Prefácio - V Workshop de Testes e Tolerância a Falhas

O Workshop de Testes e Tolerância a Falhas (WTF) foi criado em 1998 como fórum de discussões na área congregando alunos de pós-graduação, pesquisadores e profissionais dessa área. Seu objetivo foi exercer um papel complementar ao Simpósio de Computação Tolerante a Falhas (SCTF), permitindo maior interação entre grupos e fortalecendo a pesquisa emergente em Testes e Tolerância a Falhas no Brasil.

O WTF foi promovido inicialmente de forma bi-anual e mutuamente exclusiva com o SCTF, tendo ocorrido segundo este regime nos anos 1998, 2000 e 2002. A partir de discussões no WTF2000, decidiu-se reunir a comunidade de Testes e Tolerância a Falhas em conjunto com outro evento de porte nacional que tivesse alguma relação com o WTF. O evento escolhido foi o Simpósio Brasileiro de Redes de Computadores, ou SBRC, devido à sua proximidade temática (o SBRC inclui trabalhos sobre Sistemas Distribuídos e Tolerância a Falhas). A partir do WTF2002, portanto, o evento passou a ser organizado como um dos Worshops satélite do SBRC.

Durante o WTF2002 em Búzios, refletindo a transformação do SCTF em evento internacional (o Latin-American Symposium on Dependable Computing, ou LADC) e a maior maturidade da área no Brasil, decidiu-se pela transformação do WTF em evento anual. Assim sendo, tivemos o WTF2003, ocorrido em Natal junto ao XXI SBRC, e o WTF2004, em Gramado, junto ao XXII SBRC.

A transformação do WTF em evento anual tem representado um desafio aos grupos de pesquisa na área de Testes e Tolerância a Falhas: seu sucesso depende da participação da comunidade, através de submissões ao evento e da presença no workshop. (São desafios ainda maiores em um país de dimensões continentais, onde a verba para pesquisa é exígua e muito dependente do auxílio do Governo.) É com felicidade, mas não com surpresa, que constatamos um significativo aumento no número de submissões em 2004: 27, em relação aos anos imediatamente anteriores (15 e 17, em 2003 e 2002, respectivamente).

O aumento do WTF não foi apenas em tamanho, mas aparentemente também em qualidade. Apesar do ótimo nível geral dos trabalhos na edição desse ano, por uma restrição de tempo (7 horas disponíveis ao evento), optou-se pela seleção de apenas 13 trabalhos, aqueles que receberam aceitação unânime dos revisores. Ao escolher um número menor de artigos, o Comitê de Programa priorizou dois aspectos: tempo para discussão da pesquisa e qualidade dos artigos aceitos. Estes são os ingredientes para um evento de sucesso.

Com 13 trabalhos selecionados, a taxa de aceitação do WTF2004 foi de 48%. Esta taxa é um dos critérios fundamentais utilizados no Qualis para enquadramento de eventos. No entanto, tomada como valor único, a taxa cria distorções, pois não considera a qualidade dos trabalhos submetidos. É hora de garantirmos que a qualidade dos trabalhos apresentados no WTF, a composição do seu Comitê de Programa, o seu histórico (quinta edição) e o apoio da SBC lhe rendam uma qualificação apropriada no Qualis: evento nacional B.

Aproveito este espaço para agradecer aos organizadores do SBRC2004, Lisandro Z. Granville e Maria Janilce Almeida, bem como Maria Izabel Cabral, pela excelente infra-estrutura oferecida aos organizadores dos workshops, e pelo excelente trabalho sendo realizado na organização do SBRC2003. Agradeço também aos autores pela submissão de seus trabalhos e aos membros do Comitê de Programa do WTF pela contribuição na seleção dos artigos que compõem o volume final do WTF, bem como pelas discussões realizadas no momento da escolha destes artigos (sem eles o WTF não seria possível).

Marinho Pilla Barcellos

Comitê de Programa

- Marinho P. Barcellos (UNISINOS) - coordenador
- Francisco Brasileiro (UFCG)
- Walfredo Cirne (UFCG)
- Rogério De Lemos (University of Kent, UK)
- Elias P. Duarte Jr. (UFPR)
- Joni S. Fraga (UFSC)
- Fabíola Greve (UFBA)
- Ingrid Jansch-Pôrto (UFRGS)
- Lau C. Lung (PUCPR)
- Raimundo José de Araújo Macêdo (UFBA)
- Eliane Martins (UNICAMP)
- Cecília M. F. Rubira (UNICAMP)
- Taisy R. Weber (UFRGS)
- Avelino F. Zorzo (PUCRS)

Revisores

- Marinho P. Barcellos (UNISINOS) - coordenador
- Alysson Neves Bessani (UFSC)
- Francisco Brasileiro (UFCG)
- Sérgio Cechin (UFRGS)
- Walfredo Cirne (UFCG)
- Antonio Casimiro Costa (Universidade de Lisboa, Portugal)
- Luis Carlos Erpen De Bona (UFPR)
- Rogério De Lemos (University of Kent, UK)
- Roberto Drebes (UFRGS)
- Elias P. Duarte Jr. (UFPR)
- Fabio Favarim (UFSC)
- Joni S. Fraga (UFSC)
- Fabíola Greve (UFBA)
- Ingrid Jansch-Pôrto (UFRGS)
- Maria Lucia Blanck Lisboa (UFRGS)
- Lau C. Lung (PUCPR)
- Raimundo José de Araújo Macêdo (UFBA)
- Eliane Martins (UNICAMP)
- Bogdan Tomoyuki Nassu (UFPR)
- Cecilia M. F. Rubira (UNICAMP)
- Jonatan Schroeder (UFPR)
- Patricia Sereda (UFPR)
- Gabriela Jacques da Silva (UFRGS)
- Frank Siqueira (UFSC)
- Taisy R. Weber (UFRGS)
- Roverli Pereira Zivich (UFPR)
- Avelino F. Zorzo (PUCRS)

V Workshop de Testes e Tolerância a Falhas

Implementação e Análise de Desempenho de um Mecanismo Adaptativo para Tolerância a Falhas em Sistemas Distribuídos com QoS

Sérgio Gorender[1], Raimundo J. A. Macêdo, Matheus Cunha

Laboratório de Sistemas Distribuídos (LaSiD)
Departamento de Ciência da Computação (DCC)
Universidade Federal da Bahia (UFBA)
Av. Adhemar de Barros S/N, Ondina,
Salvador - BA, CEP: 40.170-11

{gorender, macedo, matheusc}@ufba.br

Resumo. Tolerância a falhas e adaptabilidade são requisitos importantes para sistemas distribuídos modernos, especialmente aqueles que precisam se adaptar dinamicamente para diferentes níveis de qualidade de serviço (*QoS*). Neste artigo, apresentamos a implementação de uma infraestrutura de comunicação e gerenciamento de recursos de *QoS*, implementada sobre uma rede de estações LINUX equipadas com um pacote de controle de trafego (*DiffServ*). Baseados nesta estrutura, mostramos a implementação de um mecanismo de tolerância a falhas adaptável em tempo de execução (*run-time*) a diferentes níveis de *QoS*, o qual é composto de um detector de defeitos e um módulo de consenso. O protocolo de consenso apresentado goza de uma propriedade bastante interessante: a de poder operar concomitantemente, numa mesma execução, com diferentes níveis de *QoS* para processos distintos, caracterizando um modelo híbrido de tolerância a falhas. Também apresentamos dados de desempenho coletados a partir de vários experimentos onde foram medidos os tempos de obtenção de consenso para cenários diversos.

Abstract. Fault tolerance and adaptability are important issues for modern distributed systems. The capability of dynamically adapting to distinct run-time conditions is especially relevant for environments where negotiated QoS conditions cannot always be delivered between communicating processes. In this paper, we present an implementation of a framework for communication and resource management, using a network of LINUX workstations, which were equipped with a traffic control packet. We also implemented a fault tolerant mechanism, composed of a failure detector and a consensus protocol, which dynamic adapts to distinct QoS levels. The consensus protocol has the property that in the same consensus distinct processes can have different QoS, which characterize a hybrid fault tolerant model. We also measure the execution of the consensus, and present the result of these experiments.

[1] Aluno de doutorado do CIN/UFPE, pesquisador do LaSiD/UFBA e professor do DCC/UFBA e da Faculdade Ruy Barbosa.

1. Introdução

Tolerância a falhas adaptativa é um requisito importante para sistemas distribuídos modernos, especialmente em sistemas que executam em ambientes com diferentes níveis de qualidade de serviço (*QoS*) [20]. Os mecanismos de tolerância à falhas, em geral, assumem um modelo de execução específico. No modelo síncrono existem limites temporais conhecidos e garantidos para os serviços de execução (execução de passos dos processos e transferência de mensagens), enquanto que no modelo assíncrono estes limites temporais não existem. Para o modelo assíncrono foi provado em [10] a impossibilidade de se obter o consenso tolerando falhas. Devido a esta impossibilidade foram criados os modelos chamados parcialmente síncronos, que estendem o modelo assíncrono com características síncronas [9, 8, 7, 19]. O consenso distribuído é freqüentemente utilizado como uma forma de testar a tolerância à falhas destes modelos, além de ser utilizado em diversas aplicações distribuídas.

Por outro lado, utilizando as novas tecnologias para prover um ambiente de execução com diferentes níveis de serviço [3, 5, 20], podemos obter um ambiente híbrido, no qual convivem serviços de execução isócronos (serviços realizados dentro de limites de tempo previamente definidos) e serviços de execução não isócronos (sem limites de tempo). Além disso, os serviços de execução podem ter o seu nível de qualidade alterado dinamicamente, o que exige dos sistemas a capacidade de se adaptar a estas alterações. Ou seja, além das aplicações, os mecanismos de tolerância a falhas precisam ser adaptativos nesses ambientes com QoS.

Neste artigo, apresentamos a implementação de uma infraestrutura de comunicação e gerenciamento de recursos de *QoS*, implementada sobre uma rede de estações LINUX equipadas com pacotes de controle de tráfego funcionando segundo a arquitetura Serviços Diferenciados (*DiffServ*) [5]. Baseados nessa estrutura, mostramos a implementação de um mecanismo de tolerância a falhas, composto de um detector de defeitos e um módulo de consenso, que são adaptáveis em tempo de execução (*run-time*) a diferentes níveis de *QoS* [11, 12]. O protocolo de consenso descrito goza de uma propriedade bastante interessante: a de poder operar concomitantemente, numa mesma execução, com diferentes níveis de *QoS* para os processos distintos, caracterizando um modelo híbrido de tolerância a falhas. Também apresentamos dados de desempenho coletados a partir de vários experimentos onde foram medidos os tempos de obtenção de consenso para cenários diversos.

Na seção 2, a seguir, são apresentadas uma infraestrutura de comunicação e gerenciamento de recursos e sua implementação. Na seção 3 são apresentados os conceitos básicos de um detector de defeitos adaptativo e de um algoritmo de consenso híbrido e adaptativo, assim como a implementação destes dois mecanismos. Na seção 4 são apresentados resultados obtidos com a execução do protótipo implementado e uma análise de desempenho. Os trabalhos correlatos são apresentados na seção 6 e na seção 7 apresentamos algumas conclusões.

2. Infraestrutura de comunicação e gerenciamento de recursos

O ambiente de comunicação é composto por canais de comunicação, os quais são providos com diferentes níveis de serviço, segundo a arquitetura Serviços Diferenciados (*DiffServ*) para prover QoS [5]. A arquitetura DiffServ determina que os fluxos de informação (pacotes de dados) a ser transferidos sejam atribuídos a classes de serviço para o encaminhamento de pacotes. Estas classes de serviço determinam o nível do

serviço de comunicação que será provido ao fluxo de pacotes. Cada classe é definida baseada em requisitos, tais como um atraso limitado ou não limitado para a transferência de pacotes, um *jitter* (variação no atraso) alto ou restrito, uma maior ou menor prioridade para o descarte de pacotes e uma maior ou menor largura de banda. Cada classe de serviço na arquitetura *DiffServ* é identificada através de um valor especial, chamado *DiffServ Code Point* (DSCP), o qual é armazenado no campo *Type of Service* (ToS) do cabeçalho IP de cada pacote. Quando um pacote entra em uma rede *DiffServ*, diversos campos de seu cabeçalho IP são analisados com o objetivo de se identificar a que classe de encaminhamento o pacote deve ser associado (classificação do pacotes), e o pacote tem um valor DSCP armazenado no seu campo ToS (marcação do pacote). O roteador que marca os pacotes é chamado de roteador de borda. Os demais roteadores no caminho do pacote até o seu destino (rota), os quais são chamados de roteadores de núcleo, apenas analisam o valor DSCP armazenado mo cabeçalho do pacote e encaminham o pacote de acordo com a classe de serviço associada a este valor.

Existem diversas classes de serviço definidas na arquitetura *DiffServ*. Entre estas, os Serviços Expressos fornecem uma comunicação isócrona, garantindo limites máximos para a transferência de pacotes. Esta garantia é fornecida através de altas prioridades para o encaminhamento, e de uma alta taxa de transmissão, suficiente para transmitir imediatamente todo pacote que chegue a um roteador. As demais classes fornecem serviços de comunicação não isócronos, sem garantias temporais. Estas classes atribuem diferentes prioridades para o descarte de pacotes, no caso de haver congestionamentos nos roteadores. Chamamos os canais de comunicação criados com serviços isócronos (Serviço Expresso) de canais de comunicação *timely*, e os canais de comunicação criados com serviços não isócronos de canais *untimely*.

Implementamos classes de serviço isócrono e não isócrono para o encaminhamento de fluxos de pacotes foram implementadas no ambiente de execução, o qual é composto por em estações LINUX configuradas para funcionar como roteadores. Para criar estas classes de serviço utilizamos um pacote de controle de tráfego para o LINUX, chamado TC. Este pacote define uma linguagem, a qual é utilizada para a definição das classes de serviço para o encaminhamento de pacotes, através da construção de *scripts*. As classes de serviço são criadas a partir da definição de filas para o armazenamento dos pacotes a serem encaminhados, uma para cada classe. Para cada fila é definida uma prioridade para o encaminhamento de pacotes, além de um algoritmo específico para o gerenciamento da fila e disciplinas de filas adequadas à definição da classe. A disciplina de fila DSMark [20] implementa a marcação de pacotes com um valor DSCP nos roteadores de borda, antes de estes roteadores encaminharem os pacotes. Esta mesma disciplina de fila é utilizada pelos roteadores de núcleo para classificarem cada pacote para a classe de serviço apropriada, dependendo apenas do conteúdo do campo ToS (valor DSCP). Na definição das filas para o encaminhamento de pacotes, o TC permite definir a largura de banda aplicada a cada fila, a prioridade para o encaminhamento de pacotes, o algoritmo de fila a ser utilizado e outros requisitos, permitindo assim a definição de uma classe de encaminhamento isócrona, com uma alta largura de banda e prioridade máxima para o encaminhamento de pacotes.

