todo o sistema de transmissão e recepção, óptico ou RF – rádio freqüência) e dispositivo de sensoriamento.

As redes *ad hoc* têm como função básica prover um suporte à comunicação entre elementos computacionais, que individualmente, podem estar executando tarefas distintas. As redes de sensores sem fio tendem a executar uma função colaborativa onde os nodos sensores provêem dados, que são processados por pontos de acesso chamados de nodos sorvedouros (*sink nodes*) ou nodos de monitoração (*monitoring nodes*) ou por estações base. As redes de sensores sem fio compartilham muitos dos desafios encontrados em sistemas distribuídos sem fio *ad hoc* tradicionais. Isto inclui a limitação de energia para cada nodo, a restrição de largura de banda e a utilização de canais, que geralmente, apresentam uma elevada taxa de erros. Contudo, as redes de sensores são formadas por uma quantidade expressiva de nodos que são projetados para operações tipicamente desacompanhadas, apresentam limitação de todos os seus recursos (processador, memória, transceptor, bateria), a fonte de energia não é recarregável e não apresentam uma comunicação tipicamente fim-a-fim, isto é, em geral os nodos não têm identidade (endereço individual). Os dados são nomeados pelos atributos e as aplicações requisitam dados correspondentes a certos atributos, o que torna a arquitetura das redes de sensores sem fio diferente das redes *ad hoc*. Como mencionado, as falhas em rede de sensores sem fio não são exceções, pois um nodo pode deixar a rede por diferentes problemas, principalmente problemas de energia.

Assim, a topologia da rede é dinâmica mesmo sendo os nodos estacionários. As redes de sensores sem fio serão discutidas nas seção 5.

Pelos argumentos apresentados acima, pode-se concluir que um *middleware* projetado para um ambiente tradicional não atende aos requisitos do ambiente móvel *ad hoc*. Logo, é necessário projetar um *middleware* que considere essas características.

2.3 Sistemas Distribuídos Nômades

Dá-se o nome nômade aos sistemas distribuídos que possuem uma infra-estrutura fixa (por exemplo, roteadores e *hosts*) e com elementos terminais dotados de mobilidade. Estes últimos se conectam à infra-estrutura através de estações-base dotadas de interfaces para redes fixas e sem fio. Conceitualmente, são sistemas intermediários entre as redes fixas tradicionais e redes móveis *ad hoc*, e exigem tratamento diferenciado das soluções apresentadas para esses ambientes.

3 *Middleware* para Sistemas Distribuídos Fixos

Um *middleware* para um sistema distribuído pode ser classificado em três categorias principais, de acordo com Capra *et al.* [4]: *middleware* orientado a objetos, *middleware* orientado a mensagens e *middleware* orientado a transações.

3.1 *Middleware* Orientado a Objetos e Baseado em Componentes

Um *middleware* orientado a objetos provê comunicação entre objetos distribuídos, ou seja, um objeto-cliente requisita a execução de uma operação a um objeto-servidor que pode residir em outro *host*. Esta classe de *middleware* evoluiu da Chamada Remota de Procedimento (*RPC –Remote Procedure Calls* [5]), onde a forma básica de interação é síncrona, ou seja, o objeto-cliente que faz o pedido é bloqueado até que o objeto-servidor tenha retornado a resposta. Esta abordagem tem a vantagem de se assemelhar bastante à semântica da chamada local de procedimentos, mas é especialmente sensível a problemas de alta latência na rede. Produtos nesta categoria incluem implementações do OMG CORBA [6], como o IONA Orbix [7] e o Bordland VisiBroker [8], o Modelo de Componentes CORBA (CCM – *CORBA Component Model* [9]), o Microsoft COM [10], Java/RMI [11] e o *Enterprise JavaBeans* [12].

3.2 *Middleware* Orientado a Mensagens

Um *middleware* orientado a mensagens provê a comunicação entre componentes distribuídos através do intercâmbio de mensagens. Um componente-cliente envia uma mensagem através da rede contendo a requisição da execução de um serviço e seus parâmetros a um componente-servidor, que pode responder com uma mensagem-resposta contendo o resultado da execução do serviço. Um *middleware* orientado a mensagens provê comunicação

assíncrona de uma forma bem natural, alcançando desvinculação entre cliente e servidor, como é requisitado por sistemas móveis: o cliente está apto a continuar seu processamento assim que o *middleware* tenha aceitado sua mensagem; o servidor acaba por enviar uma mensagem de resposta e o cliente estará apto a coletá-la em um momento conveniente. A aplicação não tem conhecimento do gerenciamento das mensagens, que é feito de forma totalmente transparente pelo *middleware*.

3.3 *Middleware* Orientado a Transações

Um *middleware* orientado a transações é usado principalmente em arquiteturas em que os componentes são aplicações de banco de dados. Este *middleware* provê transações envolvendo componentes que executam em *hosts* distribuídos. Neste caso, um componente-cliente agrupa várias operações em uma transação que o *middleware* então transporta através da rede para cada componente-servidor de forma transparente tanto para os clientes quanto para os servidores. Este *middleware* provê tanto comunicação síncrona quanto assíncrona entre *hosts* heterogêneos e oferece alta confiabilidade, garantindo a atomicidade das transações desde que os servidores participantes implementem o protocolo *two-phase-commit*. Entretanto, esta solução causa uma sobrecarga se não há a necessidade de se usar transações, o que acontece em vários cenários em ambientes móveis.

4 *Middleware* para Sistemas Móveis Distribuídos

Um *middleware* para sistemas móveis, tanto *ad hoc* quanto nômade, precisa ser leve, prover comunicação assíncrona entre componentes e permitir que os projetistas de aplicações tenham conhecimento sobre o ambiente de execução, como explicado a seguir:

- ***Carga computacional baixa***. Aplicações móveis são executadas em dispositivos com recursos escassos como pouca memória, baixa velocidade do processador e energia limitada. Desta forma, torna-se necessário definir um compromisso entre a carga computacional e os requisitos não-funcionais alcançados pelo *middleware*, pois não é factível executar um *middleware* pesado em tais dispositivos.

- ***Prover comunicação assíncrona***. Os dispositivos móveis tipicamente apresentam conexões intermitentes com a rede. Tipicamente os usuários destes dispositivos se conectam à rede por curtos períodos de tempo, e ainda há os casos da perda de conexão devido à entrada em uma área sem cobertura. Assim, é freqüente o cenário onde o cliente pede um serviço e o servidor provê o serviço mas ambos não estejam conectados ao mesmo tempo, tornando necessária uma forma de comunicação assíncrona.

- ***Prover informações sobre o contexto***. Diferentemente dos sistemas fixos, os sistemas móveis executam em contextos muito dinâmicos. Esta dinamicidade torna

inviável que o projetista da aplicação antecipe todos os contextos possíveis de execução e instrua (a priori) o *middleware* sobre como agir em cada situação. Uma vez que não existe um conhecimento completo que possa ser explorado, o *middleware* não pode tomar decisões de forma transparente para a aplicação e, ao mesmo tempo, garantir a melhor qualidade de serviço. Ao contrário, o *middleware* deve interagir com a aplicação, fazendo-a ciente das mudanças no contexto de execução e ajustando dinamicamente seu próprio comportamento ao usar a informação que a aplicação lhe retorna. Projetar a aplicação ciente do contexto de execução torna mais complexa a tarefa dos programadores de aplicações. Assim é importante a definição de interfaces simples para que as aplicações possam interagir dinamicamente com o *middleware*.

Na seção 3 vimos que um *middleware* para um sistema fixo pode ser classificado em três categorias principais. De acordo com os requisitos apresentados acima, esses tipos apresentam inadequações com relação aos ambientes móveis, como discutido a seguir.

O *middleware* orientado a objetos tem aplicabilidade limitada em ambientes móveis devido à alta carga computacional requerida pela execução destes sistemas, pela forma síncrona de requisição de objetos e pelo princípio de transparência que guiou seu projeto, que impede qualquer tipo de conhecimento sobre o contexto.

O *middleware* orientado a mensagens, por sua vez, requer dispositivos com abundância de recursos, especialmente em termos de quantidade de memória para armazenar as filas de mensagens recebidas mas não processadas ainda. Esta é uma limitação que impede a utilização de produtos como Sun Java Message Queue [13] e IBM MQSeries [14] em dispositivos portáteis. Além disso, não é provida nenhuma forma de conhecimento sobre o ambiente, isto é, as mensagens são manipuladas transparentemente pelo *middleware*, sem nenhum conhecimento por parte da aplicação.

Por fim, tem-se o *middleware* orientado a transações. Nele o protocolo *two-phase-commit*, utilizado para garantir a atomicidade das transações, representa uma sobrecarga se não for necessária a utilização de transações, como é o caso típico de aplicações para ambientes móveis. Novamente, a carga computacional e a transparência deste tipo de *middleware* nos impede de usar produtos como o IBM CICS [15] e o BEA Tuxedo [16] em ambientes móveis, apesar do sucesso de seu uso em ambientes fixos.

Assim, verifica-se que um *middleware* desenvolvido para um sistema distribuído tradicional não pode ser adotado com sucesso em ambientes móveis, uma vez que eles geram uma carga computacional alta, e/ou provêem um paradigma de comunicação síncrona, e/ou foram projetados com base no princípio da transparência. Além disso, devido à grande diferença entre os requisitos dos sistemas fixos e dos sistemas móveis, nem mesmo a adaptação de um *middleware* de sistemas fixos para ambientes móveis se mostra como uma boa solução. Portanto, novos projetos de *middleware* devem ocorrer tendo em vista, os desafios impostos pelas redes móveis (*ad hoc*, redes de sensores sem fio e nômades.

De acordo com Capra *et al.* [4], as soluções existentes de *middleware* para sistemas móveis podem ser agrupadas em três categorias principais: *middleware* reflexivo, *middleware* baseado em espaço de tuplas e *middleware* sensível ao contexto. Aqui propomos uma pequena modificação ao esquema proposto por Capra *et al.* [4]. Assumimos que o terceiro

conjunto, *middleware* sensível ao contexto, também envolve o *middleware* desenvolvido para aplicações ou serviços específicos.

A seguir são apresentados alguns exemplos de *middleware* desenvolvidos para ambientes móveis, exemplificando cada uma das vertente descritas acima.

4.1 *Middleware* Reflexivo Orientado a Objetos

O conceito de reflexão foi inicialmente introduzido por Smith [17] como um princípio que permite a um programa ter acesso a sua interpretação, refletir sobre ela e alterá-la. Este conceito é uma ruptura com a filosofia da "caixa preta", em que os detalhes de implementação são escondidos do usuário da plataforma.

A reflexão é, portanto, o meio através do qual um sistema é capaz de inspecionar e alterar seu comportamento durante a execução e de acordo com regras estabelecidas. Basicamente, um sistema reflexivo é capaz de realizar tanto a auto-inspeção quanto a auto-adaptação. Para conseguir isto, um sistema reflexivo possui uma representação de si mesmo. Esta representação é conectada de forma causal ao objeto representado, o domínio. Assim, qualquer mudança no domínio tem um efeito no sistema e vice-versa [18]. Esta conexão causal entre o sistema e sua representação é implementada pela reificação e absorção.

A reificação é a ação de expor a representação interna de um sistema em termos de entidades programáveis que podem ser manipuladas em tempo de execução. Já a absorção mostra-se como um processo inverso, que consiste em refletir as mudanças realizadas nas entidades reificadas no sistema real [19].

Um sistema reflexivo pode ser divido em duas partes: o nível-base e o meta-nível. O primeiro está relacionado com *o que* o sistema faz, ou seja, suas funcionalidades, enquanto que o segundo trata *como* o sistema faz, lidando com o processamento reflexivo.

As entidades presentes no meta-nível são chamadas de meta-objetos e a interface do meta-nível, isto é, o protocolo de interação com os meta-objetos é normalmente chamado de Protocolo de Meta-Objeto (MOP – *Meta-Object Protocol*). O MOP é que define o estilo de reflexão provido pelo sistema.

A seguir, estão descritos alguns projetos de sistemas desta categoria:

- ***OpenCorba*** [20]: é um *broker* aberto e reflexivo, baseado no modelo CORBA (*Common Object Request Broker Architecture*), que permite ao usuário adaptar dinamicamente a representação e as políticas de execução da aplicação. Através do uso de reflexão, OpenCorba distingue o que um objeto faz (classes) de como o objeto faz (meta-classes). Um protocolo específico permite a alteração, em tempo de execução, de propriedades das classes e meta-classes.

  A utilização de uma linguagem reflexiva e o uso de meta-classes facilita a reusabilidade e provê a flexibilidade necessária para ambientes dinâmicos. Entretanto, os aspectos reflexivos desse sistema têm impacto negativo no seu desempenho já que o custo do tratamento de mensagens entre classes e meta-classes e o controle da reflexão

tornam o OpenCorba inviável para ambientes móveis, onde recursos como memória e processamento são escassos.

- ***Open ORB*** [21]: é uma plataforma de *middleware* altamente configurável e dinâmica. Porém, ao contrário do OpenCorba, Open ORB já foi projetado para atender requisitos de aplicações que envolvem multimídia e mobilidade. A arquitetura Open ORB permite uma configuração personalizada da plataforma, uma vez que os componentes possuem interfaces bem definidas e são estruturados hierarquicamente (componenentes podem ser compostos por outros componentes). Isso é uma vantagem em ambientes móveis, pois um núcleo reduzido do *middleware* pode ser definido eliminando-se elementos desncessários do sistema, atendendo às restrições físicas de memória dos dispositivos portáteis.

- ***DynamicTAO*** [22]: é uma extensão do ORB Corba TAO e que permite a configuração dinâmica da plataforma ORB e das aplicações que são executadas sobre esse ele. Essa reconfiguração engloba componentes que implementam o controle de concorrência, segurança e monitoramento. DynamicTAO também permite a configuração personalizada do *middleware*, possibilitanto a carga e liberação de módulos da memória em tempo de execução. Para permitir a configuração de uma grande coleção de ORBs distribuídos, DynamicTAO também exporta uma meta-interface destinada à criação de agentes móveis de reconfiguração.

4.2 *Middleware* Baseado em Espaço de Tuplas

Como desconexões são eventos comuns em ambientes móveis, a comunicação necessita, intrisicamente, de certo assincronismo e desvinculação, no sentido de que a computação pode ocorrer mesmo que as partes envolvidas estejam desconectadas. O sincronismo implícito de certas soluções de textslmiddleware dificulta, ou mesmo inviabiliza, seu uso em ambientes móveis.

O paradigma da comunicação baseado em espaços de tuplas foi projetado para prover um assincronismo entre processos comunicantes e pode ser aplicado a ambientes móveis e dinâmicos, como *ad hoc*, mesmo que esse não seja o ambiente para o qual foi projetado. Em Linda, um dos precurssores desse modelo de comunicação em ambientes fixos, há uma memória compartilhada globalmente que atua como um repositório de estruturas denominadas *tuplas*. Tuplas são os elementos básicos de comunicação entre processos e são manipuladas através de primititas como **write**, **read** e **take**. O assincronismo da comunicação é caracterizado pela inserção e leitura de tuplas em momentos diferentes. Isto é possível pois tuplas possuem tempo de vida independente dos processos que as geram, assim escrita e leitura estão desvinculadas.

Possuir conhecimento do contexto é uma necessidade comum em aplicações para ambientes móveis. Sistemas de espaços de tuplas dificultam este processo pois suas primitivas e tipos de dados são extremamente básicos. Outra desvantagem desses sistemas é a sincrozinação: uma tupla replicada pode sofrer alterações quando suas unidades retentoras estão desconectadas, o que torna o processo de reconciliação uma tarefa árdua.

A seguir são descritas duas tecnologias implementadas segundo este paradigma:

- ***Lime*** [23]: divide o espaço de tuplas em diversos sub-espaços, definindo regras baseadas em conectividade para o compartilhamento de tuplas. Cada sub-espaço é associado unicamento a uma unidade móvel, que pode tanto ser um agente móvel (mobilidade lógica) ou um *host* móvel (mobililidade física). O modelo Lime retira os detalhes pertinentes à mobilidade do projeto de aplicações, o que é uma vantagem para os desenvolvedores de software, porém uma desvantagem em sistemas que desejam ter uma visão parcial ou completa do contexto. Para lidar com este problema, Lime define um espaço comum e transiente onde estão disponíveis informação sobre o contexto global e insere dados de localização nas tuplas, o que torna o contexto parcialmente visível. Além disso Lime permite inserir ações em resposta a mudanças de contexto no espaço de tuplas. Apesar disso, Lime não permite a adaptação do comportamento do próprio *middleware*.

- ***Tspaces*** [24]: é um *middleware* desenvolvido pela IBM para prover a comunicação, computação e gerenciamento de dados em dispositivos portáteis. TSpaces é uma união entre espaços de tuplas e banco de dados: enquanto o primeiro provê flexibilidade no modelo de comunicação, o segundo lida com a estabilidade, durabilidade e armazenamento e consulta de dados. Sua implementação em Java possui a vantagem da portabilidade. Em TSpaces há uma distinção entre clientes e servidores: os espaços de tuplas existem somente em servidores. Uma vez criado um espaço de tuplas, o cliente realiza a interação através das funções básicas de leitura e escrita.

  TSpaces se diferencia dos outros sistemas por permitir modificações dinâmicas do comportamento do *middleware* e permitir alteração nos tipos de dados e suas operações. Apesar de adequado ao ambiente nômade, TSpaces não é aplicável às redes *ad hoc* devido à sua diferenciação entre clientes e servidores.

4.3 *Middleware* para Aplicações/Serviços Específicos

Alguns projetos de *middleware* foram criados para aplicações específicas e, geralmente, lidam com um problema único e determinado referente à mobilidade. Em geral, tais projetos possuem uma implementação rígida e são adequados para poucos contextos, ou seja, a falta de flexibilidade é compensada por um ganho na eficiência.

A seguir, estão descritos alguns projetos de sistemas desta categoria:

- ***Nexus*** [25]: provê informações de localização às aplicações que necessitam desses dados. O *middleware* trabalha com GPS para ambientes externos e com sensores para ambientes internos. A localização é utilizada pelas aplicações para fornecer dados personalizados para cada usuário.

- ***Odyssey*** [26]: é um *middleware* desenvolvido para dar suporte às aplicações de acesso a informação em ambientes móveis. O modelo assume que as aplicações são

executadas em dispositivos móveis e os dados estão armazenados em servidores remotos, o que o torna ideal para o cenário nômade.

Odissey utiliza um modelo colaborativo de adaptação entre o sistema operacional (elemento que gerencia os recursos) e a aplicação (elemento que determina as políticas de adaptação). Cada aplicação deve fazer um registro no *middleware* para um ou mais recursos monitorados. Assim que uma alteração ocorre em um desses recursos, as aplicações são notificadas e as devidas ações são tomadas.

5 Middleware para Redes de Sensores Sem Fio

Na última década, houve um grande avanço tecnológico nas áreas de sensores, circuitos integrados e comunicação sem fio, que levou a criação das redes de sensores sem fio. As novas tecnologias de fabricação e integração reduziram o custo e o tamanho dos nodos sensores que formam estas redes. A figura 3 apresenta alguns tipos de micro-sensores sem fio resultantes de pesquisas em diversas instituições, como o Smart Dust [27] da *University of California, Berkeley*, WINS (*Wireless Integrated Network Sensors*) [28] da *University of California, Los Angeles* e JPL Sensor Webs [29] do *Jet Propulsion Lab* da NASA.

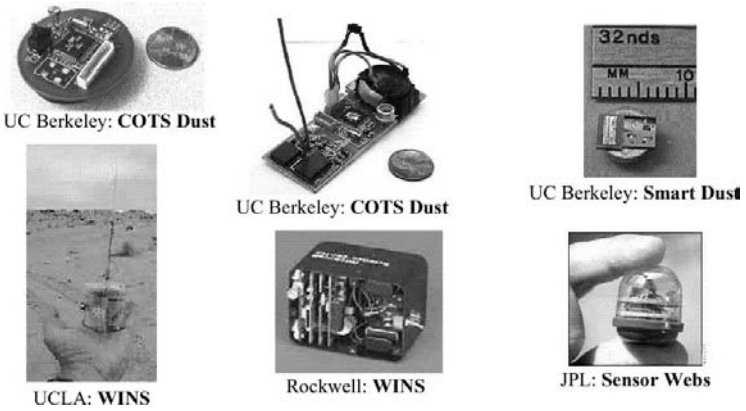


Figura 3: Projetos acadêmicos de nodos sensores.

A redução do tamanho dos nodos sensores restringe a capacidade de seus compontentes (bateria, processador, memória, transceptor, e dispositivo sensor). Já a redução do custo permite que as RSSFs sejam formadas por uma grande quantidade de nodos (centenas a milhares) e utilizadas em diversas aplicações incluindo monitoração ambiental, *smart spaces*, aplicações médicas, monitoração de tráfego, segurança, aplicações militares, monitoração industrial, agricultura de precisão e outras [30, 31, 32, 33, 34]. Esses sistemas prometem revolucionar as aplicações de monitoração, provendo granularidade de dados não realizável por outros meios [2].

Os nodos de uma RSSF são depositados (lançados) em um ambiente, caem, realizam auto-teste, descobrem sua localização, e se organizam formando uma rede *ad hoc*. Em

função do custo reduzido dos componentes e do ambiente onde as redes de sensores são depositadas, os nodos não são facilmente acessíveis para a recarga de suas baterias, intervenção humana e até mesmo para recuperação, isto é, os nodos são considerados descartáveis. Neste tipo de rede, falhas não são exceções [35].

Em sistemas distribuídos tradicionais, as conseqüências das falhas, ou seja o colapso do sistema, a interrupção no fornecimento do serviço e a perda de dados, podem ser evitados pelo uso de técnicas de tolerância a falhas. Essas técnicas devem garantir o funcionamento correto do sistema mesmo na ocorrência de falhas. A implementação de mecanismos de tolerância a falhas em qualquer tipo de sistema sugere a redundância de hardware e/ou software com um custo associado. Contudo, sistemas construídos com componentes autônomos, operando em um ambiente remoto e sem intervenção de técnicos e operadores, não conseguem ser confiáveis pela simples aplicação de tolerância a falhas. Este é o caso das redes de sensores sem fio. Embora não se consiga garantir o funcionamento correto deste tipo de rede, os mecanismos de tolerância a falhas podem ser utilizados para que o sistema continue operando dentro de um determinado nível de qualidade de serviço.

Muitas das funções associadas às RSSFs, inclusive funções de tolerância a falhas, podem ser implementadas em um *middleware*. Nas próximas seções, abordamos estes três assuntos: redes de sensores sem fio, *middleware* para RSSFs e a implementação de mecanismos de tolerância a falhas.

5.1 Rede de Sensores Sem Fio

Um nodo sensor é um elemento computacional com capacidade de processamento, memória, interface de comunicação sem fio, além do dispositivo sensor, como mostrado na figura 4. Os nodos são distribuídos, têm restrições de energia, e devem possuir mecanismos para auto-configuração e adaptação devido a problemas como falhas de comunicação, de energia e variações nas condições ambientais. Cada nodo de uma RSSF pode ser equipado com um ou mais dispositivos sensores, tais como acústico, sísmico, infravermelho, vídeo-câmera, estresse mecânico, calor, temperatura, radiação e pressão.

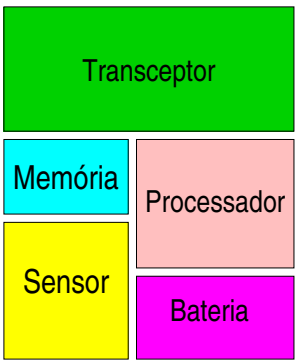


Figura 4: Hardware básico de um nodo sensor

O fluxo de informação em uma RSSF é geralmente unidirecional, isto é, os nodos

disseminam suas informações na rede sempre em direção ao ponto de acesso. O esquema de comunicação entre o nodo fonte e o ponto de acesso é normalmente *multi-hop*, como mostrado na figura 5, a fim de economizar energia.

O ponto de acesso representa o elemento computacional que processa e armazena a informação coletada pela rede. Este é o ponto onde usuários e aplicações podem ter acesso aos dados coletados pelos nodos sensores.

As RSSFs podem ser planas ou hierárquicas, dentre vários outros critérios de classificação. Nas RSSFs planas não existe agrupamento de nodos e nas redes hierárquicas os nodos se organizam em grupos e elegem um líder que recebe os dados coletados pelos nodos membros e os envia ao ponto de acesso usando comunicação *single hop*, em redes homogêneas, ou *multi hop* em redes heterogêneas.

Nas redes plans, os pontos de acesso são, em geral, nodos com maior capacidade de hardware chamados de nodos sorvedouros (*sink nodes*) ou nodos de monitoração (*monitoring nodes*). No caso das redes hierárquicas, o ponto de acesso é uma estação base [36]. Ao contrário do nodo sorvedouro, assume-se que a estação base não têm limitação de energia e capacidade computacional. Os pontos de acesso, sejam eles estações base ou nodos sorvedouros, enviam as informações coletadas pela rede aos observadores podendo utilizar nodos *gateway* para esta finalidade [37, 38, 39].

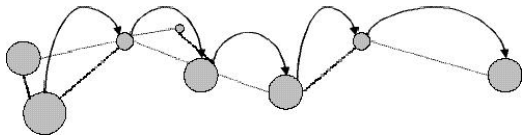


Figura 5: Transmissão Multi hop em RSSFs

O observador é uma entidade (por exemplo, usuário, outra rede ou aplicação) que tem interesse nos dados coletados e informações disseminadas pelo conjunto de sensores que compõe a rede. De acordo com o tipo da aplicação, o observador pode enviar solicitações (consultas), informando seus interesses, e receber respostas geradas por um conjunto total ou parcial dos nodos. Quando apenas alguns nodos participam da resolução de uma consulta, diz-se fidelidade.

O serviço de coleta de dados (monitoração) pode ser contínuo, periódico, realizado a partir da ocorrência de eventos ou a partir de consultas. Depois da coleta, os nodos podem processar os dados e disseminá-los ou não na rede. O serviço de disseminação de dados pode ser diferente do serviço de coleta, isto é, os nodos podem coletar dados continuamente e disseminá-los em intervalos programados. Portanto, uma RSSF é projetada a partir dos objetivos da aplicação para a qual se destina. A partir da definição das características da aplicação e do tipo de ambiente onde a RSSF irá atuar é que se define o tipo de composição (homogênea ou heterogênea), organização (plana ou hierárquica), tipo de coleta (periódica, contínua, reativa, tempo real), tipo de disseminação da informação (contínua, programada, dirigida a eventos, sob demanda) e funções de manutenção da rede [36].

O modo básico de operação de uma RSSFs é significativamente diferente das redes de computadores e sistemas distribuídos tradicionais e *ad hoc*, devido a sua integração com o

mundo físico e também porque a rede opera sem intervenção humana direta e na maioria das aplicações, os nodos realizam a mesma tarefa e colaboram entre si com um objetivo comum.

Uma RSSF tende a ser autônoma e requer um alto grau de cooperação para executar as tarefas definidas para a rede. Isto significa que mecanismos de tolerância a falhas e algoritmos distribuídos tradicionais, como protocolos de comunicação e eleição de líder, devem ser revistos para esse tipo de ambiente antes de serem usados diretamente. O mesmo acontece para o projeto do *middleware*. Na próxima seção, discutimos o projeto do *middleware* para RSSFs.

5.2 O Uso de *Middleware* em Redes de Sensores Sem Fio

O objetivo de uma RSSF é monitorar e, eventualmente controlar um ambiente. O projeto de uma RSSF é dependente da aplicação e das características do ambiente onde essas redes irão trabalhar.

Atualmente, grande parte das pesquisas sobre RSSFs tratam do desenvolvimento de algoritmos e funções que são projetados especificamente para um tipo de rede. Em sua maioria, essas funções e algoritmos são testados usando simuladores e executam diretamente sobre o hardware. A identificação e implementação de primitivas de sistema operacional para este tipo de rede ainda é uma área inicial de pesquisa. Um sistema operacional utilizado na família de nodos sensores Motes é o TinyOS [40]. Trata-se de um sistema operacional (escalonador de eventos) bastante simples. Seu objetivo é reduzir a memória ocupada e o custo do sistema. Suas principais características são: manipulação direta do hardware; não existência de gerenciamento de processos, um processo sendo executado por vez; e inexistência de suporte para memória virtual e alocação dinâmica de memória.