A infraestrutura de gerenciamento de comunicação (QoS Provider) e o mecanismo para sistemas distribuídos tolerantes a falhas (detector de defeitos e consenso) foram implementados na linguagem Java e testados sobre o ambiente de

comunicação criado com o LINUX e o pacote TC.

2.1 QoS Provider

Denominamos de *QoS Provider* o mecanismo que desenvolvemos e implementamos cuja atribuição é criar canais de comunicação para a aplicação e gerenciar o nível do serviço de comunicação fornecido a cada canal. Este mecanismo executa distribuído em módulos associados aos processos do sistema. Gerenciar a QoS dos canais de comunicação implica em negociar o nível do serviço a ser provido a cada canal e monitorar este serviço durante a sua execução. As funções *CreateChannel(), DefineQoS(), Delay()* e *QoS()* são chamadas pelos processos das aplicações distribuídas para criar canais de comunicação entre processos, negociar a *QoS* para um determinado canal de comunicação, determinar o atraso na transferência de mensagens para um canal de comunicação e monitorar a *QoS* corrente atribuída a um canal de comunicação. É importante notar que a função *Delay()* retorna um valor probabilístico se o canal de comunicação for *untimely*, mas retorna um valor determinado caso este canal de comunicação seja *timely*.

Além destas funções, o QoS Provider monitora todos os canais de comunicação *timely*, para verificar se estes canais continuam sendo providos com serviços isócronos. Esta monitoração é feita utilizando mensagens de um protocolo de sinalização, que permitam a troca de informações entre os módulos do QoS Provider e os roteadores e provedores de serviços de comunicação. Sempre que for identificada alguma alteração na qualidade de serviço provida a um canal, fazendo este canal passar a *untimely*, esta alteração é sinalizada aos processos comunicantes.

Os conceitos que foram utilizados no desenvolvimento do QoS Provider foram baseados em diversas arquiteturas desenvolvidas para prover *QoS* fim-a-fim a aplicações distribuídas [3], como por exemplo a arquitetura Omega e o dispositivo QoS Broker [18].

2.2 Implementação do QoS Provider

Foi desenvolvida uma implementação inicial do QoS Provider com o objetivo básico de criar canais de comunicação e fornecer estes canais para os processos solicitantes. Nesta implementação as funções *CreateChannel()* e *Delay()* foram implementadas. As demais funcionalidades do QoS Provider, tais como negociar a *QoS* dos canais de comunicação e monitorar esta *QoS* ainda estão sendo desenvolvidas. Nesta implementação inicial as classes de serviço isócrono e não isócrono são criadas diretamente nos roteadores LINUX, através da execução de *scripts* criados com a linguagem TC. Desta forma, a cada teste realizado, pudemos estabelecer os canais de comunicação como sendo *timely* ou *untimely*.

O QoS Provider possui duas classes principais, *Provider* e *Negotiator*. A classe *Provider* faz a interface entre o processo que está requisitando o canal de comunicação e o provedor de serviço da rede, e utiliza a classe *Negotiator* para acessar o provedor de serviço de comunicação. Quando um processo solicita a criação de um canal de comunicação com um outro processo, a instância da classe *Negotiator* pertencente ao processo solicitante envia uma mensagem *BeginNegotiation* ao outro processo. Ao receber esta mensagem, a instância da classe *Negotiator* deste outro processo registra a requisição e envia uma mensagem *EndNegotiation* para o processo solicitante, finalizando assim o estabelecimento do canal de comunicação. O canal de comunicação

criado é do tipo *UDP/IP*.

3. Implementação de um mecanismo adaptativo tolerante a falhas para sistemas distribuídos

Apresentamos uma visão geral do mecanismo adaptativo tolerante a falhas para sistemas distribuídos [11, 12], o qual foi implementado em Java/LINUX sobre a infraestrutura de comunicação apresentada na seção 2.

Um sistema distribuído é composto por um conjunto $\Pi=\{p_1, ..., p_n\}$ de processos, conectados por canais de comunicação pertencentes ao conjunto $C=\{c_{1/2}, ..., c_{1/n}, ..., c_{n-1/n}\}$. Os conjuntos Π e C formam o grafo completo $DS(\Pi, C\}$.

Assumimos no nosso sistema o modelo assíncrono aumentado com detectores de defeitos não confiáveis, como apresentado por Chandra & Toueg em [6]. Assumimos adicionalmente que os canais de comunicação podem ser fornecidos com *QoS*, de acordo com a arquitetura Serviços Diferenciados, como apresentado na seção anterior.

Assumimos também que os processos falham por *crash*.

3.1 Detector de defeitos adaptativo

O detector de defeitos adaptativo executa distribuído em módulos, um para cada processo do sistema. Os módulos do detector de defeitos dos processos em funcionamento enviam solicitações periódicas de mensagens de *heartbeat* para todos os demais módulos através dos canais de comunicação, e esperam por mensagens de *heartbeat* em resposta, até a ocorrência de um *timeout*, previamente calculado. O *timeout* para cada mensagem de *heartbeat* é calculado a partir do atraso para a transferência de mensagens, o qual é determinado para cada canal de comunicação. Por sua vez, este atraso é calculado pela função *Delay()* do QoS Provider. Para canais de comunicação *timely* o atraso informado pela função *Delay()* é garantido, enquanto o canal permanecer *timely*, enquanto que para canais *untimely* o atraso informado é probabilístico. Considerando as classes de serviço dos canais de comunicação existem dois tipos de detecção de defeitos: suspeitas, no caso de o canal de comunicação ser *untimely*, e notificações, se o canal de comunicação for *timely*. Como os módulos divulgam as notificações realizadas, um processo que possua ao menos um canal de comunicação *timely* será notificado por todos os processos corretos, caso venha a falhar.

Conseqüentemente, podemos classificar os processos operacionais como suspeitáveis ou notificáveis, inserindo a identificação destes processos, respectivamente, nos conjuntos *suspeitáveis* e *notificáveis*. Esta classificação é efetuada e atualizada por cada módulo do detector de defeitos, a partir da análise de uma representação local do grafo DS com informações sobre a *QoS* de cada canal de comunicação, obtidas através da função *QoS()* do QoS Provider, e de informações que são trocadas entre os módulos. Apenas processos cuja identificação pertence ao conjunto *notificáveis* podem ser notificados pelos módulos do detector de defeitos. Os demais processos só podem ser suspeitos de terem falhado. Processos notificáveis não podem ser erroneamente suspeitos por nenhum módulo do detector de defeitos, uma vez que um módulo que não possua canal de comunicação *timely* com este processo não irá monitorá-lo, apenas notificando uma falha deste processo, se receber uma mensagem, de outro módulo, comunicando esta notificação.

Processos que são notificados têm a sua identificação retirada do conjunto

notificáveis e inserida no conjunto *faltosos*, enquanto processos suspeitos têm a sua identificação inserida no conjunto *suspeitos*, apesar de continuarem processos suspeitáveis (a identificação destes processos não é retirada do conjunto *suspeitáveis*). Todos estes conjuntos são locais a cada módulo do detector de defeitos. Os módulos do detector de defeitos atualizam seus conjuntos quando notificam processos ou suspeitam de processos, ou quando percebem alterações na *QoS* dos canais de comunicação. Os processos obtêm informações do seu módulo do detector de defeitos a respeito do estado do sistema através dos conjuntos fornecidos pelo módulo.

Um detector de defeitos com as características descritas acima possui as propriedades *strong completeness* (todas as falhas são detectadas, em algum momento, por todos os processos operacionais), *eventual weak accuracy* (a partir de algum momento haverá um processo correto que não será suspeito erroneamente por qualquer processo operacional) e Conditional QoS (processos notificáveis não são detectados erroneamente). Definimos que um detector de defeitos com estas propriedades pertence à classe de detectores $\Diamond S^{Adapt}$. As propriedades *strong completeness* e *eventual weak accuracy* são características de detectores de defeitos da classe $\Diamond S$, conforme definidas por Chandra & Toueg em [6]. Na seção 5 apresentamos uma definição resumida dos detectores de defeitos não confiáveis.

3.2 Consenso híbrido adaptativo

O algoritmo de consenso híbrido e adaptativo implementado resolve o problema do consenso uniforme, caracterizado pelas propriedades validade (se um processo decidir por um valor *v*, *v* deve ter sido proposto por algum processo), terminação (todos os processos corretos devem decidir por algum valor) e acordo uniforme (dois processos corretos não devem decidir de forma diferente). Este algoritmo executa em rodadas assíncronas com coordenadores circulantes, adotando um padrão de troca de mensagens descentralizado, no qual o coordenador, se não falhar, envia seu valor estimado para todos os processos, e estes respondem, também a todos, com uma mensagem *ack*, se a estimativa do coordenador foi recebida, ou com uma mensagem *nack*, se o coordenador foi notificado ou suspeitado. O número de mensagens que cada processo deve receber, em cada rodada, para poder progredir, decidindo ou não, é chamado de quorum. Se um processo recebe apenas mensagens *ack* de um quorum de processos ele decide pelo valor proposto pelo coordenador.

No nosso algoritmo, o quorum é composto por todos os processos notificáveis do sistema mais uma maioria dos processos suspeitáveis, sendo, portanto um quorum híbrido. Como os conjuntos notificáveis e suspeitáveis podem ser alterados durante a execução do consenso, o quorum também será alterado dinamicamente, durante o consenso, caracterizando uma adaptação do consenso à capacidade do detector de defeitos de realizar detecções.

3.3 Implementação do Detector de Defeitos Adaptativo

O detector de defeitos foi implementado segundo o modelo *pull*, no qual uma mensagem é gerada por um módulo do detector de defeitos para cada outro módulo, solicitando um *heartbeat* de resposta.

Todos os processos possuem um módulo do detector de defeitos, o qual é composto por diversas classes, as quais são instanciadas como *threads* do processo em execução. A classe *HeartBeatCenter* envia periodicamente mensagens

HeartBeatMessage para todos os processos em execução, caracterizando uma solicitação de uma mensagem de *heartbeat*. Ao receber uma mensagem *HeartBeatMessage*, um *HeartBeatCenter* envia em resposta uma mensagem *HeartBeatAnswer*.

A *thread* detector, também instanciada para cada processo, verifica o atraso nas respostas dos *heartbeats* enviados e recebidos pelo *HeartBeatCenter* (mensagens *HeartBeatMessage* e *HeartBeatAnswear*). A *thread* detector verifica o atraso no recebimento de mensagens *HeartBeatAnswear* de cada processo, chamando o método *isLate*(p_i, p_j), sendo p_i o processo que está executando o método e p_j o processo a ser monitorado. Este método verifica se o *timeout* definido para o recebimento de uma mensagem *HeartBeatAnswear* do processo monitorado foi alcançado sem o recebimento da mensagem, e se o processo monitorado é notificável ou suspeitável. Dependendo se houve um atraso e da situação do processo, o módulo do detector pode suspeitar do processo monitorado, notificar uma falha deste processo ou nada fazer. O *timeout* para o recebimento de cada mensagem *HearBeatAnswear* é determinado a partir do atraso máximo definido para cada canal de comunicação, o qual é calculado e informado pela função *Delay()* do QoS Provider.

3.4 Implementação do Consenso Híbrido Adaptativo

O algoritmo de consenso é instanciado como uma *thread*, pertencendo a cada processo. Esta *thread* possui um método que a cada mensagem recebida verifica se o quorum foi alcançado, e se a mensagem recebida foi do tipo *ack* ou *nack*. Se o quorum for alcançado com mensagens *ack* o consenso decide.

4. Análise de resultados

Os testes foram realizados utilizando três computadores com processadores Pentium III de 900 Mhz, 128 Mb de memória principal, executando o sistema operacional LINUX Conectiva 8. O *kernel* utilizado foi o 2.4.18. Os computadores estão conectados em rede, através de segmentos de 10 *megabits*. O *kernel* do LINUX foi recompilado para que as máquinas funcionassem como roteadores *DiffServ*.

Em todos os computadores foram executados *scripts* do pacote TC para a criação de filas para o encaminhamento de pacotes provendo serviços isócronos e serviços não isócronos. De acordo com cada teste realizado o *DiffServ* foi executado um *script* específico em cada roteador, associando fluxos de dados de cada processo às classes de encaminhamento adequadas para o teste.

Nesta primeira implementação não foi nosso objetivo obter tempos expressivos na execução do consenso, por isso não dedicamos tempo demasiado em sua otimização. Espera-se que, após a otimização desta implementação, os tempos obtidos serão melhorados.

4.4 Resultados e análise de desempenho

Foram realizados dois tipos básicos de testes. O primeiro teste foi feito com o objetivo de verificar o quanto o aumento no número de processos notificáveis no sistema causaria um aumento no tempo de execução do consenso. Para tanto realizamos testes do consenso com 3 processos, com 6 processos e com 9 processos, todos eles notificáveis, e sem ocorrer falhas. Foram executados 10 testes para cada situação. Em todos estes testes o consenso foi obtido na primeira rodada.

Nos testes realizados com 3 processos obtivemos um tempo médio de execução do consenso de 97,3 milissegundos, enquanto que com 6 processos obtivemos o tempo médio de 444,4 milissegundos, e com 9 processos foi obtido o tempo médio 505,9 milissegundos. Como a implementação do detector de defeitos e do consenso ainda está sendo otimizada acreditamos que estes tempos podem ser melhorados. Entretanto, a cada mensagem recebida, o algoritmo de consenso efetua um processamento para verificar se o quorum foi obtido, e se foi possível chegar a uma decisão. Este processamento faz com que o tempo necessário para o tratamento de cada mensagem cresça, aumentando por conseqüência o tempo do consenso. Na figura 1 apresentamos um gráfico contendo os tempos médios de consenso obtidos em cada conjunto de experimentos.