A figura 6 apresenta a arquitetura do TinyOS. Um *middleware* poderia ser construído tanto entre a aplicação e o TinyOS quanto entre a aplicação e o próprio hardware.

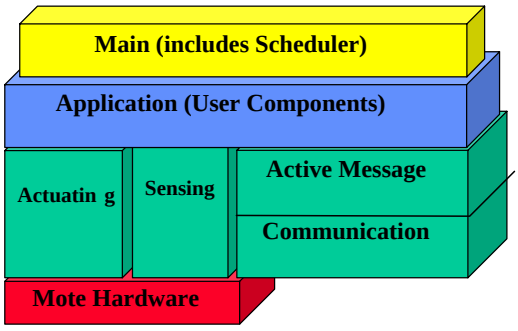


Figura 6: Arquitetura simplificada do TinyOS.

Todos os mecanismos providos por um *middleware* devem considerar os requisitos de sistemas móveis (ver seção 4) e respeitar as características das RSSFs quais sejam, eficiência em energia, robustez e escalabilidade.

Outra propriedade de um *middleware* para RSSFs é imposta pelo princípio do conhecimento da aplicação nos nodos. O *middleware* tradicional é projetado para acomodar uma grande variedade de aplicações sem necessariamente precisar de conhecimento de aplicação. Porém, o *middleware* para uma RSSF deve prover mecanismos para incorporar o conhecimento de aplicação na infra-estrutura e na RSSF [41].

***Middleware* para Configuração dos Elementos de Rede.** Alguns nodos sensores permitem configurar seus componentes (por exemplo, o transcepetor). A configuração realizada diretamente sobre o hardware pode ser uma tarefa complexa e difícil. Um *middleware* construído sob a plataforma dos nodos pode permitir a configuração em função dos requisitos da aplicação até mesmo em tempo de execução. Alguns exemplos de tarefas de configuração executadas com auxílio do *middleware* são: configuração da potência de transmissão do transceptor, configuração do tipo de coleta, calibração do dispositivo sensor, configuração do tipo de disseminação dos dados coletados.

***Middleware* para RSSFs Heterogêneas.** Nas RSSFs heterogêneas é comum a coexistência de tecnologias e plataformas diferentes de hardware, sitemas operacionais, protocolos de rede, linguagens de consulta e mesmo aplicações. Um *middleware* para RSSFs heterogêneas pode se encarregar de compatibilizar os diversos elementos heterogêneos que compõem a rede de maneira a uniformizar métodos de acesso, configuração, formato de dados, tipos de disseminação, tipos de coleta.

***Middleware* para Interação.** Os componentes de uma RSSF (homogênea ou heterogênea) interagem entre si com o objetivo de trocar informação e/ou utilizar serviços uns dos outros. O objetivo primordial de um *middleware* é facilitar essa interação. Um *middleware* pode prover interoperabilidade entre diferentes nodos sensores na mesma rede, assim como entre os observadores e a rede, diferentes redes de sensores e entre uma aplicação de gerenciamento e a RSSF. Um *middleware* pode ser disponibilizado nos elementos de rede (nodos comuns, atuadores, nodos líderes de grupo e nodos *gateways*) ou construídos nos pontos de acesso para tornar transparente ao observador a configuração e a manutenção da rede, assim como a resolução de uma consulta.

***Middleware* para Manutenção.** Assim como a configuração, as funções relacionadas à manutenção também podem ser programadas em um *middleware* com características de reflexão (ver seção 4.1). A partir do conhecimento do estado da rede sob diferentes aspectos (energia, topologia, conectividade, densidade) ou em função de variações nas condições ambientais, o *middleware* altera o comportamento do sistema durante a execução. A capacidade de configurar flexivelmente uma plataforma de *middleware* tem o potencial de tornar mais eficiente o suporte oferecido às aplicações. Um suporte dinâmico de *middleware* permite a manutenção, em tempo de execução do serviço fornecido. O uso da adaptação dinâmica permite que a configuração de uma plataforma de *middleware* seja otimizada

como resposta imediata a variações no estado da rede, variações nas condições ambientais e nos requisitos das consultas dos observadores [41].

***Middleware* para Aplicação Específica.** Um *middleware* pode ser utilizado para dar o suporte à comunicação distribuída no contexto da disseminação de dados, configuração e manutenção dos nodos, configuração e manutenção da rede e dos serviços, assim como podem ser construídos para o contexto da aplicação como, por exemplo, *middleware* para fusão de dados.

Algumas das funções de RSSFs que podem ser implementadas em um *middleware* são:

- ***Recuperação de Informação***. As RSSFs são tipicamente *data centric*, isto é, a consulta não é realizada para um nodo através do seu identificador ou endereço. A consulta é feita para a rede em função de um atributo. Por exemplo, "existe uma temperatura superior a 100 graus?". Este paradigma de comunicação se assemelha a sistemas de recuperação de informação com base no conteúdo, o que é diferente da comunicação tradicional de estilo RPC. A comunicação baseada em evento é mais adequada às características de RSSFs que esquemas de pedido-resposta tradicionais. As consultas também podem utilizar um critério de fidelidade, isto é, selecionar nodos, parâmetros ou algoritmos com os quais resolvem um certo problema de acordo com determinadas restrições de recurso.

- ***Negociação de Contexto***. Um *middleware* pode garantir que as entidades envolvidas em uma rede de sensores sem fio (observadores, nodos de monitoração, nodos comuns e atuadores) trabalhem de forma coordenada, transparente e sincronizada para prover serviços ao usuário final. Assim, conforme o estado da rede se altera em função de um determinado parâmetro, o *middleware* tem a responsabilidade de realizar a manutenção com base no contexto.

- ***Gerenciamento***. Em sistemas tradicionais, cada dispositivo de computação pertence a alguém que é responsável pela configuração, manutenção, e manipulação de erro. Em contraste, os nodos de uma RSSF têm que operar desacompanhados, o que significa que o *middleware* para RSSFs tem que prover novos níveis de apoio para configuração automática, manutenção e manipulação de erro.

- ***Tempo Real***. Desde que as RSSFs processam dados do mundo real, os conceitos de tempo físico e localização têm um papel muito mais importante que em sistemas tradicional. Tempo e localização de eventos do mundo real são elementos chaves por fundir leituras de sensor individuais e obter um alto-nível do sensoriamento resultante. Algumas áreas de aplicação podem exigir características de tempo real. Então, o apoio para tempo e gerenciamento de localização deve ser firmemente integrado em uma infra-estrutura de *middleware* para RSSFs.

- ***Manutenção da Área de Cobertura***. Em muitas aplicações a área de cobertura de uma RSSF pode variar devido a vários fatores, entre eles a mobilidade dos nodos

e a presença de fontes de energia limitadas e não recarregáveis. A mobilidade pode ampliar ou modificar os limites da área de cobertura inicialmente definida. No caso da fonte de energia limitada, nodos que tornam-se indisponíveis e não são substituídos reduzem a área de cobertura inicial e podem influenciar na precisão do resultado. A cobertura é considerada uma medida de qualidade de serviço [34]. Na literatura, a área cobertura é definida sobre diferentes pontos de vista, entre eles o melhor e o pior caso. No pior caso, tentativas são realizadas para quantificar a qualidade de serviço procurando áreas de baixa observabilidade dos nodos e detectando regiões de monitoração descobertas. No melhor caso, o objetivo é encontrar áreas de alta observabilidade dos nodos e identificar as melhores regiões de monitoração. A cobertura também pode ser representada por definições estocásticas como, por exemplo por funções de distribuição uniforme, distribuição Gaussiana e distribuição de Poisson.

- ***Tolerância a Falhas***. Nas RSSFs os nodos podem falhar, principalmente, devido a variações nas condições do ambiente físico e podem perder sua funcionalidade devido à falta de energia. Considerando que em muitas aplicações a troca ou reposição de nodos é difícil, a RSSF deve implementar mecanismos de tolerância a falhas que possam garantir o funcionamento da rede. Na próxima seção discutimos os mecanismos de tolerância a falhas aplicados a RSSFs.

6 Tolerância a Falhas em RSSFs

Segundo Weber [42], o objetivo de tolerância a falhas é alcançar dependabilidade. O termo dependabilidade[1] indica a qualidade do serviço fornecido por um dado sistema e a confiança depositada no serviço fornecido. Os principais atributos de dependabilidade são confiabilidade, disponibilidade, segurança de funcionamento (*safety*), segurança (*security*), mantenabilidade, testabilidade e comprometimento do desempenho (*performability*).

A seguir, discutimos cada um destes atributos com relação às RSSFs.

6.1 Confiabilidade em RSSFs

As RSSFs são projetadas para atenderem a uma série de requisitos. Um dos requisitos importantes é que elas sejam confiáveis tanto quanto possível e apesar do meio em que estão inseridas. A confiabilidade é definida como sendo a capacidade que um sistema tem em responder a uma dada especificação dentro de condições definidas e durante um certo tempo de funcionamento. A confiabilidade é uma característica importante para as RSSFs porque o reparo é praticamente impossível neste tipo de rede.

[1]Este termo é uma tradução literal do termo em inglês *dependability*.

6.2 Disponibilidade em RSSFs

Os nodos de uma RSSFs são depositados (muitas vezes lançados sobre uma área) em *sleep mode* para economizar energia, despertam, realizam auto-teste, descobrem sua localização (utilizando alguma técnica como GPS – *Global Position System* ou em colaboração com os vizinhos) e se auto-organizam formando uma rede *ad hoc*. Durante a deposição, alguns nodos podem ser destruídos (total ou parcialmente). Também durante a descoberta de localização e formação da rede, alguns nodos podem estar isolados dos demais. Dependendo da distribuição e do estado dos nodos, podem surgir áreas descobertas de sensoriamento e comunicação. Após o período de formação da rede, os nodos iniciam as atividades de sensoriamento, processamento local e disseminação da informação. Estas atividades consomem energia. No tempo, os nodos deixam a rede (temporariamente ou definitivamente) por diferentes motivos mas principalmente por problemas de energia. O nível de energia de um nodo define seu aspecto operacional. Já o aspecto operacional da rede é definido em termos de seu mapa de energia, isto é, se todos os nodos que compõem a rede estão operacionais em um determinado instante. Portanto, a disponibilidade de uma RSSF é a probabilidade do sistema estar operacional num determinado instante de tempo.

6.3 Segurança em RSSFs

Uma RSSF está sujeita a diferentes tipos de ameaças: internas, externas, acidentais ou maliciosas. Quando falamos de ameaças internas e acidentais, dizemos que a rede pode apresentar outros problemas além daqueles relacionados com o nível de energia. Por exemplo, os sensores dos nodos podem estar descalibrados e produzindo dados que corrompem o serviço de coleta. Um exemplo para ameaças externas e acidentais diz respeito a variações nas condições ambientais (por exemplo, ruído) que causam interferências nas comunicações ocasionando perdas de pacotes. Nos casos de ameaças internas e externas acidentais, os nodos devem descontinuar suas funções para evitar problemas e deseperdício de energia. Então, segurança de funcionamento (*safety*) é a probabilidade do sistema estar operacional, e executar sua função corretamente, ou descontinuar suas funções de forma a não provocar dano a outros sistemas que dele dependam. Quando tratamos de ameaças maliciosas, definimos segurança (*security*) como proteção contra impedimento de serviço (DoS – *Denial of Service*), falhas maliciosas, visando privacidade, autenticidade, integridade e irrepudiabilidade dos dados.

6.4 Comprometimento do Desempenho em RSSFs

O comprometimento do desempenho está relacionado à queda de desempenho provocado por falhas, onde o sistema continua a operar, mas degradado em desempenho. Em certas aplicações de RSSFs, um atraso na entrega de uma informação pode torná-la inválida ou inconsistente. Em outras aplicações, a qualidade do serviço exige que todos os nodos da rede participem da coleta de dados. Se houverem regiões descobertas em função da falhas de alguns nodos, haverá comprometimento do desempenho da rede.

6.5 Mantenabilidade em RSSFs

A mantenabilidade significa a facilidade de realizar a manutenção do sistema, ou seja, a probabilidade que um sistema com defeitos seja restaurado a um estado operacional dentro de um período determinado. Restauração envolve a localização do problema, o reparo físico e a colocação em operação. No caso das RSSFs, o reparo físico pode ser difícil ou mesmo impossível de ser realizado. A manutenção pode utilizar nodos redundantes já existentes na rede ou determinar uma deposição incremental de nodos que venham a substituir o nodo com defeito. Um serviço de manutenção de área de cobertura para RSSFs foi definido em [35]. A partir do mapa de topologia da rede, o serviço de manutenção verifica qual o conjunto mínimo de nodos que cobrem a rede. Então, o serviço coloca os nodos que não fazem parte do conjunto mínimo em "sleep mode". Quando um nodo do conjunto mínimo apresenta nível de energia mínimo, o serviço de manutenção ativa o nodo redundante para cobrir a área. Este é um esquema de tolerância a falhas que exige nodos incrementais e consumo de energia para realizar suas funções.

6.6 Testabilidade em RSSFs

A testabilidade é a capacidade de testar certos atributos internos ao sistema ou facilidade de realizar certos testes. Quanto maior a testabilidade, melhor a mantenabilidade, e por conseqüência menor o tempo que o sistema não estará disponível devido a reparos. A testabilidade em RSSFs é um assunto polêmico porque qualquer operação realizada sobre a rede consome energia.

6.7 Projetando Dependabilidade

A construção de um sistema *dependable* requer esforços em todas as fases de desenvolvimento do sistema. Existem passos a serem realizados na fase de projeto, na fase de implementação e em tempo de execução, assim como durante a manutenção e aprimoramento. Na fase de de projeto, pode-se aumentar a dependabilidade de um sistema através de técnicas de prevenção de falhas. Na fase de implementação, a dependabilidade pode ser aumentada através de técnicas de remoção de falhas. Finalmente, em tempo de execução são necessárias técnicas de tolerância a falhas e de evasão de falhas.

Uma RSSF construída com a capacidade de tolerar falhas conseguirá manter-se operando, talvez de uma forma degradada, na presença de falhas. Para que um sistema seja tolerante a falhas, ele deve ser capaz de detectar, diagnosticar e confinar as falhas, além de compensá-las e de recuperar-se delas.

O termo "evasão de falhas" diz respeito a atitude de um sistema que, quando se desvia do comportamento normal, mesmo se o comportamento continua atendendo as especificações do sistema, se reconfigura para reduzir a demanda sobre um componente com grande potencial de falhar. Este é o caso quando nas RSSFs nodos em situação crítica de energia estão participando de algum serviço.

O grau de tolerância a falhas que um sistema requer pode ser especificado quantitativamente ou qualitativamente. Quando a meta de dependabilidade é quantitativa, ela é normalmente expressa como a taxa de falhas máxima permitida (por exemplo, 10% dos sensores indisponíveis por sub-área). O problema desta forma de expressão é a dificuldade em se saber quando ela foi atingida.

Alternativamente, pode-se usar a especificação qualitativa dos requisitos de dependabilidade. Seguem algumas especificações típicas:

- ***Fail-safe***: projete o sistema de forma que, quando ele sofrer um certo número de falhas, ele falhe da forma segura. Por exemplo, quando um certo número de nodos falham e deixam uma área de monitoração descoberta, o observador deve ser informado de que os dados não correspondem a área de cobertura total.

- ***Fail-op***: projete o sistema de forma que, quando ele sofrer um certo número de falhas, ele ainda forneça um subconjunto do comportamento especificado para ele. Por exemplo, ainda que existam áreas descobertas, a rede é capaz de fornecer o serviço com relação às áreas ainda operacionais.

- ***No single point of failure***: projete o sistema de forma que a falha de um único componente, seja ele qual for, não faça o sistema falhar. Normalmente, em tais sistemas, o componente falho pode ser substituído ou consertado antes que outra falha ocorra. No caso das RSSFs os nodos não podem ser consertados visto que são descartáveis. Pode-se, então, utilizar esquemas de nodos *back-up* para substituir os nodos que falharam.

- ***Consistência***: projete o sistema de forma que toda a informação entregue pelo sistema seja equivalente à informação que seria entregue por uma instância de um sistema não falho. Este atributo é difícil de ser construído em RSSFs devido a sua integração com o mundo físico e também porque a rede opera sem intervenção humana direta e na maioria das aplicações, os nodos realizam a mesma tarefa e colaboram entre si com um objetivo comum.

Como visto, a aplicação dos atributos de dependabilidade em RSSFS representa novos desafios e grande oportunidade de pesquisa. A seguir discutimos os níveis de tolerância a falhas em RSSFs.

6.8 Níveis de Tolerância a Falhas em RSSFs

As técnicas de tolerância a falhas devem garantir o funcionamento correto do sistema mesmo na ocorrência de falhas. Todos os mecanismos de tolerância a falhas são baseadas em redundância, exigindo componentes adicionais ou algoritmos especiais.

Heimerdinger e Weinstock [43] definem três níveis em que a tolerância a falhas pode ser aplicada quais sejam, hardware, software, e sistemas. A seguir, discutimos estas definições considerando as RSSFs.

Tolerância a Falhas de Hardware. Tradicionalmente, a tolerância a falhas tem sido usada para compensar falhas nos recursos computacionais (hardware). Ao gerenciar recursos extras de hardware, o subsistema do elemento computacional aumenta sua habilidade de continuar sua operação. Medidas de tolerância a falhas de hardware incluem replicação de processadores, memória adicional, fontes de energia redundantes, e comunicação redundante. A comunicação redundante pode ser conseguida com certo custo em RSSFs mas a replicação de processadores, memória adicional e fonte de energia no mesmo elemento computacional é impossível para a atual tecnologia das RSSFs. A tolerância a falha do hardware para RSSFs pode ser realizada utilizando redundância de nodos. Além da tolerância a falhas, o uso da redundância em RSSFs pode ser motivado por um outro fator: precisão. De qualquer forma, a redundância sugere uma alta densidade de nodos, isto é, aumenta o número de sensores por área. Se a densidade for alta podem surgir áreas de interseção de coleta, informações redundantes e interferência nas comunicações, além do desperdício de energia. No caso de uma alta densidade, alguns nodos podem ser retirados administrativamente de serviço e transformados em nodos *backup*. Quando os nodos ativos começarem a deixar a rede por problemas de energia, os nodos *backup* são ativados. No caso de uma baixa densidade, a área de cobertura da rede pode estar comprometida, exigindo a ativação ou outro lançamento de nodos. A alta densidade causa a redundância que pode servir como mecanismo de precisão e tolerância a falhas mas que representa um maior consumo de energia.

Tolerância a Falhas de Software. Um segundo nível de tolerância a falhas reconhece que um hardware tolerante a falhas não consegue, por si só, garantir alta disponibilidade. Assim, também é importante estruturar o software dos elementos computacionais para compensar falhas decorrentes de erros transitórios, entrada de dados não prevista ou de erros de projeto. Esta é a tolerância a falhas de software. Uma RSSF é um sistema distribuído reativo, já que interage continuamente com o ambiente[2]. A parte de controle é normalmente modelada por uma máquina de estados finitos estendida e comunicante (CEFSM – *Communicating Extended Finite State Machine*). A máquina é estendida no sentido que predicados podem ser associados a transições e comunicante porque interage com outras máquinas. Este tipo de sistema é difícil de validar, seja através de verificação formal, simulação ou teste. Sendo assim, atenção especial deve ser dada ao projeto desses sistemas, que além da correção deve se preocupar com a eficiência da solução proposta [2].

Tolerância a Falhas do Sistema. Um terceiro nível reconhece que disponibilizar hardware e software tolerante a falhas em RSSFs não garante a alta disponibilidade. As RSSFs operam em diferentes ambientes e estão sujeitas a todo tipo de ameaça (interna, externa, acidental e maliciosa). É necessário prover funções que compensem falhas que não são baseadas no elemento computacional. Esta é a tolerância a falhas do sistema. Por exemplo,

[2]De forma genérica, um ambiente é tudo que se encontra fora do espaço de endereçamento de um processo, incluindo aí o sistema operacional, outros processos no próprio nodo, canal de comunicação e ambiente físico que gera os "eventos" que devem ser processados.

uma RSSF sofrendo de interferência nas comunicações ou no sensoriamento em função das condições ambientais.

Finalmente e importante que as medidas de tolerância a falhas sejam compatíveis em todos os níves.

7 Conclusão

Os requisitos não-funcionais como, escalabilidade, extensibilidade, heterogeneidade, compartilhamento de recursos, tolerância a falhas para sistemas móveis *ad hoc* distribuídos são válidos para RSSFs. Contudo, em RSSFs estes requisitos adquirem outras características em função dos componentes, tipo de operação e ambiente onde tais redes trabalham.

Um *middleware* para RSSFs precisam ser leves, eficientes em energia, prover comunicação assíncrona entre os componentes, prover informações sobre o contexto, permitir escalabilidade, robustez, tratar do tempo e da localização.

As redes de sensores definem novos conceitos e problemas. Alguns, tais como gerenciamento de localização e energia são assuntos fundamentais, em que muitas aplicações confiam para obter a informação necessária [34]. Muitas características das redes de sensores, tais como auto-organização, localização, mecanismos de endereçamento e serviços de *binding*, coleta de dados envolvendo problemas de cobertura de área e exposição, topologia dinâmica, fusão de dados, arquitetura da aplicação, mecanismos de segurança e tráfego são desafios em relação aos sistemas distribuídos tradicionais, mas também representam novas oportunidades de pesquisa.

Referências

[1] D. Bakken. Middleware. *Encyclopedia of Distributed Computing*, 2002.

[2] A. A. Loureiro, Linnyer B. Ruiz, Raquel A. Mini, and José Marcos S. Nogueira. Rede de sensores sem fio. Minicurso do Simpósio Brasileiro de Redes de Computadores. Natal, RN, Brasil, 2003.

[3] Praveen Rentala, Ravi Musunuri, Shashidhar Gandham, and Udit Saxena. Survey on sensor network. Submitted as per the requirements of Mobile Computing (CS6392) Course, Disponível via <http://www.citeseer.nj.nec.com/479874.html> Acessado em 25 de julho de 2002.

[4] L. Capra, W. Emmerich, and C. Mascolo. Middleware for Mobile Computing (A Survey). In *Advanced Lectures on Networking - Networking 2002 Tutorials*, pages 20–58, Pisa, Itália, Maio 2002. Springer Verlag.

[5] W. Stevens. *UNIX Network Programming*. Prentice Hall, 1997.

[6] A. Pope. *The Corba Reference Guide: Understanding de Common Object Request Broker Architecture*. Addison-Wesley, 1998.

[7] S. Baker. *Corba Distributed Objects: Using Orbix*. Addison-Wesley, 1997.

[8] V. Natarajan, S. Reich, and B. Vasudevan. *Programming With Visiobroker: A Developer's Guide to Visiobroker for Java*. John Wiley & Sons, 2000.

[9] OMG. CORBA Componet Model. http://www.omg.org/cgi-bin/doc?orbos/97-06-12.

[10] D. Rogerson. *Inside COM*. Microsft Press, 1997.

[11] E. Pitt and K. McNiff. *Java RMI: The Remote Method Invocation Guide*. Addison-Wesley, 2001.

[12] R. Monson-Haefel. *Enterprise Javabeans*. O'Reilly & Associates, 2000.

[13] R. Monson-Haefel, D. Chappell, and M. Loukides. *Java Message Service*. O'Reilly & Associates, 2000.

[14] IBM Redbooks. *MQSeries Version 5.1 Administration and Programming Examples*. IBM Corporation, 1999.

[15] E. Hudders. *CICS A Guide to Internal Structure*. Wiley, 1994.

[16] C. Hall. *Building Client/Server Applications Using TUXEDO*. Wiley, 1996.

[17] B. Smith. *Reflection and Semantics in a Procedural Programming Language*. PhD thesis, MIT, 1982.

[18] H. Duran and G. Blair. A Resource Management Framework for Adaptive Middleware. In *3th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC'2k)*, Newport Beach, California, EUA, Março 2000.

[19] F. Costa and F. Kon. Novas Tecnologias de Middleware: Rumo à Flexbilização e ao Dinamismo. In *Anais do 20o Simpósio Brasileiro de Redes de Computadores*, 2002.

[20] T. Ledoux. OpenCorba: A Reflective Open Broker. *Reflection'99*, Springer-Verlag, LNCS 1616:197–214, Julho 1999.

[21] G. Blair, G. Coulson, A. Andersen, M. Clarke, F. Costa, H. Duran, R. Moreira, N. Parlavantzas, and K. Saikoski. The Design and Implementation of Open ORB version 2. *IEEE Distributed System Online Journal*, 2(6), 2001.

[22] F. Kon, B. Gill, M. Anand, R. Campbell, and M. Mickunas. Secure Dynamic Reconfiguration of Scalable CORBA System with Mobile Agents. In *Proceedings of the IEEE Joint Symposium on Agent Systems and Applications/Mobile Agents (ASA/MA'2000)*, Setembro 2000.

[23] G. Picco, A. Murphy, and G. Roman. LIME: Linda Meets Mobility. In *Proceedings of the 21st international conference on Software engineering*, pages 368–377. IEEE Computer Society Press, 1999.

[24] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. Tspaces. *IBM Systems Journal*, Agosto 1998.

[25] D. Fritsch, D. Klinec, and S. Volz. NEXUS Positioning and Data Management Concepts for Loacation Aware Applications. In *Proceedings of the 2nd International Symposium on Telegeoprocessing*, pages 171–184, Nice-Sophia-Antipolis, France, 2000.

[26] M. Satyanarayanan. Mobile Information Access. *IEEE Personal Communications*, 3(1), 1996.

[27] Smart dust. autonomous sensing and communication in a cubic millimeter. *Disponível via <http://robotics.eecs.berkeley.edu/Epister/SmartDust>. Acessado em 11 de abril de 2002.*

[28] WINS Wireless integrated network sensors. Disponível via <http://www.janet.ucla.edu/wins/>. Acessado em 12 de março de 2002.

[29] JPL Sensor Webs. http://sensorwebs.jpl.nasa.gov/, 2002.

[30] D. Estrin, R. Govindan, and J. Heidemann. Special issue on embedding the internet. *Communications of ACM*, 5(43), May 2000.

[31] Stephanie Lindsey, Cauligi Raghavendra, and Krishna Sivalingam. Data gathering in sensor networks using the energy delay metric. Disponível via <http://www.citeseer.nj.nec.com/461425.html>. Acessado em 12 de maio de 2002.

[32] Deborah Estrin, Ramesh Govindan, John S. Heidemann, and Satish Kumar. Next century challenges: Scalable coordination in sensor networks. Proceedings of the fifth annual ACM/IEEE International Conference on Mobile computing and networking. pages 263-270, August 1999, Seattle, WA USA.

[33] B. R. Badrinath, M. Srivastava, K. Mills, J. Scholtz, and K. Sollins. Special issue on smart spaces and environments. *IEEE Personal Communications*, Oct 2000.

[34] Seapahn Meguerdichian, Farinaz Koushanfar, Miodrag Potkonjak, and Mani B. Srivastava. Coverage problems in wireless ad hoc sensor networks. In *INFOCOM*, pages 1380–1387, 2001.

[35] Linnyer Beatrys Ruiz, José M. S. Nogueira, and Antonio A. Loureiro. Manna: A management architecture for wireless sensor networks. *IEEE Communications Magazine*, 41(2):116–125, February 2003.

[36] Linnyer Beatrys Ruiz, José Marcos S. Nogueira, and Antonio A. Loureiro. Functional and information models for wireless sensors networks. In *Actes GRES03 - Colloque Francophonic sur La Gestion de Reseaux et Service*, 2003.

[37] Loren Schwiebert, Sandeep K. S. Gupta, and Jennifer Weinmann. Research challenges in wireless networks of biomedical sensors. In *Mobile Computing and Networking*, pages 151–165, 2001.

[38] Mani B. Srivastava, Richard R. Muntz, and Miodrag Potkonjak. Smart kindergarten: sensor-based wireless networks for smart developmental problem-solving enviroments. pages 132–138. Mobile Computing and Networking, 2001.