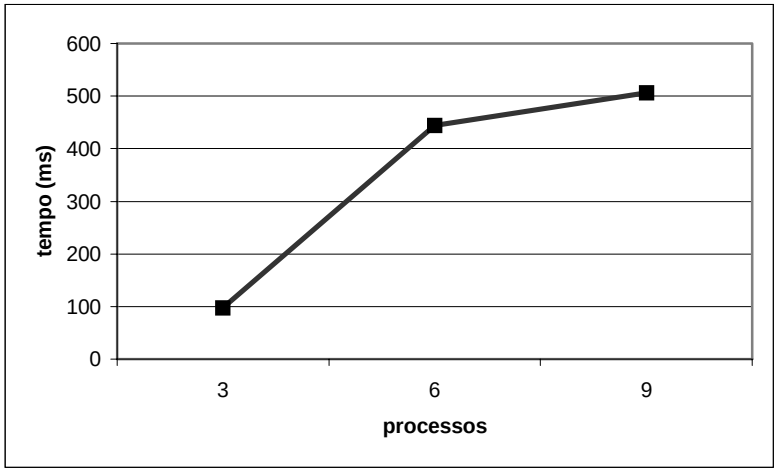

Figura 1: Tempos médios do consenso com 3, 6 e 9 processos.

Realizamos um segundo teste, com o objetivo de comparar os tempos de execução do consenso com um conjunto de processos notificáveis, e com um conjunto de processos suspeitáveis. Os resultados destes testes são apresentados na figura 2. Foram realizados 10 execuções de cada consenso, sempre com 6 processos, sem a ocorrência de falhas. No caso do consenso com processos suspeitáveis o quorum é formado por 4 processos, enquanto que no consenso com processos notificáveis o quorum passa a ser formado por todos os 6 processos. Devido ao *overhead* gerado pelo processamento das mensagens para verificar o quorum, o consenso executado com processos notificáveis apresenta tempos de execução superiores ao consenso com processos suspeitáveis.

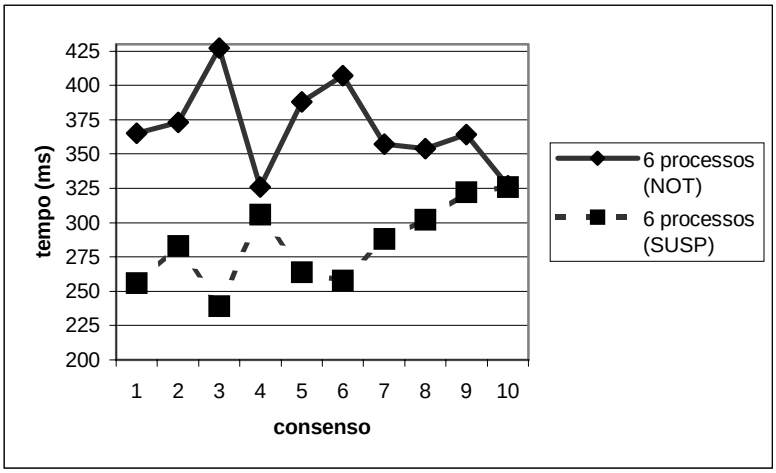

Figura 2: Consenso com 6 processos, todos notificáveis ou todos suspeitáveis.

Um outro resultado interessante que obtivemos com alguns testes realizados e com o processo de otimização da implementação, é que a necessidade de se policiar o tráfego gerado pelos processos, para que este tráfego não venha a exceder a capacidade de transmissão negociada com a rede, pode vir a aumentar o atraso na transferência de mensagens, também aumentando o tempo de execução do consenso. Em uma implementação anterior utilizamos um dispositivo para controlar o envio de pacotes, de acordo com uma taxa máxima previamente negociada, e ao retirarmos este dispositivo, os tempos obtidos para a execução do consenso diminuíram consideravelmente.

5. Trabalhos Correlatos

Tolerância à falhas em sistemas distribuídos assíncronos tem sido enfocada, de uma maneira geral, estendendo o ambiente assíncrono com alguma característica síncrona, uma vez que, como provado em [10], em ambientes assíncronos é impossível executar aplicações, como o consenso, tolerando qualquer número de falhas. Os modelos parcialmente síncronos, criados segundo este ponto de vista, permitem que aplicações sejam executadas, tolerando um número determinado de falhas. As soluções desenvolvidas dependem, em geral, do ambiente de execução e modelo de falhas adotado.

Dolev et al estenderam os resultados da impossibilidade do consenso, apresentados em [10], combinando 5 parâmetros, que podem ser favoráveis (síncronos) ou desfavoráveis (assíncronos) [8]. Dwork et al apresentam dois modelos parcialmente síncronos [9], um no qual os limites de tempo para a execução de ações e a transferência de mensagens existem, mas são desconhecidos, e outro no qual estes limites são conhecidos, mas só passam a valer depois de um tempo não determinado. Cristian e Feltzer apresentam o modelo Assíncrono Temporizado (*Timed Asynchronous*) [7], no qual os processos acessam relógios locais, e se assume que os ambientes de execução apresentam períodos de estabilidade, nos quais as ações dos processos são executadas com limites de tempo conhecidos e determinados, e apenas a um número limitado de mensagens é permitido ser entregue após um tempo limitado. Neste modelo foi criado o conceito de *fail awareness*, o qual determina que as ações que tenham excedido seu limite de tempo para executar, as mensagens que tenham sido entregues após o seu *timeout* e os relógios que apresentem um desvio de tempo acima de um limite pré-definido, sejam marcados como apresentando falha de performance. Casimiro e

Veríssimo apresentaram a Base de Computação Temporizada (*Timely Computing Base*) [19]. A TCB executa em um ambiente síncrono, monitorando a aplicação, sinalizando falhas de performance e executando ações temporizadas para a aplicação. Chandra e Toueg apresentaram o modelo assíncrono equipado com detectores de defeitos não confiáveis, os quais são distribuídos em 8 classes distintas, definidas pelas propriedades *accuracy* e *completeness* [6]. A propriedade *accuracy* diz respeito à capacidade de o detector não efetuar detecções incorretas e a propriedade *completeness* diz respeito à capacidade de o detector efetuar detecções corretas de todos os processos que falharam.

Algoritmos de consenso foram desenvolvidos para todos estes modelos, sendo que diversos protocolos de consenso, baseados em diferentes conceitos e paradigmas, têm sido desenvolvidos para executar com os detectores de defeitos não confiáveis, em especial os detectores da classe ◊S, caracterizados como os mais fracos a permitir a execução do consenso tolerando falhas. Além de Chandra e Toueg, algoritmos de consenso foram desenvolvidos por Mostefaoui e Raynal [16] e Hurfin et al [13], entre outros. Lamport desenvolveu o algoritmo de consenso Paxos para o sistema assíncrono [15], estendido com um detector de defeitos para eleição de líderes. Algoritmos de consenso híbridos, utilizando detectores de defeitos e oráculos para a geração de números aleatórios foram desenvolvidos em [1, 17]. Algoritmos de consenso que acessam oráculos para geração de números aleatórios foram introduzidos por Ben Or em [4].

De uma maneira geral, detectores de defeitos têm sido implementados utilizando o conceito de *timeout*, o qual determina limites no tempo para que mensagens sejam entregues ou ações sejam executadas. Entretanto, algumas implementações de detectores de defeitos não têm utilizado *timeouts*, monitorando os processos com base na análise das mensagens transferidas pela própria aplicação, e detectando falhas se um número demasiado de mensagens da aplicação não tiver sido entregue [2].

Existem diferentes formas para prover adaptação a sistemas distribuídos tolerando falhas. Os algoritmos de consenso descritos em [16, 13, 1, 17] possuem características de adaptabilidade, no sentido em que podem utilizar diferentes oráculos em sua execução, como os algoritmos descritos em [1, 17], os quais utilizam um oráculo detector de defeitos e um oráculo gerador de números aleatórios, quando o primeiro não funciona. Nós adotamos como objetivo do nosso mecanismo a adaptação a um ambiente de execução com QoS, que possa prover serviços de execução (em especial comunicação) com diferentes níveis de qualidade, o que difere das soluções conhecidas,

Utilizamos em nosso trabalho os detectores de falhas não confiáveis de Chandra & Toueg, porém em um ambiente de execução que altera dinamicamente o nível dos serviços fornecidos, exigindo de nosso mecanismo a capacidade de se adaptar dinamicamente a estas alterações. O nosso algoritmo de consenso é também adaptativo, sendo, além disto, híbrido, por considerar a existência de componentes isócronos e de componentes não isócronos, no ambiente de execução.

6. Conclusão

Apresentamos neste artigo uma implementação de um ambiente de comunicação que cria canais de comunicação e gerencia o nível de serviço provido a cada canal. Este ambiente de comunicação foi implementado sobre uma rede de estações LINUX executando um sistema para controle de tráfego, o qual permite definir disciplinas de enfileiramento de pacotes baseadas na arquitetura Serviços Diferenciados, para prover

QoS a serviços de comunicação. Também apresentamos a implementação em Java de uma infraestrutura tolerante a falhas para sistemas distribuídos, a qual é composta por um detector de defeitos adaptativo, e por um algoritmo de consenso híbrido, também adaptativo. As detecções de defeitos obtidas de um módulo do detector de defeitos podem ser suspeitas ou notificações, dependendo do nível de serviço provido a cada canal de comunicação. O consenso é obtido utilizando um quorum híbrido, o qual é composto por processos notificáveis e por processos suspeitáveis.

Os resultados obtidos mostraram haver um aumento no tempo de execução do consenso à medida que aumenta o número de mensagens a serem processadas. Isto se deve ao fato de que a cada mensagem recebida o consenso necessita verificar se o quorum foi alcançado e o consenso obtido. Estamos trabalhando para otimizar este processamento, e com isto, reduzir o tempo do consenso. Por outro lado, verificamos que, com um número maior de processos, o tempo total para a obtenção do consenso por todos os processos operacionais não será tão elevado, uma vez que, o primeiro processo a obter o consenso irá comunicar sua decisão aos demais processos, os quais poderão decidir ao receber a mensagem de decisão, antes de completarem o seu quorum.

Na próxima versão de nossa implementação o QoS Provider executará de forma a negociar e monitorar a *QoS* provida a cada canal de comunicação.

Referências

[1] Aguilera, M. K. e Toueg, S., *Failure Detection and Randomization: A Hybrid Approach to Solve Consensus*, SIAM Journal of Computing, 28(3), 1998, pp. 890-903, junho, 1999.

[2] Aguilera, M. K., Chen, W. e Toueg, S., *Heartbeat: a timeout-free failure detector for quiescent reliable communication*, *Proceedings of the 11th International Workshop on Distributed Algorithms*, Lecture Notes on Computer Science. Springer-Verlag, setembro, 1997.

[3] Aurrecoechea, C., Cambell, A. T. e Hauw, L., *A Survey of QoS architectures*, Multimedia Systems 6(3), Springer-Verlag, pp – 138-151, maio, 1998.

[4] Ben Or, M., *Another Advantage of Free-Choice: Completely Asynchronous Agreement Protocols*, Proceedings of the 2nd Annual ACM Symposium on Principles of distributed Computing (PODC 1983), pp. 27-30 , agosto, 1983.

[5] Blake, S. at al, "An Architecture for Differentiated Services", RFC 2475, dezembro, 1998.

[6] Chandra, T. D. e Toueg, S., *Unreliable Failure Detectors for Reliable Distributed Systems*, Journal of the ACM, Vol. 43 (2), pp. 225-267, março, 1996.

[7] Cristian, F. e Fetzer C., *The Timed Asynchronous Distributed System Model*, IEEE Transactions on Parallel and Distributed Systems, Vol. 10 (6), pp. 642-657, junho, 1999.

[8] Dolev, D., Dwork, C. e Stockmeyer, L., *On the Minimal Synchronism Needed for Distributed Consensus*, Journal of the ACM, Vol. 34 (1), pp. 77-97, janeiro, 1987.

[9] Dwork, C., Lynch, N. e Stockmeyer, L., *Consensus in the Presence of Partial Synchrony*, Journal of the ACM, Vol. 35 (2), pp. 288-323, abril, 1988.

[10] Fisher, M. J., Lynch, N. A. e Paterson, M. S., *Impossibility of Distributed Consensus with One Faulty Process*, Journal of the ACM, vol. 32 (2), pp. 374-382, abril, 1985.

[11] Gorender, S. e Macêdo, R. *Um Modelo para Tolerância a Falhas em Sistemas Distribuídos com QoS*, Anais do Simpósio Brasileiro de Redes de Computadores (SBRC2002), pp. 277-292, maio, 2002.

[12] Gorender, S. e Macêdo, R., *A Dynamically QoS Adaptable Consensus and Failure Detector*, The IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2002 – Fast Abstract Track, pp. B80-B81, junho, 2002.

[13] Hurfin M., Macêdo R., Mostefaoui A. e Raynal M., *A Consensus Protocol based on a Weak Failure Detector and a Sliding Round Window*, Proceedings of the 20th IEEE Int. Symposium on Reliable Distributed Systems (SRDS'01). New Orleans, USA, pp. 120-129, outubro, 2001.

[14] Keidar, I. e Rajsbaum, S., *On the Cost of Fault-Tolerant Consensus When There Are No Faults – A Tutorial*, SIGACT News 32(2), Distributed Computing column, pp. 45-63, junho,2001.

[15] Lamport, L., *The Part Time Parliament*, ACM Transactions on Computer Systems, 16(2), pp. 133-169, maio, 1998.

[16] Mostefaoui, A. e Raynal, M., *Solving Consensus Using Chandra-Toueg's Unreliable Failure detectors: a General Quorum-Based Approach*, in Proceedings of the 13th International Symposium on Distributed Computing (Disc1999), pp. 49-63, setembro, 1999.

[17] Mostefaoui, A., Raynal, M. e Tronel, F., *The Best of Both Worlds: a Hybrid Approach to Solve Consensus*, Proceedings of the 2000 International Conference on Dependable Systems and Networks (DSN 2000), pp. 513-522, junho, 2000.

[18] Nahrstedt, K. e Smith, J. M., *The QoS Broker*, IEEE Multimedia, 2(1), pp – 53-67, março, 1995.

[19] Veríssimo, P., Casimiro, A. e Fetzer, C., *The Timely Computing Base: Timely Actions in the Presence of Uncertain Timeliness*, Proceedings of the International Conference on Dependable Systems and Networks, pp. 533-542, junho, 2000.

[20] Xiao, X. e Ni, L. M., "Internet QoS: A Big Picture", IEEE Network, pp. 8 – 18, março/abril, 1999.