[39] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. In Proceedings of the 2001 ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean, April 2001.

[40] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for network sensors. In *ASPLOS*, 2000. http://webs.cs.berkeley.edu/tos/.

[41] Kay Römer, Oliver Kasten, and Friedmann Mattern. Middleware challenges for wireless sensor networks. *Mobile Computing and Communications Review*, 6(2), 2002.

[42] T. Weber, I. Jansch-Pôrto, and R. Weber. Um roteiro para exploração dos conceitos básicos de tolerância a falhas. http://www.inf.ufrgs.br/ taisy/, 2003.

[43] W. Heimerdinger and C. Weinstock. A conceptual framework for system fault tolerance. Technical Report CMU/SEI-92-TR-33, Carnegie Mellon University, Pittsburgh, PA, 1992.

Tolerância a Faltas Adaptativa em um Modelo de Componentes

Fábio Favarim[1], Joni Fraga[1], Frank Siqueira[2]

[1]Departamento de Automação e Sistemas – DAS
[2]Departamento de Informática e Estatística – INE
Universidade Federal de Santa Catarina – UFSC
Florianópolis – SC – Brasil – CEP 88040-900
{fabio,fraga}@das.ufsc.br, frank@inf.ufsc.br

Abstract. *This paper presents a fault tolerant model based on components for building distributed applications. The TFA-CCM model allows that quality of service (QoS) requirements be used to select the configuration of replicated services during execution time, employing a set of components that deal with the non-functional aspects of the application. The characteristics of this model and its implementation are described along this paper.*

Resumo. *Este artigo apresenta um modelo de tolerância a faltas baseado em componentes para a construção de aplicações distribuídas. O modelo TFA-CCM permite que requisitos de qualidade de serviço (QoS) guiem a seleção da configuração de serviços replicados em tempo de execução, utilizando um conjunto de componentes que tratam dos aspectos não-funcionais da aplicação. As características deste modelo e a sua implementação são descritos ao longo deste artigo.*

1. Introdução

Sistemas de software atuais estão cada vez mais distribuídos e operam em ambientes dinâmicos como a Internet. Aplicações distribuídas com requisitos de tolerância a faltas são difíceis de serem construídas e mantidas, quando consideradas a complexidade e as características destes ambientes. A tolerância a faltas deve ser construída fazendo uso de suportes de *middleware* e fornecendo mecanismos que permitam a adaptação de técnicas de redundâncias às mudanças do sistema.

Modelos de desenvolvimento de software baseado em componentes distribuídos, tais como *Enterprise JavaBean*s (EJB), desenvolvido pela *Sun Microsystems* [Sun, 2001], Microsoft *.NET* [Microsoft, 2001] e *CORBA Component Model* (CCM) [OMG, 2002], estão sendo usados e tem mostrado melhora no processo de desenvolvimento e manutenção de software. Porém, tais tecnologias fornecem suporte limitado para tolerância a faltas, geralmente através de mecanismos de persistência de dados.

Neste trabalho é usada *tolerância a faltas adaptativa*, que permite que uma aplicação distribuída baseada em componentes tenha um comportamento esperado mesmo diante de mudanças no seu ambiente computacional, como falhas em componentes e em sítios (*hosts*). Diferentes níveis de confiabilidade e de disponibilidade podem ser fornecidos com diferentes configurações de replicações, alocando somente os recursos necessários para obter os requisitos desejados.

O modelo TFA-CCM - *Tolerância a Faltas Adaptativa no Modelo de Componentes CORBA*, fornece suporte a tolerância a faltas adaptativa totalmente transparente à aplicação sem a necessidade de mudanças no modelo de componentes e na sua implementação. O TFA-CCM é composto por componentes de software que são responsáveis

por implementar técnicas de tolerância a faltas, definindo e controlando o comportamento de um serviço replicado.

O modelo integra mecanismos para especificação de requisitos de QoS que guiam a seleção da configuração de serviços replicados. Deste modo, diferentes níveis de QoS podem ser especificados, visando atender diferentes requisitos de tolerância a faltas. O TFA-CCM tem mecanismos de detecção de falhas parciais que identificam a necessidade de adaptação e mecanismos para efetivar mudanças no sistema visando atender os requisitos de QoS.

Este artigo está organizado da seguinte forma: a seção 2 apresenta uma definição de componentes de software e descreve o modelo de componentes CORBA (CCM); a seção 3 faz uma pequena introdução sobre tolerância a faltas adaptativa. A seção 4 apresenta o TFA-CCM. Na seção 5 são apresentados os aspectos de implementação. A seção 6 discute trabalhos relacionados e na seção 7 o artigo é concluído.

2. Componentes de Software

A abordagem de programação baseada em *componentes de software* está fundamentada na composição de programas a partir de componentes pré-existentes. Segundo [Szypersky, 1998], "Componente é uma unidade de composição com interfaces contratualmente especificadas e apenas com dependências de contexto explícitas. Um componente pode ser instalado isoladamente e estar sujeito à composição por terceiros". O uso de componentes permite a implementação e a manutenção de programas distribuídos de maneira eficiente. Além disso, o custo de processo de desenvolvimento pode ser reduzido de forma significativa devido ao reuso de código.

A especificação de um componente é normalmente publicada separadamente de seu código fonte por meio da especificação de suas interfaces, que são os pontos de acesso aos serviços do componente. Interfaces separam a especificação dos componentes da sua implementação, não permitindo que os usuários conheçam os detalhes de implementação do componente.

2.1 CORBA *Component Model* (CCM)

O conceito de componentes CCM tem como base a extensão do modelo de objetos distribuídos do CORBA. As fases de modelagem, programação, empacotamento, implantação e execução de componentes, previstas nas especificações do CCM, organizam objetos em componentes, aportando à programação distribuída do CORBA todas as características atribuídas anteriormente à programação por componentes. As interfaces dos componentes são especificadas em CORBA IDL (*Interface Description Language*), que a partir da versão 3.0 passou a permitir a definição de componentes.

Componentes CORBA possuem atributos e portas de comunicação. Atributos são propriedades configuráveis do componente e têm como propósito à configuração do componente. As portas de comunicação são pontos de conexão entre os componentes. São definidos quatro tipos de portas [OMG, 2002], ilustradas na Figura 1.

- Facetas (*Facets*): são as interfaces IDL através das quais um componente oferece seus serviços aos seus clientes.
- Receptáculos (*Receptacles*): permitem ao componente invocar os serviços acessíveis através da interface IDL de outros componentes.
- Produtores de Eventos (*Event Sources*): são interfaces que emitem eventos de

um tipo específico para um ou mais consumidores de eventos.

- Consumidores de Eventos (*Event Sinks*): são as interfaces através das quais um componente é notificado da ocorrência de eventos de um determinado tipo.

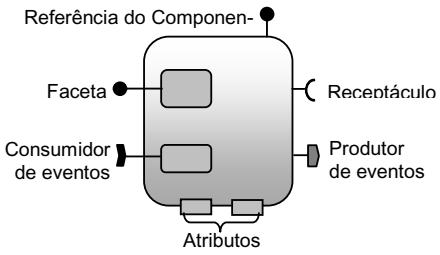


Figura 1 – Componente CORBA

Os componentes CORBA são executados em *containers*. O *container* é o responsável por fornecer um ambiente de execução para um componente, permitindo acesso transparente a serviços comuns do CORBA (aspectos não funcionais). Em outras palavras, o *container*, torna possível separar os aspectos funcionais do componente (implementação da funcionalidade do componente) dos aspectos não-funcionais (interação com o suporte e seus serviços). Com isto, a programação de aplicações se torna mais simples, pois o desenvolvedor se preocupa apenas com a parte funcional da aplicação.

Componentes são empacotados para que sejam posteriormente distribuídos e implantados. Um pacote agrupa uma ou mais implementações de um componente e um conjunto de descritores. Os descritores são escritos em XML [W3C, 1998], e especificam os arquivos binários necessários para implantar os componentes nas diferentes plataformas, assim como os serviços CORBA utilizados pelos componentes e as características de cada componente, como a definição das portas.

3. Tolerância a Faltas Adaptativa

Segundo Kim e Lawrance [1990], tolerância a faltas adaptativa é conseguida com mecanismos que satisfaçam requisitos de tolerância a faltas variáveis dinamicamente através da utilização eficiente (e adaptativa) de uma quantidade limitada e variável de recursos de processamento redundantes. Pode-se definir tolerância a faltas adaptativa como a propriedade que permite um sistema mudar a tolerância a faltas do sistema através da adaptação a mudanças no ambiente computacional ou nas políticas de tolerância a faltas.

Se considerarmos as características dinâmicas dos sistemas distribuídos atuais que ganham em escala dia a dia, a noção fixa e pré-estabelecida na coordenação de modelos de redundâncias os torna ineficientes. Sistemas distribuídos em larga escala, como os construídos com os recursos da Internet, podem se beneficiar de políticas adaptativas. Mecanismos de *tolerância a faltas adaptativa* fazem uso de políticas adaptativas na gerência de redundâncias, de maneira a se manter eficaz mesmo diante de mudanças freqüentes do seu ambiente computacional. A tolerância a faltas adaptativa pode envolver a troca de algoritmos, em tempo de execução, para atender às mudanças do ambiente [Chen et al., 2001].

4.O Modelo TFA-CCM

O TFA-CCM tira proveito da flexibilidade propiciada pela programação baseada na tecnologia de componentes. Neste projeto escolhemos o uso do CCM devido à sua rica semântica de comunicação (vários tipos de portas) e à possibilidade de usá-lo em ambi-

entes heterogêneos, nos quais diferentes linguagens, *middleware* e sistemas operacionais diferentes podem coexistir.

O TFA-CCM permite ao usuário de uma aplicação especificar requisitos de qualidade de serviço (QoS), definindo níveis desejados de disponibilidade e de desempenho do serviço, e o suporte de execução define uma configuração e emprega uma técnica de replicação de modo que os requisitos especificados sejam atendidos. O suporte monitora a ocorrência de falhas na aplicação e adapta seu comportamento de modo a manter o nível de QoS especificado pelo usuário.

A Figura 2 ilustra os diferentes componentes que compõem o TFA-CCM, em uma possível configuração. Essencialmente, estes componentes não-funcionais fazem a configuração e o monitoramento da aplicação, replicando componentes e usando uma técnica de replicação, implementada pelo *coordenador de replicação* de modo a alcançar os níveis de confiabilidade desejados. No exemplo mostrado na Figura 2, o componente localizado no sítio 2 (o servidor primário) é replicado no sítio 3 (servidor *backup*), e o estado dessas réplicas é sincronizado através dos coordenadores de replicação usando a técnica de replicação passiva[1]. A técnica de replicação pode ser facilmente trocada em tempo de execução através da substituição do coordenador de replicação por outro componente que implementa uma técnica de replicação diferente.

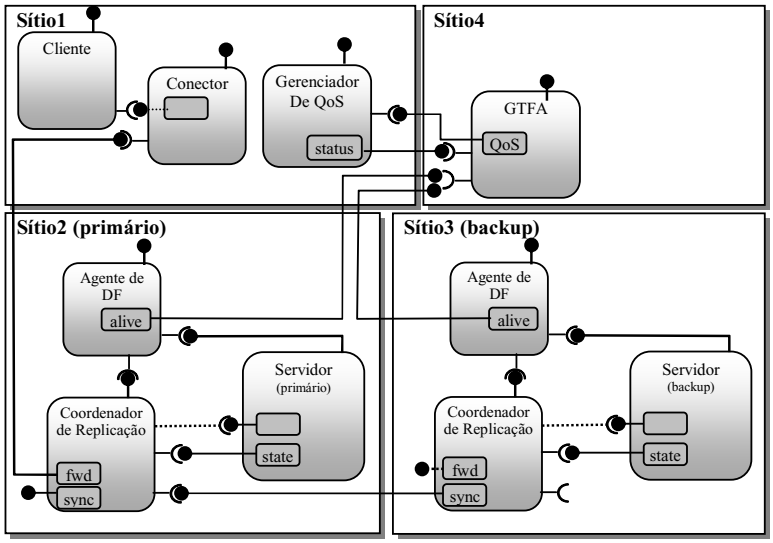


Figura 2 - Visão geral do TFA-CCM

O cliente obtém acesso aos serviços do servidor através do componente *conector*, que tem a função de tornar transparente para o cliente as possíveis mudanças de configuração da aplicação. Por exemplo, a troca de réplicas, onde a réplica *backup* substituiria a primária na configuração, não afeta o comportamento do cliente. Na falha da réplica primária, o conector é redirecionado para a réplica *backup* do servidor.

Para cada componente com requisitos de tolerância a faltas existe um *gerenciador de tolerância a faltas adaptativa* (GTFA), que define a configuração atual baseado nos requisitos de QoS exigidos pela aplicação e na freqüência de falhas parciais nos componentes do sistema. A reconfiguração da aplicação é iniciada quando dados moni-

[1] Na replicação passiva [Powell, 1991], após um envio de resultados ao cliente, a réplica primária deve sincronizar as secundárias (réplicas *backups*) ao seu estado de processamento, enviando uma cópia de seu estado às mesmas.

torados pelo GTFA mostram que a configuração atual não é mais apropriada para manter os níveis de QoS especificados, o que pode ocorrer devido à falta ou excesso de recursos (réplicas) ou pelo uso de uma técnica de replicação inadequada.

O monitoramento das falhas é feito através de *agentes de detecção de falhas* (agentes de DF, na Figura 2**)**. Cada réplica do componente é associada em sua máquina com um agente de DF, que é o responsável pelo envio dos dados de monitoramento ao GTFA. No modelo são previstos dois níveis de detecção de falhas: nível de sítio – no caso do agente de DF parar de responder – e nível de componente – quando o componente para de responder as chamadas do agente de DF.