Implementando Recuperação por Retorno Baseada em *Checkpointing* em Sistemas Distribuídos Assíncronos

Clairton Buligon , Sérgio Cechin , Ingrid Jansch-Pôrto

Universidade Federal do Rio Grande do Sul
Programa de Pós-Graduação em Computação
91501-970 – Caixa Postal 15064
Bairro Agronomia – Porto Alegre – RS

{clairton, cechin, ingrid}@inf.ufrgs.br

Abstract. *The rollback-recovery from previous checkpoints is largely employed as a fault-tolerant technique. The complexity of distributed system models has motivated the development of different algorithms, which aim at offering simpler and more efficient solutions than the preceding ones. In our Fault Tolerance Group, an algorithm has been recently proposed: it envisages asynchronous distributed systems based on message passing, operates with coordinated non-blocking checkpointing and ensures treatment for orphan and lost messages. This paper describes the algorithm implementation challenges, the decisions and our results until the present moment.*

Resumo. *A recuperação por retorno baseada em pontos de recuperação é largamente usada como técnica de tolerância a falhas. O modelo complexo de sistemas distribuídos tem motivado o desenvolvimento de diversos algoritmos buscando soluções mais simples e eficientes. No Grupo de Tolerância a Falhas da UFRGS, foi proposto recentemente um algoritmo que é voltado para aplicações em sistemas distribuídos assíncronos baseados na troca de mensagens, opera com salvamento coordenado de pontos de recuperação e prevê o tratamento de mensagens órfãs e perdidas. Este artigo descreve as decisões de projeto, a implementação do algoritmo e resultados obtidos até o momento.*

1. Introdução

O uso de recuperação de processos para garantir tolerância a falhas em sistemas distribuídos já é um assunto bastante conhecido. Vários protocolos foram propostos na literatura [Elnozahy et al., 2002] para *checkpointing* (salvamento das informações que poderão ser usadas posteriormente) e recuperação de processos em ambientes distribuídos, mas pouco se conhece a respeito de implementações destas propostas. A implementação de algoritmos de recuperação em sistemas distribuídos, principalmente aqueles enquadrados na categoria assíncrona, tem sido alvo de pesquisa, uma vez que ainda restam pontos em aberto. A complexidade associada a resolução de problemas de consistência e garantia de transparência para aplicações que incorporam recuperação de processos são bons exemplos de pontos em aberto que ainda merecem serem discutidos.

Em vista desta complexidade, é comum implementações ficarem restritas a ambientes fechados, como em ambientes MPI (*Message Passing Interface*), sistemas agregados (*Clusters*), ou nos recentes ambientes em grade (*Grids*), onde existe infra-estrutura física (*hardware*) e lógica (*software*) coordenando o processamento e garantindo qualidade nos serviços. Em ambientes distribuídos convencionais, onde não dispomos desta infra-estrutura, trabalhos de implementação de *checkpointing* geralmente concentram-se no desenvolvimento de bibliotecas e conjuntos de primitivas que implementam mecanismos de tolerância a falhas. Há, portanto, carência de implementações completas de recuperação sobre as quais possam ser executadas aplicações distribuídas.

A recuperação por retorno considera sistemas distribuídos como uma coleção de processos que se comunicam através de uma rede. Durante a execução livre de falhas, os processos efetuam o armazenamento dos pontos de recuperação (*checkpointing*) em uma memória estável que sobrevive às falhas previstas no modelo. Estes pontos de recuperação contêm informações sobre os processos, necessárias para, em caso de ocorrência de falhas, garantirem a retomada do processamento a partir de um momento intermediário, anterior ao da manifestação da falha, reduzindo assim a quantidade de computação perdida. As informações de recuperação incluem, no mínimo, o estado dos processos participantes. Diferentes aplicações e os algoritmos de recuperação adequados podem requerer informações adicionais, tais como o estado dos periféricos de entrada e saída, eventos que ocorram durante o processamento ou mensagens trocadas entre os processos.

Sistemas envolvendo a troca de mensagens apresentam maiores desafios a recuperação por retorno porque as mensagens induzem dependências entre os processos durante a execução livre de falhas. Em caso de falhas, processos que não falharam podem ser forçados a retroceder a computação, causando a propagação dos efeitos do retorno da computação. Em alguns cenários, esta propagação pode provocar inclusive o retorno para o estado inicial da computação, causando o chamado efeito dominó [Jalote, 1994], que deve ser evitado. Para isso, uma das técnicas envolve a coordenação entre processos visando garantir linhas de recuperação em execuções livres de falhas, que representam um estado consistente sob o ponto de vista do sistema. Obviamente, técnicas como esta exigem que cada processo mantenha seus pontos de recuperação baseados nas informações das mensagens recebidas de outros processos. Dependendo do ambiente, ainda devem ser tratados os problemas das mensagens órfãs e perdidas em conseqüência dos próprios mecanismos de recuperação [Cechin, 2002].

Segundo Elnozahy, a recuperação por retorno possui muitos enfoques quanto à transparência desta técnica, sob o ponto de vista da aplicação ou dos programadores do sistema. O sistema pode contar com a aplicação para decidir “quando” e “o quê” salvar na memória estável. A transparência pode limitar-se a algumas características de uso, provendo apenas alguns mecanismos que auxiliam a estruturar a aplicação ou ser mais abrangente, não requerendo nenhuma intervenção por parte da aplicação ou do programador. O sistema automaticamente obtém os pontos de recuperação de acordo com alguma política pré-estabelecida e recupera-os se uma falha ocorrer. A vantagem desta abordagem é liberar os programadores de tarefas complexas e com tendência a erros na implementação de tolerância a falhas nas aplicações, além de garantir a inserção desta propriedade em aplicações convencionais de maneira plenamente transparente.

Implementar um protocolo de recuperação por retorno em sistemas distribuídos

assíncronos, levando em consideração aspectos de transparência e inexistência de infra-estrutura de apoio, é o objetivo do presente trabalho. O algoritmo de recuperação escolhido [Cechin, 2002] opera por retorno, é coordenado, não determinístico e não bloqueia a aplicação. Este artigo aborda as atividades de implementação deste sistema. Nas seções seguintes, serão apresentados: o modelo de sistema distribuído, a descrição do algoritmo empregado, o suporte disponível para possibilitar sua implementação, a implementação propriamente dita e, por fim, as conclusões obtidas até o momento e atividades futuras.

2. Modelo de sistema distribuído

O modelo de sistema distribuído é composto por um conjunto de processos, interconectados através de uma rede de comunicação, que realizam um processamento distribuído baseado exclusivamente na troca de mensagens. Os processos são distribuídos individualmente em diferentes computadores, com diversas capacidades de processamento, e estes processos podem sofrer colapso (*crash*) segundo o modelo de defeitos definido por Cristian [1991].

A rede de comunicação corresponde a um modelo lógico [Jalote, 1994] composto de canais unidirecionais, do tipo assíncrono, interligando cada par de processos do sistema. Neste modelo, cada par de processos possui associados dois canais de comunicação, possibilitando a troca bidirecional de mensagens. Através da utilização de canais unidirecionais, o receptor de uma mensagem pode identificar o transmissor. Os canais introduzem um atraso imprevisível porém finito às mensagens.

A computação é modelada por uma seqüência de eventos, geralmente associados à troca de mensagens. Cada processo é composto por, no mínimo, quatro módulos (Figura 1): módulo de aplicação, módulo de recuperação, módulo detector de defeitos e módulo de comunicação.

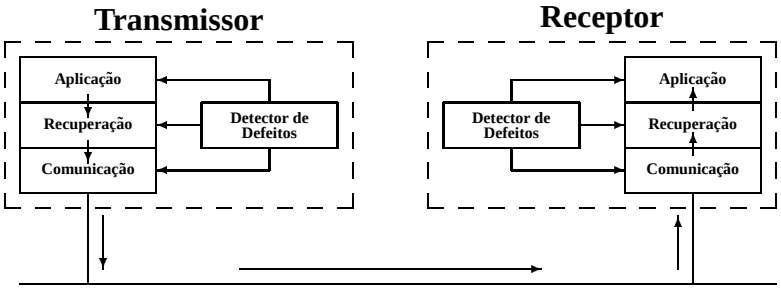

Figura 1: Modelo de computação distribuída

No módulo de aplicação, é executada a computação distribuída. No módulo de recuperação, são implementados os mecanismos para o estabelecimento das linhas de recuperação e, em caso de falhas, posterior restauração de estados e retomada das atividades. A detecção de erros deve informar sobre a ocorrência destes, em processos e no meio de comunicação, e sobre a restauração de um componente defeituoso. O módulo de comunicação é responsável pela troca de informações: envio, recebimento e entrega de mensagens.

3. Algoritmo utilizado

O algoritmo a ser implementado baseia-se na coordenação não-bloqueante entre processos para o salvamento dos pontos de recuperação, garantindo o estabelecimento de linhas de recuperação. No algoritmo, um coordenador inicia o estabelecimento de uma linha de recuperação, enviando uma solicitação a cada processo do sistema. Ao receber esta solicitação, cada processo estabelece seu ponto de recuperação e responde ao coordenador. O coordenador, após receber as respostas de todos os processos, envia a todos a confirmação e encerra a operação de estabelecimento desta linha de recuperação, estando pronto para iniciar uma nova. Para tolerar falhas no coordenador, foi sugerida a implementação de um mecanismo de rotação de coordenadores.

Como parte do mecanismo de consistência, todas as mensagens têm associado um índice que identifica qual foi o último ponto de recuperação estabelecido pelo seu processo transmissor, denominado "índice do intervalo do ponto de recuperação". Ao receber uma mensagem, o receptor verifica este índice. Se for maior que o seu próprio índice, então um novo ponto de recuperação é estabelecido antes da entrega da mensagem para a aplicação. Caso contrário, a mensagem é entregue sem que sejam realizados outros procedimentos. O algoritmo foi projetado para permitir a troca de mensagens da aplicação, simultaneamente às mensagens de coordenação, necessárias ao estabelecimento das linhas de recuperação.

O controle de mensagens perdidas e mensagens órfãs também é efetuado pelo algoritmo. As mensagens podem ser perdidas por falha no meio de comunicação ou devido ao próprio mecanismo de recuperação.

O algoritmo possibilita a coleta de lixo, ou seja, descartar os pontos de recuperação que fazem parte de linhas de recuperação que deixaram de ser úteis. Esta coleta é acionada pelo processo coordenador ao término do estabelecimento de cada linha de recuperação.

O retorno consistente é garantido pela existência de uma linha de recuperação previamente estabelecida pelo algoritmo. Os mecanismos de recuperação serão disparados a partir da detecção das falhas toleradas pelo sistema ou nas situações de retomada do processamento interrompido intencionalmente.

Para verificar a correção do algoritmo, o autor optou pela utilização da lógica temporal (TLA) de Lamport [1994], usada na especificação dos mecanismos propostos e na demonstração de sua correção e consistência. Cabe ressaltar que a compreensão dos formalismos componentes do algoritmo facilitou o seu pleno entendimento e favoreceu a sua implementação.

4. Ambiente de implementação

A definição do ambiente de implementação do algoritmo e de condições de funcionamento (em oposição a uma implementação genérica) tornou-se obrigatória para viabilizar a primeira fase do sistema. Decisões como a escolha do sistema operacional, adoção alternativa de bibliotecas e ambientes de simulação, estratégias de programação e definição do perfil das aplicações são requisitos que, previamente definidos, direcionam a implementação para um determinado caminho. Esta seção tem por objetivo apresentar as decisões tomadas durante a fase preliminar do trabalho e suas justificativas, procurando

especificar o ambiente de desenvolvimento do algoritmo.

Identificou-se que determinados mecanismos empregados pelo algoritmo influenciam diretamente na escolha do ambiente de implementação e das ferramentas de apoio. Mecanismos de *checkpointing* e recuperação de processos e interceptação de mensagens da aplicação são bons exemplos. Além disso, a implementação destes mecanismos sem acarretar alterações ou parametrizações da aplicação, seria facilitada se o ambiente de desenvolvimento fornecesse o suporte necessário.

A disponibilidade de aplicações para testes também é um aspecto determinante no projeto de implementação. É necessário identificar no ambiente de implementação aplicações adequadas, que efetuem computação distribuída baseada na troca de mensagens, com resultados suficientes para uma validação preliminar. Conhecer, por exemplo, o perfil destas aplicações com relação a parâmetros como: volume de mensagens geradas, o tipo e quantidade de informação a ser manipulado e parâmetros de configuração e execução garantirá a correta interpretação dos resultados de avaliações posteriores de desempenho do algoritmo.

Com base nestas considerações, bem como no modelo de sistema distribuído adotado, partiu-se para uma investigação dos ambientes disponíveis para implementação e execução de tolerância a falhas em sistemas distribuídos.

4.1. Sistema operacional

Um caminho possível a ser seguido é a implementação direta sobre o sistema operacional. A investigação partiu dos sistemas operacionais mais difundidos atualmente: MS Windows e Linux.

Para o primeiro, pouco se localizou na literatura a respeito de trabalhos relacionados à recuperação por retorno em sistemas distribuídos. Entretanto, através de uma consulta nas propriedades deste sistema operacional, concluiu-se ser uma boa alternativa para uma implementação completa, ou seja, com a confecção de todos os mecanismos necessários. Baseado em Win32 API, que é a interface das aplicações com o sistema operacional, é possível implementar todas as funcionalidades exigidas pelo algoritmo através de alguma linguagem de programação [Nebbet, 2000].

Por outro lado, existe uma riqueza de propostas e trabalhos baseados no sistema operacional Linux. Apesar de algumas implementações existentes serem bastante restritas [Sankaran et al., 2003], seu código-fonte aberto e sua arquitetura amplamente divulgada, como apresentado em Beck et al. [1999], Bar [2000] e Rubini [1999], tornaram-no um ponto de partida bastante confortável para o desenvolvimento deste trabalho. No caso em estudo, onde o algoritmo impõe o uso de mecanismos não convencionais, não houve dificuldades em perceber a viabilidade de sua implementação.

Foram primeiramente identificadas no sistema operacional Linux alternativas para implementação de mecanismos para interceptação de mensagens e manipulação de processos. Um exemplo dessas alternativas é a utilização de chamadas de sistema (*system call*), como a chamada `ptrace`, que permite a um processo o controle total sobre outro processo [Beck et al., 1999]. Seu uso pode ser encontrado no trabalho de Fontoura [2002], onde foi apresentado um estudo sobre as possíveis abordagens para captura de informações da aplicação: integração, interceptação e serviço; avaliação de desempenho

destas abordagens e testes de uso efetuados a partir de protótipos. Concluiu-se, dentro de um contexto bem definido, que as abordagens de integração e interceptação poderiam ser empregadas no desenvolvimento do presente trabalho.

Quanto aos mecanismos de *checkpointing* e recuperação de processos, foram encontradas informações suficientes para sustentar a possibilidade de uma implementação transparente sob o ponto de vista das aplicações. Visando obter embasamento para desenvolver código específico ao salvamento de pontos de recuperação e sua retomada posterior, foram estudadas algumas bibliotecas reunidas no sítio *Checkpointing.org* (http://www.checkpointing.org). São exemplos destas: `libckpt`, `epckpt`, `crak` [Zhong and Nieh, 2001], `ckpt` e `chpox`, todas com código-fonte disponível, implementadas em linguagem C/C++ e sobre o sistema operacional Linux, explorando diferentes abordagens. `Libckpt`, por exemplo, implementa uma alternativa em nível de usuário, exigindo alteração do código-fonte e recompilação da aplicação alvo. Por outro lado, `epckpt` e `crak` permitem utilização transparente em diferentes níveis: dentro do sistema, em camada entre o sistema e a aplicação e como processo independente. O estudo destas implementações contribuiu com o presente trabalho de duas formas distintas: uso como bibliotecas, na sua forma original, ou como fonte de estudo para uma possível implementação dos mecanismos necessários.

Os resultados da investigação do ambiente determinaram a implementação diretamente sobre o sistema operacional Linux. A disponibilidade de bibliotecas que oferecem alguns mecanismos acessórios do algoritmo, tais como bibliotecas de *checkpointing* e recuperação de processos, serão exploradas para reduzir consideravelmente os esforços para implementação do algoritmo.

4.2. Nível de *kernel* ou nível de usuário

A investigação das bibliotecas resultou na identificação de possíveis implementações em diferentes níveis: nível de sistema e nível de usuário. Alessandro Rubini [1999] e Richard Stones [Stones and Matthew, 1999] referem-se a estas duas alternativas como implementação em "nível de *kernel*" e "nível de usuário", respectivamente, e apresentam detalhes de uso. Neste trabalho, desejava-se identificar qual das abordagens mais facilitaria a implementação do algoritmo, além de estudar aspectos relacionados à proteção do sistema operacional. Em nível de usuário, somente usuários privilegiados podem executar determinadas tarefas. Em nível de *kernel*, o acesso ao sistema é total.

Neste contexto, novamente devem ser observadas as características do algoritmo, e feita a avaliação das possibilidades e conveniência de mecanismos ou níveis alternativos. No presente trabalho, a interceptação de mensagens da aplicação é um mecanismo do algoritmo que pode ser citado como exemplo. Em Linux, identificou-se que as mensagens de um processo podem ser interceptadas de duas formas: em nível de usuário, através da chamada `ptrace` e, em nível de *kernel*, através de "ganchos" do *kernel* [Fontoura, 2002]. Estes "ganchos" também são referenciados em outros trabalhos como *syscall track*. A primeira forma é relativamente fácil de implementar. Entretranto, segundo Fontoura, exige processamento adicional além de comprometer o desempenho. A complexidade da segunda forma vai depender do conhecimento do *kernel*, por parte do desenvolvedor, e do refinamento dos mecanismos que visam garantir o desempenho, o qual depende da forma de implementação definida pelo programador. Em seu trabalho, Fontoura demonstrou ser

esta uma alternativa interessante, comprovando com medidas de desempenho a partir de um conjunto de testes realizados.

4.3. Ferramenta `crak`

Uma decisão importante no trabalho, que praticamente definiu o ambiente de desenvolvimento, foi a escolha da biblioteca de *checkpointing* e recuperação de processos. Uma análise preliminar das bibliotecas estudadas foi suficiente para escolher a biblioteca `crak` desenvolvida por Hua Zhong [Zhong and Nieh, 2001]. Desenvolvida na forma de um *device driver* [Rubini, 1999], em nível de *kernel*, ela implementa *checkpointing* e recuperação de processos em Linux. A partir do identificador de um determinado processo (PID), é possível salvar o seu estado em disco e posteriormente restaurar este estado, inclusive em outro computador. O foco do autor foi a garantia de transparência, desempenho, atomicidade e consistência nas operações. Apesar de ser voltada exclusivamente para migração de processos, seu conceito e funcionalidade vieram ao encontro das necessidades deste trabalho.

Por ter código-fonte aberto, foi possível adaptar a biblioteca para uso nesta implementação, adicionando as funcionalidades necessárias. Destas, destacam-se a implementação de mecanismos de interceptação e retransmissão de mensagens, manipulação de descritores de conexões de rede, inserção e remoção de informações de controle do algoritmo nas mensagens da aplicação, novas funções de interface com o usuário entre outras.

Por ser voltada para processos individuais, a biblioteca não implementa o salvamento do estado das conexões de rede dos processos, em sua forma original. Na última versão, seu autor iniciou uma tentativa de manipulação de conexões TCP (*network socket*), procurando garantir o restabelecimento destas conexões, no caso da recuperação de um processo. Entretanto, devido à complexidade da operação que envolve dois processos distintos, os mecanismos empregados transferem o controle do problema para as ferramentas `rsh` e `ssh`. Para o presente trabalho, necessita-se ter o controle também dos estados das conexões de rede dos processos, implementando seu restabelecimento em caso de falhas. O controle sobre o resultado do restabelecimento destas conexões é imprescindível para a efetivação do protocolo em um ambiente distribuído, onde uma possível restauração global da aplicação distribuída deve ser precisa e executada com segurança.

A partir de uma investigação a respeito do restabelecimento de conexões de rede dos processos, identificou-se que em *network sockets* não orientados a conexão, como conexões UDP, o tratamento é menos complexo. Decidiu-se então restringir as aplicações àquelas que fazem uso de conexões UDP para a troca de mensagens. Dessa forma, a biblioteca foi modificada para garantir a manipulação de conexões UDP e, na sua totalidade, a biblioteca passou a suprir os principais mecanismos necessários para a implementação do algoritmo: *checkpointing* e recuperação de processos de forma transparente, garantindo restabelecimento de conexões de rede e interceptação de mensagens da aplicação.

4.4. Implementação restante

A partir do exposto nas seções precedentes, ficou evidente que para atingir-se os objetivos do trabalho, faltaria basicamente reunir os mecanismos pré-desenvolvidos para obter a implementação do algoritmo propriamente dito.

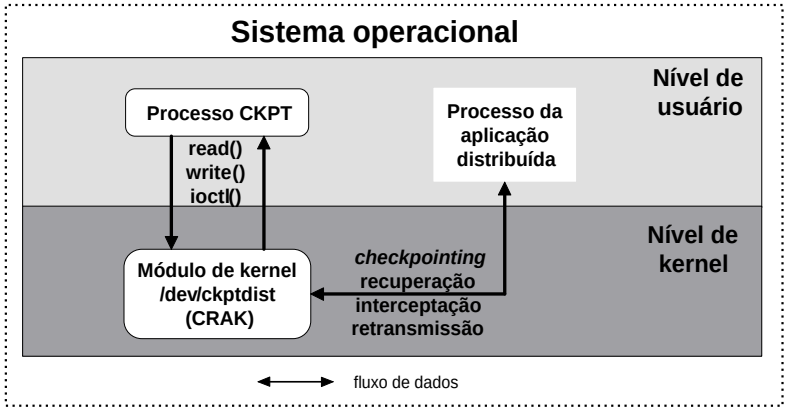

Figura 2: Estrutura da implementação sobre sistema operacional Linux

Por razões de familiaridade e facilidade de programação, optou-se por implementar o restante dos mecanismos em nível de usuário, ou seja, como um processo de usuário. Não há impeditivos para que tal implementação seja efetuada diretamente em nível de *kernel*, na forma de um *device driver* ou alterando-se o código-fonte original do sistema operacional. Adicionalmente, o restante dos mecanismos a serem implementados não exige acesso privilegiado ao sistema operacional. Além disso, em nível de usuário existem bibliotecas e funções disponíveis que facilitam sua implementação [Stones and Matthew, 1999], tais como *message queues*, *curses*, *threads*, funções de acesso a rede (*network sockets*) e arquivos, enquanto que em nível de *kernel*, esses mecanismos deveriam ser totalmente implementados.

5. Implementação

A partir do entendimento do algoritmo e seus formalismos, da definição do ambiente de desenvolvimento e da escolha das ferramentas e bibliotecas de apoio, iniciou-se a fase de implementação. Esta seção tem por objetivo apresentar as decisões e suas justificativas, na busca da especificação do ambiente de desenvolvimento do algoritmo.

5.1. Estrutura geral

Como referenciado na seção anterior, o desenvolvimento do algoritmo procurou explorar os níveis de *kernel* e de usuário do sistema operacional Linux. No primeiro, partindo-se da ferramenta `crak`, implementou-se na forma de módulo de *kernel*, ou *device driver*, aquelas funcionalidades consideradas não convencionais: *checkpointing*, recuperação de processos e interceptação de mensagens da aplicação. Por outro lado, implementou-se em nível de usuário, na forma de um processo, os demais mecanismos do algoritmo, como comunicação em grupo, estabelecimento das linhas de recuperação, gerência de configurações, mecanismos de detecção de erros, recuperação em caso de falhas, e demais mecanismos necessários para o funcionamento do algoritmo.

A Figura 2 representa as camadas de usuário e de *kernel* do sistema operacional e, respectivamente, o processo CKPT e o módulo de *kernel* `/dev/ckptdist` que trabalham em conjunto durante a execução do algoritmo. Pode-se considerar o módulo de *kernel* como uma entidade passiva, onde as funções implementadas aguardam que eventos

ocorram para então serem acionadas. Estes eventos podem ser gerados pelo próprio sistema, como o envio ou recebimento de mensagens da aplicação, ou pelo processo CKPT, através das chamadas de função necessárias.

O módulo `/dev/ckptdist` mantém na memória, estruturada em um *buffer*, uma cópia de todas as mensagens enviadas e recebidas pela aplicação. De acordo com o algoritmo, é necessário salvar determinadas mensagens da aplicação para prevenir eventual necessidade de retransmissão em caso de falha ou devido aos próprios mecanismos de recuperação. Esta funcionalidade é gerenciada pelo processo CKPT, que se encarrega de ler as mensagens deste *buffer*, descartando, solicitando a retransmissão ou salvando as mensagens de acordo com a execução do algoritmo.

O interfaceamento entre o processo CKPT e o módulo `/dev/ckptdist` é realizado através das funções `read`, `write` e `ioctl`. O acesso a estas funções é realizado de forma análoga ao acesso a arquivos. A função `read` é usada para ler as mensagens armazenadas no *buffer* do módulo. A função `write` escreve aquelas mensagens da aplicação que devem ser retransmitidas e a função `ioctl` é empregada para demais configurações e controles necessários, tal como o estabelecimento de um novo *checkpoint*.

5.2. Interceptação de mensagens

A interceptação de mensagens, outro ponto importante do algoritmo, também é totalmente realizada em nível de *kernel*. Neste nível, ao contrário da implementação em nível de usuário, é possível interceptar-se somente as *system calls* que interessam, evitando sobrecarga desnecessária no sistema. No caso do algoritmo, duas *system calls* são interceptadas: `sys_execve` e `sys_socketcall`. Com a primeira, é possível obter o PID do processo no momento da sua execução inicial, possibilitando o controle do processo desde sua chamada pelo `shell` do usuário. A segunda, e mais importante, refere-se à interceptação das mensagens da aplicação. Através desta interceptação é possível implementar o algoritmo no que diz respeito à garantia de consistência das mensagens, tratando possíveis mensagens órfãs e perdidas. Também nesta interceptação é possível anexar informações de controle do algoritmo às mensagens da aplicação, de maneira transparente.

Na Figura 3, é apresentada parte do código-fonte do módulo `/dev/ckptdist` relacionada à inicialização do módulo e interceptação das *system calls*. Basicamente o "gancho", ou *syscall track*, resume-se em alterar a tabela `sys_call_table`, trocando-se a referência da chamada da função original para outra função, implementada pelo programador. Esta troca de ponteiros é efetuada no momendo da inserção do módulo no *kernel*. Na remoção do módulo, a referência original deve ser restaurada.

5.3. Salvamento de estados e recuperação em caso de falhas

O salvamento dos estados dos processos e a recuperação também são realizados em nível de *kernel*. O estado salvo envolve a área de memória usada pelo processo, o conjunto de registradores, descritores de arquivos abertos, diretório de trabalho corrente, estado do terminal, *signal handlers*, mensagens sem confirmação de entrega e estado das conexões de rede envolvendo *socket UDP*. O conteúdo dos estados é salvo em arquivos, localmente, e o mecanismo de restauração parte destes arquivos para recriar os processos e seus respectivos estados.

```
/* funções a serem registradas */
struct file_operations ops = {
  ioctl:          ckpt_ioctl,
  open:           ckpt_open,
  read:           ckpt_read,
  write :         ckpt_write ,
};
/* remove o módulo e o device driver */
void cleanup_module() {
   /* Restaura as system    calls  originais na tabela    sys_call_ table  */
   sys_call_table[__NR_socketcall ] = original_sys_socketcall ;
   sys_call_table[__NR_execve ]     = original_sys_execve ;
   /* remove o device driver*/
   devfs_unregister  _chrdev(0, "/dev/ckptdist ");  }
```

```
/* Initializa o módulo e insere o device driver */
int init_module()   {

   /* Registra o device driver */
   devfs_register_ chrdev(0, "/dev/ckptdist ", &ops);

   /* armazena a system call original para posterior restauração */
   original_sys_socketcall  = sys_call_table[__NR_socketcall ];
   original_sys_execve      = sys_call_table[__NR_execve ];

   /* atribui a "system call criada" para a tabela causando
  o "desvio" desejado*/
   sys_call_table[__NR_socketcall ] = my_socketcall ;
   sys_call_table[__NR_execve ]     = my_execve ;  }
```

Figura 3: Código-fonte para interceptação de mensagens

Havendo a necessidade de efetuar um *checkpoint*, o processo é colocado em estado de *wait* enquanto o seu estado é obtido. Em seguida, antes de salvar seu estado em disco, o processo é retirado do estado de *wait* através da restauração do seu estado de execução original. Este controle é necessário para evitar inconsistências durante o salvamento do estado, uma vez que o processo pode gerar algum evento durante o processo de salvamento. De forma semelhante, após ter sido restaurado um estado previamente salvo, o processo somente retorna ao seu estado original de execução após ser liberado pelo módulo `/dev/ckptdist`. Este procedimento também é executado quando o processo é iniciado pelo usuário, antes de gerar qualquer evento.

Testes realizados demonstraram que o salvamento e restauração de estados e a interceptação de mensagens não sobrecarregam significativamente a execução das aplicações. E também garantem a execução transparente do algoritmo, sem a necessidade de intervenção do usuário para ajustes ou reinicialização de processos.