O *gerenciador de QoS* permite que o usuário especifique seus requisitos de QoS. Através do gerenciador de QoS, diferentes níveis de QoS relacionados a tolerância a faltas podem ser especificados. Esses requisitos de QoS são repassados ao GTFA, que os interpreta e define uma configuração adequada para a aplicação, como o número de réplicas, a técnica de replicação a ser utilizada, entre outros. O GFTA tenta com os recursos disponíveis manter o nível de QoS requisitado. O GFTA volta a atuar na configuração do sistema sempre que uma mudança no sistema leva ao não-atendimento do nível selecionando. Os requisitos podem ser alterados a qualquer momento, em tempo de execução, através do gerenciador de QoS. Qualquer mudança nos requisitos de QoS é informada ao GTFA para que este se encarregue de implementar as mudanças necessárias de configuração. O gerenciador de QoS também é usado pelo GTFA para passar informações de configuração ao usuário, como por exemplo a técnica de replicação utilizada, a localização dos componentes replicados, o número de réplicas sendo utilizadas pelo TFA-CCM e estatísticas sobre a freqüência de falhas na aplicação.

5. Implementação do Modelo TFA-CCM

A implementação dos componentes do TFA-CCM tem como plataforma de desenvolvimento o OpenCCM versão 0.2 [Marvie et al., 2002], que é uma implementação parcial da especificação do CCM. O OpenCCM é totalmente desenvolvido em Java e pode ser usado com três diferentes ORBs. O ORB usado neste trabalho foi o ORBacus, versão 4.0.5 [Iona Technologies, 2002]. O OpenCCM foi escolhido por ter sido a primeira implementação de código aberto disponível da especificação CCM.

A implementação do TFA-CCM fornece três diferentes coordenadores de replicação: um primeiro que implementa a técnica de replicação passiva, um segundo que implementa a técnica de replicação semi-ativa[2] e um terceiro que é um coordenador "vazio", o qual não implementa nenhuma técnica de replicação. Este último é usado quando se deseja apenas requisitos de disponibilidade, onde informações de estado não sejam necessárias. A técnica de replicação ativa não foi implementada, pois requer mecanismos de comunicação de grupo, que não são fornecidos pelo ORB utilizado.

O componente coordenador de replicação tem duas facetas: `sync` e `fwd`. A faceta `sync` é usada pelos coordenadores de replicação passiva e semi-ativa para sincronização do estado das réplicas. A faceta `fwd` é implementada por todos os coordenadores de replicação, para que as requisições dos clientes sejam repassadas do conector para o coordenador de replicação. Além desse uso, a faceta `fwd` é usada pelo coordenador que im-

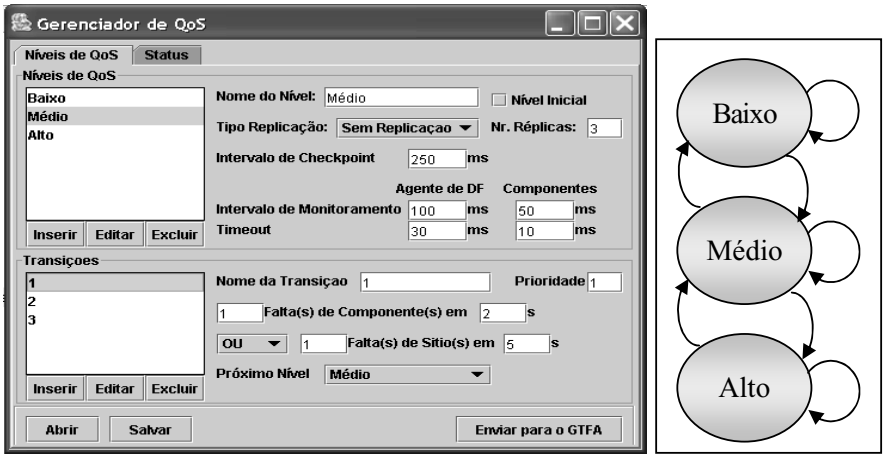
[2] Na replicação semi-ativa (ou replicação *leader/followers*) [Powell, 1991] as requisições são difundidas entre todas as réplicas (seguidoras) pela réplica líder (primária), mas somente a líder envia os resultados ao cliente.

plementa a técnica de replicação semi-ativa, para repassar as requisições recebidas dos clientes para as réplicas seguidoras. Como o coordenador não sabe antecipadamente qual a interface IDL (porta de comunicação) ao qual está associado, é usado o mecanismo de invocação dinâmica do CORBA (DII), o qual permite descobrir em tempo de execução como fazer chamadas aos métodos daquela interface.

De modo a permitir a atualização do estado de suas réplicas, o componente deve fornecer os métodos para acesso ao seu estado. Portanto, estes componentes devem implementar a faceta `state`, que provê os métodos para recuperação (`get_state`) e atualização (`set_state`) do estado do componente.

Para fazer a conexão entre componentes é necessário que as portas de interconexão sejam do mesmo tipo. De modo a tornar o conector genérico (independente do tipo de porta de comunicação), foi utilizada a interface de esqueleto dinâmico (DSI) do CORBA. Esta invocação dinâmica está representada na Figura 2 por uma linha tracejada. Quando uma requisição chega ao conector, este simplesmente a repassa para o coordenador de replicação primário através de uma conexão com a faceta `fwd` deste.

O GTFA armazena em uma estrutura de dados a visão global de todos os componentes implantados no sistema. Essa estrutura contém a referência de todos os componentes, da localização de cada componente, o tipo de replicação que está sendo utilizado, entre outras informações. Estes dados permitem que as reconfigurações necessárias no sistema sejam efetuadas, como por exemplo interligar as portas dos componentes, remover componentes de algum sítio, etc. A falha do GTFA e a conseqüente perda destes dados compromete todo o funcionamento do sistema. Para evitar isso, as informações de estado do GTFA são gravadas em um meio de armazenamento persistente.



(a) (b)

Figura 3 – Interface gráfica do Gerenciador de QoS (a) e os níveis de QoS (b)

O monitoramento de componentes é realizado periodicamente pelo GTFA em intervalos definidos pelo usuário através do gerenciador de QoS. O GTFA invoca periodicamente o método `is_alive` do agente de DF, o qual responde enviando o estado atual dos componentes que ele monitora. Se o agente de DF não responde, uma falha de sítio é assumida e a aplicação pode ser reconfigurada pelo GTFA. Falhas em componentes são detectadas pelo agente de DF através de chamadas ao método `_non_existent` (da classe `CORBA::Object`, que é implicitamente herdada por todos os componentes) e

informadas ao GTFA na próxima chamada do método `is_alive` do agente de DF. A reconfiguração da aplicação também pode ser realizada neste caso, dependendo dos níveis de QoS definidos pelo usuário através da interface gráfica do gerenciador de QoS, mostrada na Figura 3a. Através do uso desta interface, o usuário pode definir os níveis de QoS e as condições que fazem com que as transições entre níveis aconteçam, formando assim, um máquina de estado como mostrado na Figura 3b.

6. Trabalhos Relacionados

A arquitetura AQuA (*Adaptive Quality of Service Availability*) [Cukier et al., 1998] visa fornecer tolerância a faltas adaptativa para aplicações distribuídas. AQuA permite que os programadores de aplicações especifiquem os níveis de confiabilidade desejados, que são alcançados através da configuração do sistema em função da disponibilidade de recursos e de acordo com as falhas ocorridas. AQuA utiliza o QuO [Zinky, 1997] para especificar os requisitos de QoS em nível de aplicação, o gerenciador de confiabilidade do Proteus [Sabnis et al. 1998] para configurar o sistema em resposta a falhas e aos requisitos de disponibilidade. O Ensemble [Hayden 1998] é usado no AQuA para fornecer serviços de comunicação de grupo. Apesar de possuírem mecanismos de especificação de QoS com funcionalidade equivalente, enquanto QuO e AQuA exigem que os requisitos de QoS sejam definidos em tempo de projeto, o TFA-CCM permite que os requisitos de QoS sejam modificados em tempo de execução, tirando proveito da flexibilidade propiciada pela utilização de componentes de software.

Chamaleon [Bagchi et al., 1998] é uma infra-estrutura adaptativa, que permite que diferentes níveis de requisitos de disponibilidade sejam fornecidos através de ARMORs – objetos móveis adaptativos e reconfiguráveis que visam prover os requisitos exigidos pela aplicação. O *Chamaleon* pode usar seletivamente combinações de ARMORs a fim de prover diferentes níveis de disponibilidade, que podem ser introduzidos de maneira incremental no sistema. Apesar de empregar mecanismos de composição semelhantes aos existentes em modelos de componentes, o Chamaleon não segue um modelo de componentes padrão como o CCM ou o EJB.

Em [Marangozova e Hagimont, 2002] é apresentada uma abordagem para replicação de componentes CORBA. Nessa abordagem são usados objetos de interceptação, responsáveis por capturar as requisições feitas para o componente de modo a disparar as ações necessárias para o gerenciamento da replicação. Esses objetos de interceptação possuem a mesma interface do componente que será replicado, o que implica que toda a vez que se desejar replicar uma nova aplicação é necessário implementar um novo objeto de interceptação com a mesma interface do componente de aplicação. No modelo TFA-CCM, é empregado um conector genérico, que independe da interface do componente replicado e não precisa ser modificado para ser utilizado em diferentes aplicações.

7. Conclusão

Este artigo apresenta uma visão geral do modelo TFA-CCM, o qual oferece mecanismos flexíveis para a construção de software baseado em componentes CORBA com requisitos de tolerância a faltas. O modelo TFA-CCM adapta a configuração do sistema levando em conta os requisitos de QoS associados ao componente e as faltas que ocorrem no sistema. Qualquer componente CORBA pode fazer uso do TFA-CCM para oferecer requisitos de tolerância a faltas.

De modo a prover a tolerância a faltas adaptativa, o TFA-CCM foi construído a

partir de um conjunto de componentes de software, que combinados fornecem requisitos de tolerância a faltas para aplicações baseadas em componentes. Apesar de técnicas adaptativas também serem empregadas em vários outros trabalhos para prover requisitos de tolerância a faltas, estes trabalhos não tiram proveito da flexibilidade provida pela utilização de componentes de software.

Pretende-se aprimorar a implementação do TFA-CCM, com a sua evolução para novas implementações da especificação do CCM. Outra possibilidade é fornecer o suporte de tolerância a faltas adaptativa através do *container*, uma vez que este é responsável por fornecer acesso aos serviços não funcionais utilizados por um componente. Além disso, pretendemos avaliar o TFA-CCM através do seu uso em aplicações com requisitos de tolerância a faltas.

Referências Bibliográficas

Bagchi, S. et al. (1998) The Chameleon Infrastructure for Adaptive, Software Implemented Fault Tolerance. In: *17th IEEE Symposium on Reliable Distributed Systems*. p. 261–267, West Lafayette, Indiana. IEEE Computer Society.

Chen, W. K., Hiltunen, M. A. e Schlichting, R. D. (2001). Constructing Adaptive Software in Distributed Systems. In *21st International Conference on Distributed Computing Systems*. p. 635-643. Phoenix. AZ. IEEE Computer Society.

Cukier, M. et al. (1998). AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects. In *17th IEEE Symposium on Reliable Distributed Systems*. p. 245-253, West Lafayette, Indiana. IEEE Computer Society.

Hayden, M. G. (1998). *The Ensemble System*. PhD thesis, Cornell University.

Marangozova, V. e Hagimont, D. (2002). An Infrastructure for CORBA Component Replication. In 1st *IFIP/ACM Working Conference on Component Deployment*. p.222-232, Berlin, Alemanha.

Marvie, R., Merle, P., e Vadet, M. (2002). *The OpenCCM Plataform*. http://corbaweb.lifl.fr/OpenCCM/

Iona Technologies (2001). *ORBacus for C++ and Java,* version 4.0.5.

Kim, K.H., Lawrence, T.(1990).Adaptive Fault Tolerance: Issues and Approaches. In *2nd IEEE Workshop on Future Trends of Distributed Computing Systems.* p. 38-46, Cairo, Egypt. IEEE Computer Society.

Microsoft (2001). *Overview of the .NET Framework*. MSDN Library White Paper.

OMG (2002). *CORBA Components*. OMG Document formal/02-06-65.

Powell, D. (1991). *Delta-4 Architecture Guide*. Esprit II P2252, Delta-4 Phase 3.

Sabnis, C. et al. (1998). Proteus: A Flexible Infrastructure to Implement Adaptive Fault Tolerance in AQuA. In *7th IFIP International Working Conference on Dependable Computing for Critical Applications*. p. 137–156. San Jose, CA, USA

Sun Microsystems, v. (2001). *Enterprise JavaBeans Specification*. v2.0.

Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*. ACM Press/Addison-Wesley Publishing Co.

W3C (1998). *eXtensible Markup Language (XML)* v1.0. World Wide Web Consortium.

Zinky, J. A., Bakken, D. E. e Schantz, R. E. (1997). Architectural Support for Quality of Service for CORBA Objects. *Theory e Practice of Object Systems*, v.3, n.1, p.53–73.

Uma Implementação Tolerante a Faltas e Transparente da Plataforma J2EE

André Andrade Costa, Francisco Vilar Brasileiro*

Universidade Federal da Campina Grande
Coordenação de Pós-Graduação em Informática
Av. Aprígio Veloso, s/n, Bodocongó
58109-970, Campina Grande, PB, Brasil
Tel: (+55) 83 310 1123 Fax: (+55) 83 310 1124

andrec@infonet.com.br,fubica@dsc.ufcg.edu.br

Abstract. *The use of developing platforms to support the implementation of distributed applications has become a trend. These platforms provide a number of specialised services that help programmers to focous on the bussiness logic of the applications they develop, instead of wasting precious time with the implementation of infra-structure services. J2EE (Java 2 Enterprise Edition) is a platform backed up by SUN Microsystems that has lately gain a lot of attention. Unfortunately, the J2EE specification does not provide any support for fault tolerance, a non-fucntional requirement more and more necessary for distributed applications. Developers of such applications must themselves provide the necessary mechanisms to fulfil the requirements of the applications. Alternatively, they can use implementations of the platform that are themselves fault-tolerant. In this paper we present the design and implementation of such a platform. Unlike other implementations available, our implementation provides a solution that is highly reliable and totally transparent to the application.*

Resumo. *Um grande número de aplicações distribuídas tem seu projeto e implementação sustentados por plataformas de desenvolvimento. Estas plataformas provêm uma série de serviços especializados, permitindo assim que os programadores possam se concentrar mais nas regras de negócio das aplicações que desenvolvem. Atualmente a plataforma J2EE (Java 2 Enterprise Edition) da SUN Microsystems é uma das mais populares para este fim. Infelizmente tolerância a faltas, um requisito não funcional cada vez mais presente nas aplicações, não é diretamente suportada pela especificação J2EE. Aplicações desenvolvidas sobre essa plataforma devem, elas mesmas, implementar os mecanismos para tolerância a faltas requeridos, ou usar implementações da plataforma que possuam características de tolerância a faltas. Neste artigo nós apresentamos o projeto e a implementação de um servidor de aplicações J2EE que implementa esses mecanismos. Diferentemente de outras soluções disponíveis, o nosso sistema provê alta confiabilidade de forma totalmente transparente para as aplicações.*

1 Introdução

Devido ao aumento da complexidade dos sistemas de informação, a tendência atual de desenvolvimento tem feito amplo uso de arquiteturas baseadas em componentes servidores. Nesse modelo de desenvolvimento, além de se perceber uma melhor reutilização de software, o desenvolvimento

*Este trabalho é parcialmente financiado pelo CNPq (processo 300.646/96).

torna-se bastante simplificado uma vez que os serviços mais complexos requeridos pelo sistema estão implementados no ambiente de execução. Atualmente, uma das mais promissoras tecnologias de componentes servidores é a especificação *Java 2, Enterprise Edition* (J2EE), criada pela SUN Microsystems em parceria com um número de empresas [4]. A especificação J2EE reúne outras especificações como *Java Servlets*, *Enterprise JavaBeans* (EJB), *Java Naming and Directory Interface* (JNDI), *Java Remote Method Invocation* (RMI), *Java Database Connectivity* (JDBC), *Java Transaction API* (JTA), *Java Transaction Service* (JTS), etc, para oferecer uma maneira transparente de desenvolvimento baseado em componentes servidores.