5.4. Demais mecanismos

Implementados em nível de usuário, os demais mecanismos do algoritmo foram concentrados no processo CKPT. Este processo é composto por três *threads* concorrentes. A primeira encarrega-se da leitura do *buffer* de mensagens do módulo `/dev/ckptdist`. Para cada mensagem lida deste *buffer*, ou é enviada uma resposta de confirmação de recebimento, no caso de mensagens recebidas, ou a mensagem lida é transferida para outro *buffer* de mensagens enviadas pela aplicação, mas que ainda não receberam confirmação por parte do emissor.

A segunda *thread* encarrega-se de receber mensagens através de uma porta específica. Estas incluem mensagens de controle do algoritmo e confirmações de recebimento de mensagens dos demais processos e atualização do *buffer* de mensagens enviadas, solicitando a retransmissão das não comfirmadas e cujo prazo de resposta (*timeout*) expirou. As mensagens de controle do algoritmo são simplesmente entregues para outras funções do algoritmo para serem processadas.

A terceira *thread*, considerada a função `main` do programa, implementa o fluxo de controle central do algoritmo. Nela estão reunidas operações como o controle da inicialização dos mecanismos do algoritmo, o protocolo de comunicação em grupo para a execução das rodadas, a detecção de erros, a restauração do sistema para um estado an-

terior ao de alguma falha, a geração de informações de dados para avaliações estatísticas, etc. As três *threads* acessam as mesmas estruturas de dados do algoritmo e o acesso concorrente é garantido através do uso de semáforos. Está em estudo a implementação de uma quarta *thread*, responsável pela geração de informações de dados estatísticos. Dessa forma, a *thread* principal ficaria liberada desta operação. A coleta destes dados poderia ser efetuada por um processo centralizado, cabendo a ele a tarefa de formatar as informações coletadas de todos os processos CKPT participantes do algoritmo.

Os mecanismos implementados no processo CKPT ainda estão em fase de refinamento. É necessária ainda a realização de testes de prova para detectar possíveis falhas de implementação. Esta etapa depende da identificação de aplicações adequadas para a realização destes testes. A partir da certificação da implementação, espera-se partir para a última fase deste trabalho, a qual envolve a avaliação de desempenho do algoritmo baseada nos dados coletados em testes com aplicações reais.

6. Conclusão e trabalhos futuros

Foram descritos, neste artigo, os passos para a implementação de recuperação por retorno baseado em *checkpointing* em sistemas distribuídos assíncronos, sem recorrer ao uso de infra-estrutura especializada. A implementação atual apresenta limitações oriundas do algoritmo e da própria implementação. Limitações do algoritmo envolvem a necessidade de uso de mensagens de controle; uso de um coordenador e, possivelmente, de um algoritmo de rotação de coordenadores; necessidade de adicionar um índice em cada mensagem da aplicação; o algoritmo força todos os processos envolvidos a estabelecerem novos pontos de recuperação, mesmo que não tenham trocado mensagens. As limitaçõe da aplicação envolvem a necessidade de modificações de bibliotecas, o uso com aplicações baseadas na troca de mensagens UDP e finalmente a utilização do sistema operacional específico, o que compromete a portabilidade.

Este trabalho diferencia-se por usar um algoritmo provado formalmente e pelo objetivo de fornecer *checkpointing*/recuperação de forma transparente aos programadores das aplicações. Esta transparência poderá ser alcançada na medida que não será requerida a alteração do código fonte das aplicações nem do sistema operacional.

Com relação à especificação formal, observou-se ser uma ferramenta poderosa pois auxilia significativamente a tarefa de implementação, uma vez que todos os detalhes necessários estão descritos. Além disso, caso o algoritmo tenha sido provado formalmente, pode-se utilizar, com segurança, qualquer tipo de implementação para as situações que não aparecem na especificação.

Apesar dos autores de bibliotecas de uso geral, disponíveis na literatura, afirmarem a simplicidade do uso de seus códigos, a realidade foi outra. A reutilização não foi trivial e demandou um estudo aprofundado de cada biblioteca de maneira que fosse possível identificar as características e limitações de cada uma. Percebeu-se que o salvamento de estado não é uma tarefa trivial. Além das variáveis de cada processo, que estão em memória, é necessário salvar e reestabelecer o estado das conexões de comunicação. Esta tarefa não foi implementada em nenhuma das bibliotecas estudadas, devido a complexidade exigida.

No caso específico do Linux, descobriu-se que a intercepção de chamadas de sis-

tema é uma tarefa simples e de baixo impacto no desempenho. Este mecanismo foi fundamental para se obter uma implementação do algoritmo de recuperação em que a aplicação não necessitasse ser alterada.

A partir do trabalho inicial de implementação aqui realizado, e das características de desempenho observadas em futuros experimentos, será possível sugerir-se melhorias buscando o refinamento do algoritmo. Mecanismos como injeção de falhas, soluções possíveis para o armazenamento de *checkpoints* em memória "objetivamente estável", implementação em outros sistemas operacionais e implementação totalmente em nível de *kernel*, bem como comparações com outras implementações disponíveis, podem ser extensões possíveis ao presente trabalho.

Referências

Bar, M. (2000). *Linux Internals*. McGraw-Hill, New York.

Beck, M., Böhme, H., Dziadzka, M., Kunitz, U., Magnus, R., and Verworner, D. (1999). *Linux Kernel Internals*. Addison Wesley, Harlow, 2nd edition.

Cechin, S. L. (2002). *Protocolo de Recuperação por Retorno, Coordenado, não Determinístico*. Tese (Doutorado em Ciência da Computação), UFRGS, Porto Alegre.

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

Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408.

Fontoura, A. B. (2002). *Avaliação de Abordagens para Captura de Informações da Aplicação*. Dissertação (Mestrado em Ciência da Computação), UFRGS, Porto Alegre.

Jalote, P. (1994). *Fault Tolerance in Distributed Systems*. Prentice-Hall, New Jersey.

Lamport, L. (1994). The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923.

Nebbet, G. (2000). *Windows NT/2000 Native API Reference*. Macmillan Techinical Publishing, Indianapolis.

Rubini, A. (1999). *Linux Device Drivers*. Market Books, São Paulo.

Sankaran, S., Squyres, J. M., Barrett, B., Lumsdaine, A., Duell, J., Hargrove, P., and Roman, E. (2003) *The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing*. In Proceedings, LACSI Symposium, Sante Fe, New Mexico, USA.

Stones, R. and Matthew, N. (1999). *Beginning Linux Programming*. Wrox Press, Birmingham, 2nd edition.

Zhong, H. and Nieh, J. (2001). Crak: Linux checkpointing/restart as a kernel module. Technical Report CUCS-014-01, Department of Computer Science, Columbia University, Columbia USA.

Programando um Subsistema Síncrono para Suporte a Mecanismos Eficientes de Tolerância a Falhas

Andrey E. M. Brito , Francisco V. Brasileiro

Coordenação de Pós-graduação em Informática
Universidade Federal de Campina Grande
Laboratório de Sistemas Distribuídos
Av. Aprigio Veloso, 882, Bodocongó, CEP 58.109-970
Campina Grande, Paraíba, Brasil

{andrey,fubica}@dsc.ufcg.edu.br

Abstract. *In this work we propose the design and implementation of a wormhole - a synchronous subsystem - to be appended to an asynchronous system to allow the solution of fault-tolerant distributed problems that otherwise would have no deterministic solution in a pure asynchronous system. The wormhole architecture encompasses basic services such as clock synchronization and node level failure detection, as well as a programming interface that allows the deployment of specialized synchronous services. One of these services is presented to illustrate the use of the wormhole by an application.*

Resumo. *Nesse trabalho nós apresentamos o projeto e a implementação de um subsistema síncrono (wormhole) a ser incorporado em um sistema assíncrono, a fim de viabilizar a solução de problemas distribuídos tolerantes a falhas, que de outra maneira não teriam solução determinística em um sistema puramente assíncrono. A arquitetura do wormhole incorpora serviços básicos, como sincronização de relógios e detecção de falhas de máquinas do sistema, como também oferece uma interface de programação que permite a instalação de serviços síncronos especializados. Um desses serviços é apresentado para ilustrar a utilização do wormhole por uma aplicação.*

1 Introdução

A maioria das infraestruturas disponíveis para implantação de aplicações distribuídas são caracterizadas pela ausência de limites superiores conhecidos nos atrasos de transmissão de mensagens e de escalonamento de processos, *i.e.* elas são sistemas assíncronos. O resultado apresentado por Fischer *et al.* [Fischer et al., 1985] prova que é impossível atingir consenso [Chandra and Toueg, 1996] (requisito básico para diversos mecanismos para tolerância a falhas) em um sistema distribuído assíncrono sujeito a faltas. Este resultado pode ser resumido da seguinte forma: devido às incertezas no envio e entrega das mensagens, é impossível distinguir entre um processo que falhou e um que está muito lento.

Por outro lado, os sistemas síncronos garantem um limite máximo de tempo nas transmissões de mensagens e de escalonamento de processos, permitindo de forma mais simples soluções para o problema do consenso. A maior parte dos sistemas

práticos não são assíncronos puros, tampouco são síncronos. Eles possuem algum grau de sincronismo. Desta forma, foram propostos vários modelos de sistemas que adicionam algum sincronismo ao modelo assíncrono puro de forma a melhor representar os sistemas práticos. Alguns destes modelos permitem soluções deterministas para o problema do consenso [Cristian and Fetzer, 1999, Verissimo and Almeida, 1995, Chandra and Toueg, 1996].

Entre estes modelos, o modelo assíncrono com detectores de falhas de Chandra e Toueg [Chandra and Toueg, 1996] tem recebido bastante atenção. Isto se deve ao fato de nenhuma consideração sobre os tempos de comunicação e de escalonamento ser feita na construção das aplicações. O sistema é modelado como puramente assíncrono e o sincronismo necessário para resolver problemas como o consenso é abstraido pelo detector de falhas que fornece uma interface e comportamento bem definidos. Desta forma, uma aplicação desenvolvida para este modelo ganha portabilidade, pois independentemente de que componentes do sistema possuem sincronismo, a aplicação não será afetada, apenas a implementação do detector de falhas mudará.

A semântica do serviço de detecção é caracterizada através da definição de duas propriedades básicas: i) abrangência, que determina o mínimo de processos cujas falhas deverão ser assinaladas através de uma suspeição; e ii) exatidão, que limita as falsas suspeições sobre processos que não falharam. Graduando os níveis de abrangência e exatidão, Chandra e Toueg criaram oito classes de detectores [Chandra and Toueg, 1996].

Resolver o problema do consenso utilizando um detector de falhas requer um nível de sincronismo que pode ser descrito da seguinte forma: (*abrangência forte*) após um tempo, todo processo que falha é permanentemente suspeitado por todos os processos corretos; (*exatidão forte consequente*) após um tempo, ao menos um processo correto jamais será erroneamente suspeitado. Essas propriedades definem um detector de falhas conhecido como $\diamond S$ [Chandra and Toueg, 1996] e permitem ao detector suspeitar de processos que não falharam, contanto que em algum momento, deixem de suspeitar erroneamente de algum processo que permaneceu correto.

No entanto, existem problemas que são mais complexos que o problema do consenso e não toleram suspeitas incorretas (por exemplo, a eleição [Sabel and Marzullo, 1995]). Além disso, melhor desempenho pode ser alcançado se os protocolos não precisarem considerar suspeitas incorretas. Para estas situações, entre as classes de detectores de falhas propostos em [Chandra and Toueg, 1996], a classe P (Perfeito) é a mais forte delas e satisfaz as seguintes propriedades: (*abrangência forte*) após um tempo, todo processo que falha é permanentemente suspeitados por todos os processos corretos; (*exatidão forte*) nenhum processo correto é suspeitado antes que falhe.

Implementar um detector de falhas perfeito requer um sistema síncrono [Larrea et al., 2001]. Para contornar as limitações de um sistema síncrono, algumas abordagens foram propostas [Veríssimo and Casimiro, 2002, Fetzer, 2003, Oliveira et al., 2003]. Essencialmente elas assumem um sistema onde ao menos uma pequena parte do sistema irá funcionar de forma síncrona, independente de quão assíncrono seja o resto do sistema. A implementação do detector de falhas perfeito na parte síncrona torna-se então um problema muito simples.

Duas grandes dificuldades existem na implementação de um subsistema síncrono

a ser acoplado a um subsistema assíncrono: a dependência de máquinas comuns para a realização de tarefas síncronas e a interface com a parte assíncrona do sistema. Alguns trabalhos propõem mecanismos para contornar estas dificuldades [Casimiro et al., 2000, Fetzer, 2003, Oliveira et al., 2003]. Esses trabalhos lidam com a imprevisibilidade dos sistemas assíncronos através de pedaços do mesmo que são mais confiáveis (do ponto de vista de sincronismo) e que monitoram o resto do sistema. Entretanto, esses trabalhos fornecem serviços simples e previamente implantados que não precisam lidar com alguns aspectos da interface entre os ambientes síncrono e assíncrono.

Os sistemas baseados em sistemas assíncronos (ou parcialmente síncronos) que poderiam, apenas quando necessário, tirar proveito de um recurso escarço, um subsistema síncrono, são denominados sistemas híbridos. O subsistema síncrono, por sua vez, é chamado de *wormhole* [Veríssimo, 2003] e é simples o suficiente para que sua implementação confiável seja possível.

Finalmente, uma vez implementado o subsistema síncrono, resta a seguinte dúvida: *se sistema síncronos são mais poderosos que detectores de falhas perfeitos na solução de problemas, será que subsistemas síncronos não poderiam ser utilizados para criar abstrações mais fortes que os detectores de falhas perfeitos?* De fato, existem problemas que não podem ser resolvidos em sistemas assíncronos mesmo quando equipados com detectores de falhas perfeitos e requerem um sistema síncrono [Charron-Bost et al., 2000]. É um problema em aberto determinar se existem abstrações que possibilitam a um sistema assíncrono resolver problemas que exigem sistemas síncronos (ou ao menos problemas que exigem mais que um detector de falhas perfeito). Então, equipar sistemas assíncronos com componentes que implementem essas novas abstrações pode habilitá-los a resolver mais problemas que os solúveis utilizando detectores de falhas perfeitos. Além disso, outros problemas já solúveis podem ter seu desempenho melhorado, seja no tempo de terminação ou no número de falhas suportadas.

Nesse trabalho nós apresentamos uma arquitetura para a implementação de um *wormhole* a ser acoplado a um sistema assíncrono. Esta arquitetura fornece além dos serviços básicos de sincronização de relógios e detecção de falhas de nós, providas por outras soluções relatadas na literatura, uma interface de programação que permite a instalação de serviços síncronos especializados. A implementação da abstração de um *compilador de estados globais* usando o *wormhole* é apresentada como uma forma de exemplificar esta interface de programação. O restante deste artigo está estruturado da seguinte forma. A Seção 2 apresenta as principais idéias do projeto de um sistema híbrido. A Seção 3 detalha algumas considerações feitas. Os aspectos de *hardware* e software no projeto são discutidos na Seção 4. A forma de uso do *wormhole* para a implementação de serviços e protocolos é apresentada na Seção 5. Finalmente, conclusões e trabalhos futuros são discutido na Seção 6.

2 Arquitetura do Sistema

O sistema híbrido é formado por uma parte assíncrona e uma parte síncrona e o seu objetivo é possibilitar que um número limitado de serviços simples (por exemplo, um detector de falhas) possa ser implementado na porção síncrona enquanto as aplicações executam na porção assíncrona do sistema. Para possibilitar a implementação de serviços síncronos,

a porção síncrona fornece duas funcionalidades: um serviço de comunicação com garantias de limite máximo de tempo para a entrega de mensagens e um serviço que força o escalonamento dos serviços síncronos nos momentos em que isto é necessário.

A porção assíncrona do sistema é então formada por um conjunto de máquinas interligadas por uma rede local de acesso livre. A porção síncrona, por sua vez, corresponde a um conjunto de dispositivos conectados a essas máquinas e uma rede local privativa que interconecta os dispositivos. Os dispositivos, máquinas associadas e suas redes estão ilustrados na Figura 1, onde um nó é constituído de um dispositivo e uma máquina associada.

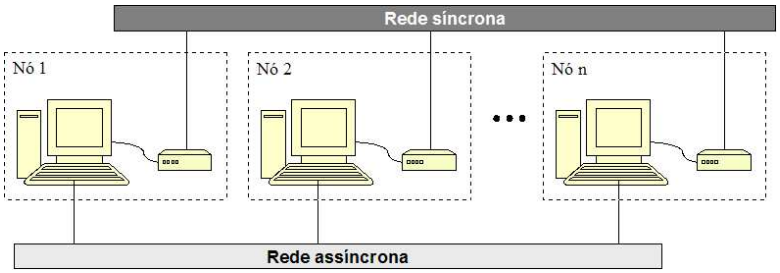

Figura 1: Arquitetura básica

Para que o subsistema síncrono seja utilizado pelo sistema assíncrono *drivers* e serviços são implementados nas máquinas, conforme ilustrado na Figura 2. O dispositivo é acessado exclusivamente pelo seu *driver*. Este controla o acesso, pelos serviços, às funcionalidades do subsistema síncrono. Por fim, as aplicações obtém seus benefícios através de interfaces específicas para cada serviço. Esta organização tem a função primordial de possibilitar a interface entre os subsistema síncrono e o sistema assíncrono na medida que: (1) o subsistema síncrono não pode atender às requisições esporádicas, e talvez concorrentes, das aplicações assíncronas; (2) as aplicações assíncronas podem não conseguir consumir as informações geradas pelo subsistema síncrono na mesma velocidade que são produzidas.

Desta forma, esta organização isola o subsistema síncrono das possíveis interferências das aplicações assíncronas, que poderiam fazer requisições em taxas imprevisíveis e acima da capacidade do sistema síncrono. Por outro lado, os serviços a serem oferecidos às aplicações que executam no sistema assíncrono devem ser construídos de tal forma a tolerar possíveis perdas da informação gerada pelo sistema síncrono (quando a aplicação não consegue executar rápido o suficiente para consumir a informação sendo produzida).

O funcionamento do sistema está ilustrado na Figura 3 e compreende os seguintes passos: (1) um protocolo híbrido se inscreve junto ao *wormhole* e solicita sua execução, caso existam recursos suficientes livres (como largura de banda da rede síncrona), a solicitação é aceita; (2) periodicamente, o dispositivo sinaliza que o protocolo deve ser escalonado; (3) o *driver* força o escalonamento da tarefa que implementa o protocolo; (4) uma vez em execução, o protocolo tem acesso ao *buffer* de envio e recepção da rede síncrona e pode enviar mensagens de comprimento limitado e receber mensagens endereçadas a ele (as mensagens circulam pela rede síncrona e têm um limite máximo para o tempo de entrega); (5) periodicamente e em momentos determinados pelo dispo-

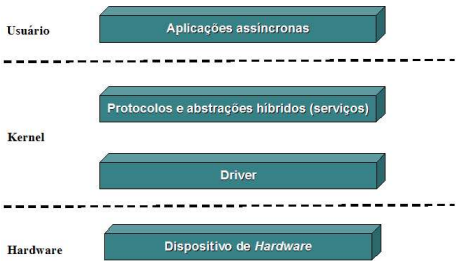

Figura 2: Divisão em camadas do sistema

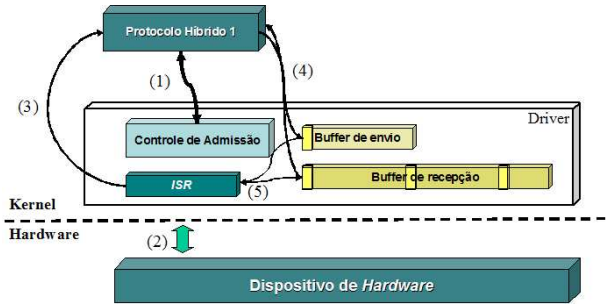

Figura 3: Funcionamento do sistema

sitivo, o *buffer* de envio é repassado ao dispositivo e o *buffer* de recebimento é lido do dispositivo pelo *driver*.

3 Considerações

Como a rede local não garante limites de tempo nas comunicações, ela será chamada de rede assíncrona. O dispositivo de *hardware*, referenciado como "o dispositivo", tem acesso a uma rede privativa. Esta rede garante limites nos tempos de comunicação, desta forma será chamada de rede síncrona. As mensagens serão chamadas de mensagens síncronas ou mensagens assíncronas de acordo com a rede na qual trafegam.

Na rede síncrona, toda comunicação é realizada através de difusões e de forma periódica. Em cada período cada um dos os nós pode transmitir uma mensagem. Assume-se que estas difusões são atômicas, *i.e.* ou o enviador foi bem sucedido em difundir a mensagem e portanto todos os nós corretos irão recebê-la em um intervalo de tempo limitado, ou o enviador falhou durante (ou antes) da difusão e nenhum nó vai receber a mensagem.

Quanto ao processamento, o dispositivo é síncrono por construção. Além disso, ele é capaz de perceber quando a parte síncrona (*driver* e serviços síncronos) executando no PC sofre uma falha de desempenho. Estas falhas de desempenho podem ser conseqüência de uma falha de desempenho de algum serviço do *wormhole* ou do próprio sistema operacional. Em ambos os casos o mecanismo aciona um mecanismo de segurança, que força todo o nó a entrar em um estado seguro (por exemplo, falhar por parada ou reiniciar). Desta forma, falhas em um nó não contaminam os outros nós através da rede síncrona.

4 Implementação

4.1 Interface de *harwdware*

O dispositivo é construído de forma que a parte de *software* do *wormhole* tenha alguns recursos para garantir a comunicação com garantias de tempo e o escalonamento das tarefas. Esses recursos são acessados através da comunicação com uma interface de *hardware*[1] e são os seguintes:

[1]Nossas implementações atuais utilizam a porta paralela ou a porta serial do PC.

- **Interrupção programada**: O dispositivo interrompe periodicamente a máquina, a cada interrupção seu *driver* é executado e algumas pequenas tarefas podem ser executadas de forma síncrona.
- **Comunicação com garantia de limite de tempo**: O dispositivo aceita mensagens para serem enviadas em blocos, periodicamente, com garantias de limite de tempo até sua entrega aos nós de destino.
- **Cão-de-guarda**: Um cão-de-guarda é configurado no dispositivo para que quando alguma tarefa que deveria ter sido executada pelo *driver* ou pelos serviços (em conseqüência das interrupções periódicas do dispositivo) sofra uma falha de desempenho ou de parada, o nó seja colocado em um estado seguro e não comprometa as propriedade de segurança do sistema.

Para garantir o sincronismo e a periodicidade na rede síncrona o dispositivo implementa o protocolo TDMA (*Time Division Multiple Access*) proposto por Brito [Brito, 2004]. Este protocol controla o acesso ao canal de comunicação dividindo-o em períodos. Em cada período, todos os nós que fazem parte da rede síncrona possuem uma fatia de tempo. Para implementar o protocolo TDMA, um líder é utilizado para sincronizar todos os nós e para alocar fatias de tempo aos nós que desejam fazer parte da rede. A falha deste líder é tolerada através de um mecanismo que elege automaticamente um dos dispositivos corretos para assumir as responsabilidades do líder.

4.2 Interface do software

A porção de *software* do *wormhole* consiste em módulos carregáveis do sistema operacional Linux [Bovet and Cesati, 2003]. Um módulo é um pedaço de código compilado que pode ser acoplado dinamicamente ao sistema operacional em execução. Esses módulos utilizam as funcionalidades disponibilizadas pelo dispositivo e implementam três serviços básicos. Estes serviços básicos são acessados a partir de funções que eles exportam para o núcleo do sistema operacional. Desta forma, outros módulos do sistema operacional podem acessá-las diretamente e, da mesma forma, podem disponibilizá-las para os processos de usuário a partir de chamadas de sistema, arquivos especiais ou soquetes de comunicação

local. Estes serviços são os seguintes:

- Relógio Global;
- Serviço de detecção de falhas a nível de nós; e
- Controle de admissão.

Além destes, um conjunto de funções de propósito geral foi definido para permitir a recuperação de informações de configuração do *wormhole*. As funções são as seguintes:

- $id\_list \Leftarrow get\_ids()$: retorna uma lista dos identificadores dos nós que fazem parte atualmente da rede síncrona.
- $max\_nodes \Leftarrow get\_max\_nodes()$: retorna o número máximo de nós que podem fazer parte da rede síncrona.
- $max\_delay \Leftarrow get\_max\_delay()$: retorna o tamanho máximo do período TDMA, calculado a partir da quantidade máxima de nós da rede síncrona. O tamanho máximo do período corresponde ao atraso máximo que uma mensagem enviada pela rede síncrona pode sofrer.

Os serviços básicos são detalhados a seguir.

Relógio global sincronizado. As mensagens periódicas no *wormhole* são utilizadas na construção de uma referência de tempo global. Esta referência de tempo pode ser acessado a partir da função exportada para o núcleo:

- $current\_time \Leftarrow get\_global\_time()$: retorna o valor da referência de tempo global.

Serviço de Detecção de Falhas. O serviço de detecção de falhas identifica que nós participantes da rede síncrona estão corretos. Este serviço pode ser acessado a partir das seguintes funções:

- $ip\_list \Leftarrow get\_corrects()$: retorna a lista de IPs dos nós corretos que fazem parte atualmente da rede síncrona.
- $correct \Leftarrow is\_correct(ip)$: verifica se o nó cujo IP é ip faz parte atualmente da rede síncrona.

O controle de admissão. Na prática, sincronismo só pode ser obtido através de acesso controlado. Além disso, para que o sistema síncrono de capacidade limitada possa atender as requisições do sistema assíncrono chegando de forma esporádica, deve haver um intermediário no sistema assíncrono que colete as requisições assíncronas e se comunique de forma controlada com a parte síncrona.

Os serviços básicos não necessitam de um controle de acesso pois o detector de falhas e a referência de tempo global têm como resultado um valor monotônico e somente de leitura que pode ser compartilhado por todas as aplicações interessadas. Entretanto, existirão serviços e protocolos carregados dinamicamente, estes serviços utilizam a banda do *wormhole* para comunicar-se com outras instâncias suas em outros nós da rede síncrona. Desta forma, a quantidade de serviços dinâmicos carregados deve ser limitada. A iniciação destes serviços dinâmicos é gerida pelo controle de admissão. Como a banda é compartilhada entre os serviços dinâmicos e eles precisam ser carregados simultaneamente em todos os nós que participam do serviço, os nós devem entrar em acordo em todas as requisições de iniciação de um serviço.

O serviço de controle de admissão especifica o serviço a ser carregado, uma lista dos participantes e o número mínimo de participantes que precisam confirmar a participação para que o serviço seja carregado. O serviço de controle de admissão é acessado a partir das seguintes funções:

- $return \Leftarrow wh\_subscribe\_service(name, bandwidth, handler)$: cadastra um serviço na lista de serviços localmente disponíveis em um nó do *wormhole*. O parâmetro $name$ corresponde ao nome do serviço, o parâmetro $bandwidth$ corresponde ao número de bytes que o serviço utilizará para o envio de mensagens em cada período do *wormhole* e o parâmetro $handler$ é um apontador para a função responsável pelo serviço (detalhada nas próximas seções). Retorna um número negativo em caso de falha na inscrição.
- $wh\_unsubscribe\_service(name)$: remove um serviço da lista de serviços localmente disponíveis em um nó do *wormhole*.

- $handler \Leftarrow wh\_request\_service(name, parameters, participants, quorum)$: faz uma difusão na rede síncrona, solicitando uma alocação de uma porção do canal síncrono do serviço $name$ nos nós especificados pelo parâmetro $participants$ e solicitando também o carregamento desse serviço. Os parâmetros necessários ao serviços devem ser conhecidos e são encapsulados na cadeia de bytes $parameters$. O parâmetro $quorum$ corresponde ao número mínimo de participantes para que o serviço seja iniciado.
- $wh\_remove\_service(name)$: encerra a execução de um serviço.

5 Usando o sistema implementado

5.1 O Compilador de Estados Globais

O Compilador de Estados Globais (CEG) é um exemplo de implementação de um serviço que utiliza o *wormhole* implementado. O CEG produz uma representação resumida, limitada e consistente dos estados locais de todos os processos que executam um protocolo, na forma de uma seqüência ordenada de "resumos de estados globais" (REGs).

No caso do consenso, o problema escolhido para exemplificar a utilização do CEG, ele fornecerá informações que possibilitam que o protocolo se adapte às variações nos níveis de contenção vividos pelo sistema durante uma execução do protocolo - uma característica que não está presente em nenhum outro protocolo de consenso baseado em detectores de falhas perfeito. Dessa forma, o consenso terminará tão rápido quanto o nó mais rápido consiga difundir sua mensagem de proposta entre os outros nós.