Entretanto, os serviços da plataforma J2EE não são especificados de forma a dar suporte à implementação de aplicações com requisitos de confiança no funcionamento. Assim, os mecanismos para tolerância a faltas podem ser implementados de duas maneiras: pela própria aplicação, ou pelos serviços que compõem a plataforma J2EE. A primeira alternativa, além de tornar o desenvolvimento da aplicação mais complexo, exige que todas as aplicações incluam, normalmente de forma *ad hoc*, esses mecanismos ao seu código. A segunda alternativa é mais adequada, pois libera o desenvolvedor da tarefa de implementar os mecanismos para tolerância a faltas. Além disso, dependendo da concepção dos servidores que implementam os serviços, pode permitir que os mecanismos para tolerância a faltas executem de forma transparente para as aplicações.

Neste artigo nós apresentamos o projeto de um servidor J2EE no qual os mecanismos para tolerância a faltas são utilizados pelas aplicações de forma totalmente transparente. Diferentemente das outras soluções existentes, nossa abordagem utiliza replicação ativa. O modelo de replicação ativa implementado é totalmente transparente para o desenvolvedor, de forma que basta instalar os componentes da aplicação nas instâncias do servidor para que cada réplica possa processar as requisições da camada cliente. Como as réplicas estarão sincronizadas, qualquer uma das respostas emitidas pelas réplicas poderá ser utilizada, o que eliminará qualquer "janela de vulnerabilidade". Por oferecer suporte a tolerância a faltas totalmente transparente, a plataforma também permite que aplicações legadas executem com maior robustez, sem qualquer modificação no código fonte original. A nossa implementação se baseia em software aberto (o servidor de aplicações JBoss[1] e a ferramenta de comunicação em grupo JChannel[2]) e se consitui em uma alternativa às soluções proprietárias existentes.

O restante desse artigo está organizado da seguinte forma. A Seção 2 apresenta de forma abreviada a plataforma J2EE, enfatizando algumas vunelrabilidades da mesma. Na Seção 3 são discutidos os mecanismos para tolerância a faltas usados em nosso projeto. A Seção 4 apresenta a nossa implementação (maiores detalhes sobre a implementação podem ser encontrados em [5]). A Seção 5 conclui o artigo com nossos comentários finais[3].

2 A Plataforma J2EE

A plataforma J2EE foi projetada para oferecer um padrão de desenvolvimento e um ambiente de execução de aplicações distribuídas baseadas em componentes. Através de um *Servidor de Aplicações* que implemente os serviços dessa plataforma, pode-se fazer uso de tecnologias como transações distribuídas e objetos remotos de maneira simples e transparente.

A seguir discutiremos alguns dos principais serviços oferecidos pela plataforma, colocando enfâse nas possíveis vulnerabilidades dos mesmos.

[1] Disponível em http://www.jboss.org/.

[2] Disponível em http://www.sourceforge.net/projects/javagroups/.

[3] Comparação com trabalhos relacionados e uma avaliação de desempenho detalhada podem ser encontradas na versão estendida desse artigo em http://www.lsd.dsc.ufpb.br/publications/JBossFT.ps.

2.1 *Java Servlets*

Para permitir que a aplicação possa ser utilizada através de *browsers* (clientes *Web*) via Internet ou em *intranets*, a plataforma J2EE inclui a especificação *Java Servlets* [1]. Esse serviço foi projetado para estender as funcionalidades de um servidor *Web* para que, além do conteúdo estático provido através de páginas no formato HTML (*HyperText Markup Language*), conteúdo dinâmico também pudesse ser oferecido. Um *servlet* pode ser definido como um componente *Web* gerenciado por um processo para gerar conteúdo dinâmico [1]; este processo é denominado *container*. O *servlet* consiste de uma classe Java que será carregada e executada a partir de invocações do servidor *Web*. Os *servlets* interagem com os *browsers* através de um protocolo de requisição/resposta baseado no HTTP (*Hypertext Transfer Protocol*) implementado pelo *container* do *servlet* [1].

O protocolo HTTP foi projetado para ser um protocolo sem estado [1]. Entretanto, muitas aplicações *Web* necessitam que uma seqüência de requisições originadas de um mesmo cliente possam ser associadas umas com as outras, por exemplo, quando um cliente deve ser autenticado antes de utilizar um serviço. Depois de o cliente informar a sua identificação e sua senha, a aplicação deve "lembrar" que toda requisição proveniente daquele *browser* está associada ao cliente que foi autenticado anteriormente. Assim, a aplicação deve manter um estado (sessão) em nome do cliente. A especificação *Java Servlet* define que o *container* deve implementar algum mecanismo de acompanhamento da sessão do cliente de forma que isso fique transparente para o desenvolvedor do *servlet* [1]. Através de uma interface única, o *servlet* pode salvar objetos na sessão associando-os com um nome para que o mesmo *servlet* ou um outro em execução no mesmo *container* possa recuperar esse objeto para utilizar em seu processamento.

Para garantir um nível maior de disponibilidade das aplicações *Web*, faz-se necessário que os *servlets* passem a estar em execução em mais de uma instância do servidor J2EE. Entretanto, uma vez que a aplicação pode armazenar diversas informações na sessão, um segundo servidor só poderá continuar o processamento de um primeiro que falhou se aquele tiver acesso às informações das sessões que estavam ativas antes da falha deste.

2.2 *Java Naming and Directory Interface*

Um serviço de grande importância na plataforma J2EE é o JNDI [7]. Ele consiste de uma especificação para que aplicações clientes em Java possam interagir com sistemas de nomes e diretórios, provendo uma interface comum para as várias implementações existentes. Através desse serviço é possível registrar objetos com um determinado nome para que aplicações distribuídas pela rede possam ter acesso aos mesmos a partir desse nome [6]. Os serviços de nomes e diretórios desempenham um papel muito importante em *intranets* e na Internet uma vez que provêm compartilhamento de informações por toda a rede.

A utilização de apenas uma instância do servidor que implementa o serviço JNDI pode provocar a perda das informações do serviço. Portanto, para que faltas sejam toleradas, deve-se optar pela utilização de mais de uma instância do servidor. Entretanto, fica a cargo da aplicação garantir que as instâncias do serviço JNDI permanecerão atualizadas uma vez que a especificação desse serviço não determina que isso deva ser feito pela plataforma J2EE. Além disso, quando uma aplicação deseja consultar o serviço JNDI, ela deve implementar o tratamento de eventuais falhas, isso inclui a detecção da falha e a localização de outra instância do serviço para realizar a consulta.

2.3 *Enterprise JavaBeans*

Outro serviço presente na plataforma J2EE é o EJB [8]. Nesse serviço, os componentes podem ser customizados sem a necessidade de alteração do código fonte, possibilitando que o desenvolvimento da aplicação se torne bastante simples. Os serviços providos pela infra-estrutura isentam

o desenvolvedor de lidar com aspectos mais complicados da implementação, como gerenciamento de transações, ciclo de vida de objetos, concorrência e segurança. Os componentes EJB ficam armazenados em um ou mais servidores e constituem o *framework* de serviços do desenvolvedor, podendo ser utilizado por qualquer sistema. Com essa tecnologia de componentes, pode-se fazer uso de uma programação declarativa, onde o componente possui propriedades que, quando modificadas, podem mudar o comportamento do componente. O serviço EJB provê um *container* que oferece para os componentes servidores um contexto de execução, além de gerenciamento e controle de serviços [8]. Com o EJB é possível criar componentes persistentes que representam os dados (*entity beans*) e componentes que contêm apenas lógica de negócio (*session beans*). Dessa forma, os desenvolvedores das aplicações apenas têm de criar a interface com o usuário que passaria os dados do usuário para os *session beans* que manipulariam os *entity beans* quando necessário.

Para tornar o serviço EJB tolerante a faltas, é necessário que cada componente esteja hospedado em mais de um servidor de aplicação para que, em caso de falha de um dos servidores, uma réplica do componente hospedada em um servidor livre de faltas possa assumir o processamento. Uma vez que o componente EJB caracteriza-se por manter um estado interno ao longo do seu ciclo de vida [7], é preciso que os estados das réplicas de um componente estejam sincronizados para que o fluxo do processamento possa continuar de forma consistente.

3 Projeto de uma Plataforma J2EE Tolerante a Faltas e Transparente

A solução adotada teve como base incluir em um servidor J2EE de código aberto mecanismos de replicação ativa. Como cada requisição a um serviço realizada por uma aplicação cliente será executada por todas as instâncias do grupo de replicação, a falha de uma instância não irá implicar na indisponibilidade da aplicação. A sincronização entre as réplicas será mantida através de um mecanismo de comunicação em grupo que provê as seguintes propriedades: **ordem** - as requisições são processadas por todas as réplicas na mesma ordem; e **atomicidade** - uma requisição ou é processada por todas as réplicas ou por nenhuma delas.

Os serviços da plataforma J2EE são utilizados pelas aplicações por meio de bibliotecas clientes que se encarregam de realizar a comunicação entre a aplicação e a instância do servidor. Dessa forma, para que o mecanismo de comunicação em grupo possa ser utilizado de forma transparente pelas aplicações clientes, as bibliotecas clientes tiveram de ser alteradas para que enviem a requisição feita pelo cliente para o grupo de replicação e não mais para uma instância. Em cada instância do servidor J2EE deverá haver um serviço em execução, denominado *GroupProxy*, que será responsável por interceptar as requisições enviadas ao grupo e repassá-las ao servidor J2EE. Após a requisição ser processada pelo servidor, o *GroupProxy* devolverá a reposta da requisição para a aplicação cliente que fez a requisição. A biblioteca cliente, que permanece esperando a resposta de requisição, retorna a primeira resposta recebida para o cliente que fez a requisição. A Figura 1 ilustra de maneira geral a arquitetura da solução.

Cada serviço da plataforma J2EE oferece uma maneira específica de acesso. A seguir estão descritas as alterações necessárias para cada serviço.

3.1 *Java Servlet* Tolerante a Faltas

Como foi explicado anteriormente, os *servlets* são acessados através de *browsers*. Uma solução que implique na alteração do *browser* para que estes passem a enviar uma requisição para o grupo de replicação ao invés de enviar a requisição para um servidor específico não seria viável. Entretanto, como muitas organizações já possuem um servidor *Web* para o conteúdo estático, os servidores J2EE geralmente oferecem um *plug in* para o servidor *Web* para que este redirecione as

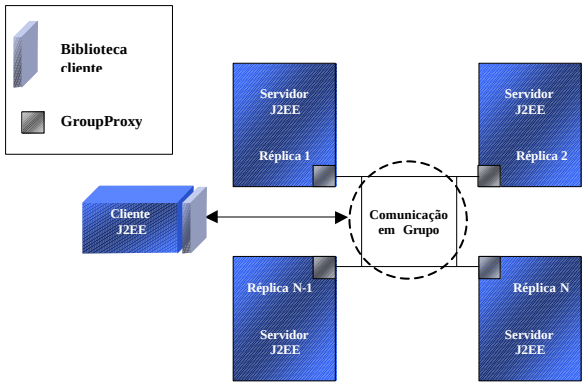


Figura 1: Arquitetura da solução

requisições ao conteúdo dinâmico para o *container* do *servlet*. Assim, este *plug in* teve de ser alterado para que, ao receber a requisição originada do *browser*, envie esta requisição para o grupo de replicação através do mecanismo de comunicação em grupo. O *plug in* será responsável também por filtrar as respostas geradas pelas réplicas para que apenas a primeira retorne ao *browser*.

3.2 JNDI Tolerante a Faltas

Um cliente JNDI acessa este serviço através de uma classe Java chamada *Context*, onde é especificado qual servidor que hospeda o serviço JNDI. Dessa forma, essa classe foi alterada para interagir com o mecanismo de comunicação em grupo de forma que, quando um cliente requisite a criação de um novo registro, a requisição seja recebida por todas as instâncias que formam o grupo de replicação. De forma similar, a consulta a um nome será realizada por todas as instâncias e o *Context* repassará ao cliente apenas a primeira resposta recebida, descartando as demais.

3.3 EJB Tolerante a Faltas

Um componente EJB hospedado em um servidor J2EE é utilizado pelas aplicações clientes através de um mecanismo de RMI. Na especificação RMI, qualquer objeto cujos métodos podem ser invocados de uma outra máquina virtual é chamado de *objeto remoto*. A localização física dos objetos remotos e dos clientes que os utilizam é irrelevante. Essa característica permite que o objeto remoto possa ser utilizado da mesma maneira tanto por objetos clientes da sua própria máquina virtual quanto por objetos em execução em outra máquina virtual (possivelmente em outra máquina).

Uma das grandes vantagens da tecnologia RMI é que todo o processo de comunicação entre os objetos fica transparente para os desenvolvedores do objeto remoto e do cliente. Dessa forma, a implementação do objeto remoto consiste apenas na lógica da aplicação. Para que isso seja possível, a infra-estrutura do RMI oferece objetos auxiliares que irão permitir que uma invocação de método feita pelo objeto cliente seja recebida pelo objeto remoto através do protocolo de rede [3]. Um desses objetos auxiliares é o *stub* que age como uma *referência remota* para a instância do objeto remoto. Este objeto (gerado a partir da compilação da interface remota e da implementação do objeto remoto) residirá na máquina virtual do cliente e irá interceptar as chamadas aos objetos remotos. O *stub* será responsável por interagir diretamente com o protocolo de rede para enviar as requisições (contendo a invocação do método) para a implementação do objeto remoto e receber desta a reposta para a invocação do método.

Para que o objeto remoto possa ser invocado por objetos clientes em execução em outras máquinas virtuais, ele deve ser registrado em um serviço de nomes como o RMIRegistry ou

JNDI [1]. Esse registro consiste em associar um nome com a instância do objeto remoto para que os objetos clientes possam obter uma referência remota para esse objeto a partir desse nome. Quando um objeto cliente deseja utilizar um objeto remoto, em primeiro lugar é estabelecida uma sessão com o serviço de nomes. Em seguida, a aplicação cliente consulta o serviço de nomes, informando o nome do objeto remoto desejado e o resultado dessa consulta é o *stub* do objeto remoto. A partir daí, os métodos que o cliente invocar serão enviados pelo *stub* para a instância real do objeto remoto para serem processados. Como o *stub* obtido é um objeto que implementa a mesma interface remota implementada pelo objeto remoto, o objeto cliente tem a "ilusão" de estar referenciado o objeto remoto, o que torna a utilização do *stub* transparente para o cliente.

A nossa abordagem para prover confiança no funcionamento para componentes EJBs consiste em implementar replicação ativa alterando o comportamento do serviço RMI. Sendo assim, para garantir que a replicação ativa será transparente tanto para o desenvolvedor da aplicação cliente quanto para os componentes EJBs que estarão fazendo uso do serviço RMI, o *stub* do objeto remoto deverá ser responsável por enviar a requisição para o grupo de instâncias do objeto remoto, utilizando o mecanismo de comunicação em grupo, e receber a primeira resposta da invocação, filtrando as demais.

4 JBossFT

Seguindo o projeto apresentado na seção anterior, foram implementados os mecanismos para prover replicação ativa para os serviços *Servlet*, JNDI e EJB. O servidor J2EE escolhido como base para implementação desses mecanismos foi o JBoss, esse servidor de aplicações é estruturado de tal forma que novos módulos podem ser adicionados sem que o seu código precise ser alterado. Basta que o novo módulo implemente algumas interfaces do *framework* do JBoss para que ele seja executado como mais um serviço desse servidor.

O mecanismo de comunicação em grupo adotado foi o JChannel, uma ferramenta implementada em Java que acompanha a biblioteca JavaGroups. Essa biblioteca oferece vários *padrões de projeto* para serem utilizados sobre qualquer ferramenta de comunicação em grupo [2]. Como o JavaGroups permite que a ferramenta possa ser desacoplada sem a necessidade de alteração da aplicação, outro mecanismo mais eficiente pode ser escolhido.

Por ser um serviço essencial para todo o projeto, o *GroupProxy* foi o primeiro a ser implementado. Ele consiste de um serviço em execução no JBoss aguardando as mensagens que estão sendo enviadas ao grupo de replicação. Através da mensagem recebida, o *GroupProxy* recupera a requisição da mensagem e a redireciona para o serviço apropriado da sua instância. Ele foi implementado utilizando um bloco básico fornecido pelo JavaGroups chamado *Listener* que notifica um determinado objeto quando alguma mensagem deve ser recebida pela réplica. Após o serviço processar a requisição, o *GroupProxy* envia a reposta do processamento para o integrante do grupo de replicação que originou a requisição.

Utilizando o JavaGroups, foi criado um novo bloco básico chamado *RequestDispatcher* que implementa um protocolo de requisição/resposta para ser usado nos outros componentes do projeto. Através dele, uma classe pode enviar uma mensagem que será recebida por todos os integrantes do grupo de replicação e receber apenas a primeira resposta referente à sua requisição. Esse bloco básico realiza a conexão com o grupo de replicação no momento de sua criação, e, quando recebe uma mensagem do seu cliente para ser enviada, ele adiciona um cabeçalho a esta mensagem contendo um identificador de requisição. Através das primitivas do JavaGroups, o *RequestDispatcher* envia essa mensagem como um *multicast* para o grupo ao qual ele pertence. Quando o *RequestDispatcher* recebe uma mensagem através do grupo de replicação, ele verifica se a identificação contida no cabeçalho corresponde à identificação de requisição da mensagem

enviada. Caso as identificações sejam iguais, essa mensagem é devolvida ao cliente que realizou a requisição, e as demais mensagens não serão mais consideradas. Utilizando o *RequestDispatcher*, o *plug in* que interage com o servidor *Web* foi alterado para enviar a requisição do *browser* para o grupo de replicação e não mais para uma instância.

Para facilitar a implementação do cliente JNDI (*Context*), foi implementado o bloco básico *GroupMethodCall* que consiste em um modelo de invocação replicada de método. Através dele, uma classe pode invocar um método a ser processado em todas as instâncias do grupo de replicação e receber a primeira resposta gerada. O cliente desse bloco básico precisa apenas informar o nome da classe, qual o nome do método dessa classe que deseja invocar e os argumentos do método. Com isso o *GroupMethodCall* monta uma mensagem e a envia utilizando *RequestDispatcher* para que esse método seja processado em todas as instâncias do grupo de replicação. Em sua forma original, o *Context* implementado pelo JBoss é apenas um cliente de um objeto remoto hospedado no servidor: em cada método do *Context*, é feita uma invocação via RMI para esse objeto remoto. Em nossa solução, o *Context* foi alterado para passar a utilizar o *GroupMethodCall* permitindo que a invocação ao método fosse replicada para todas as instâncias do objeto remoto no grupo de replicação.

A Figura 2 ilustra as modificações feitas no servidor de aplicação JBoss para implementar o JBossFT, que incorpora os mecanismos para tolerância a faltas descritos acima. Como pode-se perceber, o cliente J2EE continua utilizando as mesmas interfaces ao serviço JNDI (classe *Context*). Entretanto, essa classe foi alterada para que as requisições fossem direcionadas para o grupo por meio do JChannel. Em cada instância do JBossFT, o *GroupProxy* realiza a intercepatção das mensagens para serem processadas pelo serviço J2EE implementado pelo JBoss.

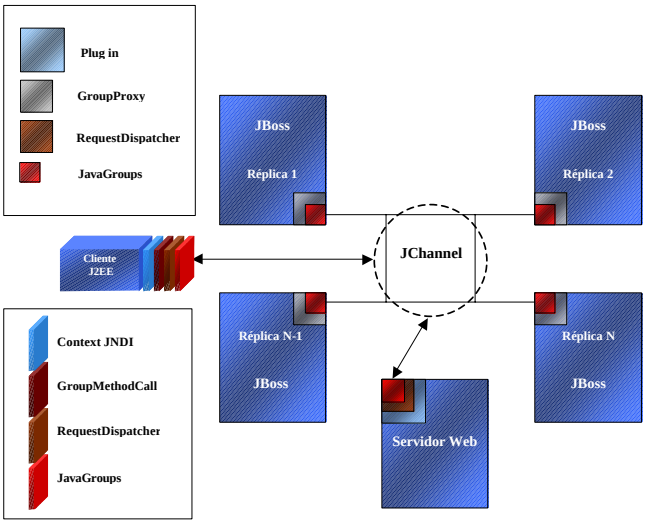


Figura 2: Arquitetura do JBossFT

Já a implentação de tolerância a faltas dos componentes EJB se baseia na implementação de um mecanismo transparente de invocação remota de método em um grupo de réplicas (*Group Method Invocation*, ou GMI). Para entender seu funcionamento é necessário, primeiro, entender como um objeto remoto não replicado é implementado.

Para permitir que um objeto possa ser referenciado a partir de uma outra máquina virtual, o desenvolvedor deve criar uma interface que herde de *java.rmi.Remote*. Nessa interface, devem ser declarados todos os métodos que poderão ser invocados remotamente. No exemplo mostrado na Figura 3, está definida a interface remota para um objeto remoto que irá somar dois valores.

```
public interface Calculador extends Remote {
    public int adicionar(int n1, int n2) throws RemoteException;
}
```

Figura 3: Interface de um objeto remoto

Definida a interface remota, o desenvolvedor deve criar uma classe que implemente essa interface (provendo a implementação dos métodos nela definidos). Essa classe normalmente herda de outras classes pré-definidas (ex. *java.rmi.server.UnicastRemoteObject*) algumas funcionalidades básicas do serviço RMI. A Figura 4 mostra a implementação do objeto remoto cuja interface foi definida na Figura 3.

```
public class CalculadorImpl extends UnicastRemoteObject implements Calculador {
    public int adicionar(int n1, int n2) throws RemoteException {
        return n1 + n2;
    }
}
```

Figura 4: Implementação de um objeto remoto

Quando o objeto cliente deseja utilizar um objeto remoto, ele realiza uma consulta ao serviço de nomes para obter o *stub* desse objeto, como indicado na Figura 5.

```
public class Cliente {
    public static void main(String[] args) throws Exception {
        // Conexão com o serviço JNDI
        Context ctx = new InitialContext();
        // Obtendo uma referência remota para o objeto Calculador
        Calculador objetoRemoto = (Calculador) ctx.lookup("Calculador");
        // Invocando o método remoto
        int resultado = objetoRemoto.adicionar(2, 2);
    }
}
```

Figura 5: Acesso a um objeto remoto

Deve-se destacar que o objeto cliente não sabe qual é a classe do *stub*, e sim que ele implementa a mesma interface remota implementada pelo objeto remoto. Essa característica permitiu que na nossa implementação fosse possível retornar um *stub* de uma outra classe, no momento que o objeto cliente consulta o serviço de nomes. Essa outra classe, a *GroupStub*, é responsável por interagir com o mecanismo de comunicação em grupo para replicar as invocações de método[4]. A classe do *GroupStub* é criada dinamicamente no momento em que é feita a consulta ao serviço JNDI.

Foi necessário implementar um serviço adicional que irá funcionar em conjunto com o serviço JNDI em cada instância do servidor de aplicação. Esse serviço, o *JNDIProxy*, intercepta as consultas para detectar quando o resultado da consulta é um *stub* de um objeto remoto. Caso a consulta seja a um objeto remoto, o *JNDIProxy* descobre qual a interface remota que o *stub* implementa através da API *Java Reflection*, que permite obter meta-informações (nome da classe, métodos, etc) de objetos [3]. Após identificar a interface remota do *stub*, o *JNDIProxy* cria a classe do *GroupStub* para o objeto remoto utilizando a classe *java.lang.reflect.Proxy*. Esta classe, do *Java Reflection*, é responsável pela criação de instâncias de classes *proxy* dinâmicas. No momento de criação de uma instância de uma *proxy* dinâmica, é possível especificar quais interfaces

[4]Esta classe herda boa parte de suas funcionalidades da classe *GroupMethodCall* discutida anteriormente.

essa instância irá implementar. Dessa forma, o *JNDIProxy* irá devolver um *GroupStub* que implementa as mesmas interfaces do *stub* do objeto remoto, sendo que este processo ficará totalmente transparente para o objeto cliente e não exigirá nenhuma alteração na implementação do objeto remoto.

Também nesse caso, o serviço *GroupProxy* em execução em cada instância do servidor J2EE é responsável por interceptar as requisições enviadas ao grupo e repassá-las ao objeto remoto. Após a requisição ser processada pelo objeto remoto, o *GroupProxy* devolve a reposta da invocação para o *GroupStub* do objeto cliente que fez a invocação. O *GroupStub*, que permanece esperando a resposta do método, retorna a primeira resposta recebida para o objeto cliente que realizou a invocação.

5 Conclusão

Esse artigo apresentou uma solução para desenvolver aplicações J2EE com requisitos de confiança no funcionamento onde os mecanismos para tolerar faltas estão implementados no servidor de aplicações. A nossa proposta teve como base implementar mecanismos de replicação ativa em um servidor J2EE de código aberto que provê alta confiabilidade e transparência para as aplicações. Essa implementação não exigiu que a lógica de nenhum serviço J2EE precisasse ser modificada. Foi necessário apenas que dois novos serviços (*GroupProxy* e *JNDIProxy*) fossem adicionados e que as bibliotecas clientes fossem modificadas. No projeto do servidor J2EE apresentado, o grupo de réplicas permanece totalmente transparente para aplicação e, em caso de falha de um ou mais servidores, o cliente poderá receber a resposta do seu processamento imediatamente, uma vez que sua requisição é processada pelas réplicas ainda operacionais.

Referências

[1] Java Servlet specification, version 2.2. SUN Microsystems, 1999.

[2] BAN, B. JavaGroups - group communication patterns in Java, 1998. http://www.cs.cornell.edu/home/bba/Patterns.ps.gz.

[3] CAMPIONE, M., WALRATH, K., AND HUML, A. *The Java Tutorial*. Addison-Wesley, 2000.

[4] CATTEL, R., AND INSCORE, J. *Criando aplicações comerciais com a plataforma Java 2, Enterprise Edition*. Editora Campus, 2001.

[5] COSTA, A. A. Usando replicação ativa para prover tolerância a falhas de forma transparente a uma implemetação da plataforma J2EE. Dissertação de mestrado, COPIN - Universidade Federal da Paraíba, Campina Grande, dezembro 2002.

[6] KASSEM, N., AND (EDITORS), E. T. *Designing Enterprise Applications with the Java 2 Platform*. Addison-Wesley, 2000.

[7] ROMAN, E. *Mastering Enterprise JavaBeans and the Java 2 Platform*. John Wiley & Sons, 1999.

[8] THOMAS, A. Enterprise JavaBeans technology server: Component model for the Java Platform. Patricia Seybold Group, 1998. http://java.sun.com/products/ejb/white/white_paper.html.