Para resolver o problema do consenso entre um conjunto de n processos $\Pi = \{p_1, p_2, ..., p_n\}$, o CEG provê REGs com a seguinte estrutura:

- *detection_vector*: um vetor de bits com n bits, onde o elemento i representa o estado operacional do processo p_i (inicialmente 0 e ajustado para 1 se p_i falha);
- *reception_matrix*: uma matriz de bits de dimensões $n \times n$ onde o bit $[i, j]$ indica se p_i recebeu uma mensagem do protocolo de consenso vinda de p_j (inicialmente 0 e ajustado para 1 quando uma mensagem é recebida); e
- *consensual_identity*: um campo com $\lceil log_2 n \rceil$ bits[2] que contém a identidade do processo que propôs a mensagem consensual do protocolo (inicialmente $\perp$).

5.1.1 Implementando o CEG

Na implementação do *wormhole* e do CEG detalhada no trabalho de Brito [Brito, 2004], os serviços foram mapeados em arquivos especiais no sistema de arquivos. Para isso foi adicionado um parâmetro $file\_operations$ à função $wh\_subscribe\_service()$. Este parâmetros contém um apontador para as funções que manipulam os acessos a um arquivo especial. Desta forma, as aplicações podem acessar os REGs acessando um arquivo especial diretamente conectado ao CEG. Para acessar o CEG, uma aplicação acessa um arquivo especial de número maior 252 e número menor 31 (criado com $mknod\ wh\_adm\_control\ c\ 252\ 31$)[3]. Esse arquivo está diretamente conectado ao serviço

[2] $\lceil log_2 n \rceil$ é a quantidade de bits necessária para representar um numero x, onde $0 < x < n$.

[3] O número maior e o número menor são dois parâmetros que o Linux usa para identificar os *drivers* associados a um arquivo especial [Bovet and Cesati, 2003]

de controle de admissão. Uma vez aberto o arquivo, a aplicação escreve um pacote contendo o nome do serviço (10 bytes), os parâmetros do serviço (10 bytes), a lista de participantes (10 bytes, limitada atualmente a 10 participantes) e o quórum (1 byte) – bytes não utilizados devem ser preenchidos com o caracter $'\backslash 0'$. Quando o processo da aplicação escreve nesse arquivo especial, ele fica bloqueado enquanto o serviço é iniciado. Depois de iniciado, o processo é desbloqueado e as funções que manipulam acessos a esse arquivo passam a ser as funções do próprio CEG. O CEG tem três funções associadas ao arquivo especial: leitura, escrita e o fechamento do arquivo (já que a abertura do arquivo foi tratada pelo próprio *wormhole*).

A leitura do arquivo especial é feita em blocos que representam REGs. O tamanho de cada REG é proporcional a quantidade de participantes. De forma semelhante, na escrita, cada bloco representa o estado local do processo que participa do protocolo. O estado local consiste no conjunto dos identificadores dos remetentes cujas mensagens do protocolo já foram recebidas. Para encerrar o serviço é necessário que um valor especial seja escrito no arquivo especial antes que ele seja fechado. Caso o arquivo seja fechado sem a escrita deste valor, o serviço continuará em execução até que algum nó encerre o serviço ou que todos os nós falhem. Esse procedimento é necessário pois uma falha no processo pode causar o fechamento acidental do arquivo, o que poderia encerrar o serviço em todos os nós que o executam.

Por baixo do arquivo especial, existe um módulo que implementa o serviço em si. Este módulo é composto de seis funções básicas: a função $init\_module()$, a função $cleanup\_module()$, a função de processamento do serviço, além de outras três funções necessárias para manipular os acessos de leitura, escrita e fechamento do arquivo especial.

A função $init\_module()$ é responsável pelo procedimento de iniciação do módulo e é executada pelo próprio Linux. A função $init\_module()$ de um serviço do *wormhole* realiza as seguintes tarefas: (1) inscreve-se no *wormhole* através da função $wh\_subscribe\_service()$, informando seu nome, a largura de banda necessária (em bytes por período) e o apontador para a função que realiza o processamento de suas mensagens; (2) realiza os procedimentos necessários para interagir com o ambiente assíncrono (por exemplo, alocando números maiores ou especificando a estrutura responsável pelas operações no arquivo especial) e quaisquer recursos que o módulo utilize durante a realização de seus serviços.

A função $cleanup\_module()$ é responsável pela procedimento de finalização do módulo antes de sua remoção da memória e do núcleo. O *wormhole* requer que esta função remova o cadastro do serviço junto ao mesmo. Além disso, esta função deve liberar quaisquer recursos utilizados durante sua execução.

A função de processamento do serviço é uma função que será executada como um *tasklet* pelo *driver* do *wormhole* sempre que um período acaba. Esta função tem acesso aos *buffers* de entrada e de saída do *wormhole*, de forma que ela tem acesso às mensagens recebidas no período anterior e pode escrever mensagens para serem enviadas no período seguinte. No caso do CEG, a função de processamento constrói os REGs a partir do *buffer* de mensagens recebidas (dos outros nós que participam do serviço) e escreve estados locais no *buffer* de mensagens a serem enviadas. Desta forma, em cada máquina, a instância do CEG envia através do *wormhole* apenas seu estado local e recebe através

do *wormhole* os estados locais de todas as máquinas que executam o CEG. De posse de todos os estados locais, cada instância do CEG constrói a matriz $reception\_matrix$ e a partir dela, o identificador $consensual\_identity$. O vetor $detection\_vector$ pode ser construindo simplesmente avaliando se a instância ceg_i executando na máquina i enviou seu estado local durante o último período.

As funções de manipulação dos acessos aos arquivos especiais, são encapsuladas numa estrutura do tipo $file\_operations$ e então atribuídas ao respectivo campo na estrutura $file$ do arquivo aberto. Um acesso $READ$ retorna o REG mais recente, de comprimento igual a soma das estruturas $detection\_vector$, $reception\_matrix$ e $consensual\_identity$ ($n+n^2+\lceil log_2 n \rceil$ bits). Um acesso $WRITE$ deve informar o estado local, ou seja, um vetor de n bits onde o i-ésimo bit indica se o processo em questão recebeu ou não uma mensagem de proposta do consenso do processo p_i. Um acesso $CLOSE$ ativa a função que libera o arquivo e pode encerrar a execução do serviço, desde o encerramento do arquivo tenha sido liberado por um acesso $WRITE$ com o valor especial de encerramento.

5.1.2 Resolvendo consenso com o CEG

O problema do consenso uniforme consiste em cada processo p_i propor um valor v_i e todos os processos que decidem devem decidir por um dos valores propostos. Formalmente, o consenso pode ser definido pelas seguintes propriedades [Charron-Bost et al., 2000]:

- **terminação**, após um tempo finito todo processo correto decide algum valor;
- **integridade uniforme**, todo processo decide apenas uma vez;
- **validade uniforme**, se um processo decide por um valor v, então v foi proposto por algum processo; e,
- **acordo uniforme**, todos os processos que decidem, corretos ou não, decidem o mesmo valor.

O CEG suporta uma família de protocolos de consenso que se diferenciam por dois parâmetros. O primeiro, denominado *quórum*, define o número de processos que são necessários para "eleger" o processo que propôs o valor consensual. Este parâmetro afeta apenas a parte síncrona do protocolo. O valor do quórum é tal que $f + 1 \leq quorum \leq n$. O segundo parâmetro, denominado *proponentes*, define o número de processos que irão propor um valor durante a execução do protocolo. Este, afeta apenas a parte assíncrona do protocolo e o seu valor é tal que $f + 1 \leq proponentes \leq n$. Desta forma, cada membro da família de protocolos de consenso é descrita como *Consenso-CEG(Q,P)* onde Q e P são os parâmetros quórum e proponentes, respectivamente.

A parte assíncrona do protocolo (a aplicação) é estruturada na forma de três tarefas concorrentes. Na primeira tarefa, a tarefa de *proposição*, P processos enviam mensagens para os outros processos contendo suas propostas. A segunda tarefa, a tarefa de *recebimento*, é responsável por receber e armazenar as mensagens de propostas enviadas por outros processos. Ela também notifica o CEG que determinada mensagem foi recebida. A tarefa final, a tarefa de *decisão*, é responsável por detectar que uma decisão pode ser tomada e que a execução do protocolo está terminada.

A tarefa de decisão é também bastante simples, ela permanece em um laço consultando o CEG. Quando um resumo de estado global é entregue com um campo

$consensual\_identity$ preenchido com o identificador x de algum processo, a tarefa de decisão verifica se a mensagem de p_x já foi recebida. Caso não tenha sido, a tarefa espera até que ela seja recebida. Em ambos os casos, depois de recebida a mensagem de p_x, a tarefa decide pelo valor contido na mensagem e termina a execução enviando a mensagem de p_x para todos os processos corretos que ainda não a receberam. O Algoritmo 1 é o pseudo-código das tarefas que implementam a parte assíncrona do protocolo (a prova formal que este algoritmo resolve o problema do consenso uniforme pode ser encontrada em [Brito, 2004]).

Algoritmo 1 Pseudo-código do protocolo Consenso-CEG(Q, P) executado pelo processo p_i

```
% variáveis compartilhadas
bagOfMessages_i = ∅
decided_i = falso

% Tarefa de proposição
quando execute propose(v_i)
  se i ≤ p então envie m_i(v_i) para todos os processos fim se
fim
||
% Tarefa de recebimento
enquanto não decided_i faça
  quando recebe m_j(v_j) de p_j
    se m_j(v_j) não pertence à bagOfMessages_i então
      adicione m_j(v_j) à bagOfMessages_i
      notifique ceg_i do recebimento de uma mensagem de proposta vinda de p_j
    fim se
  fim
fim enquanto
||
% Tarefa de decisão
enquanto não decided_i faça
  reg = read(CEG)
  x = reg.consensusal_identity
  se x ≠⊥ então
    espera até m_x(v_x) em bagOfMessages_i
    m_x(v_x) = getConsensualMessage(x, bagOfMessages_i) % recupera mensagem de p_x
    envie m_x(v_x) para todo p_k tal que reg.detection_vector[k] = 0 ∧ reg.reception_matrix[k, x] = 0
    decided_i = true
    return(v_x) % decide pelo valor proposto por p_x
  fim se
fim enquanto
```

6 Conclusão

Neste trabalho nós apresentamos como um *wormhole* pode ser construído com o objetivo de possibilitar que serviços simples possam ser implementados na porção síncrona enquanto as aplicações executam na porção assíncrona do sistema.

Como exemplo de um serviço, detalhamos como o *wormhole* foi utilizado para implementar uma nova abstração, o Compilador de Estados Globais (CEG). O CEG é uma abstração mais forte que os detectores de falhas perfeitos e no entanto, sua implementação requer as mesmas considerações que a implementação de um detector de falhas perfeito. Nós discutimos a utilização do CEG para solução do problema do consenso uniforme que atinge o consenso em uma única rodada de comunicação.

Agradecimentos

Os autores agradecem a Walfredo Cirne pelas discussões ao longo do desenvolvimento desse trabalho e aos revisores anônimos, pelos comentários pertinentes. Este trabalho foi parcialmente financiado pelo CNPq (processo 300646/1996-8) e pelo PRH-25/ANP.

Referências

Bovet, D. and Cesati, M. (2003). *Understanding the Linux Kernel*. O'Reilly, 3 edition.

Brito, A. E. M. (2004). Uma arquitetura híbrida para o suporte de protocolos distribuídos tolerantes a falhas. Dissertação de mestrado, COPIN - Universidade Federal da Paraíba, Campina Grande.

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

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

Charron-Bost, B., Guerraoui, R., and Schiper, A. (2000). Synchronous system and perfect failure detector: solvability and efficiency issues. In *Proceedings of the IEEE Int. Conf. on Dependable Systems and Networks (DSN)*, pages 523–532, New York, USA. IEEE Computer Society.

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

Fetzer, C. (2003). Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112.

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

Larrea, M., Fernández, A., and Arévalo, S. (2001). On the impossibility of implementing perpetual failure detectors in partially synchronous systems. In *Brief Announcements 15th Int'l Symp. Distributed Computing (DISC 2001)*.

Oliveira, E. W., Brito, A. E. M., and Brasileiro, F. V. (2003). Projeto e implementação de um serviço de detecção de falhas perfeito. In *Simpósio Brasileiro de Redes de Computadores*, pages 697–712, Natal/RN, Brasil.

Sabel, L. S. and Marzullo, K. (1995). Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Cornell University.

Verissimo, P. and Almeida, C. (1995). Quasi-synchronism: a step away from the tradicional fault-tolerant real-time system models. *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, 7(4):35–39.

Veríssimo, P. (2003). Uncertainty and predictability: Can they be reconciled? *Future Directions in Distributed Computing, Springer Verlag LNCS 2584*, pages 108–113.

Veríssimo, P. and Casimiro, A. (2002). The Timely Computing Base model and architecture. *Transactions on Computers*, 51(8):916–930.

Suporte à Execução Replicada de Serviços Orientados a Prioridade

Marcelo Jorge Aragão[1], **Francisco Brasileiro**[1,2]

[1] Universidade Federal de Campina Grande
[1]Coordenação de Pós-Graduação em Engenharia Elétrica
[2]Coordenação de Pós-Graduação em Informática
Av. Aprígio Veloso, 882, Campina Grande, Paraíba, 58.109-970, Brazil
Tel: (+55) 83 310 1433 Fax: (+55) 83 310 1365

aragao@dee.ufcg.edu.br fubica@dsc.ufcg.edu.br

Abstract. *This paper presents the design and implementation of a service that suports the construction of fault tolerant e-commerce applications that use an adaptive priority-based policy for resource allocation. Active replication is implemented to tolerate failures and* Group Priority Inversions *are avoided.*

Resumo. *Esse artigo apresenta o projeto e a implementação de um* framework *de suporte à construção de aplicações de comércio eletrônico tolerantes a falhas, que utilizam políticas adaptativas de alocação de recursos baseadas em prioridade. A técnica de replicação ativa é implementada para tolerar falhas e o problema de inversão de prioridade em grupo é tratado.*

1. Introdução

Alta disponibilidade é extremamente importante para um número crescente de aplicações distribuídas. Por exemplo, em julho de 1999, o site de leilões virtuais *eBay* ficou cerca de 22 horas fora do ar por problemas de infra-estrutura, o que representou uma perda no seu faturamento estimada entre US$ 3 milhões e US$ 5 milhões [3].

A indisponibilidade de serviços é normalmente causada pela escassez de recursos ou pela ocorrência de falhas. A *superprovisão* [7] e a *alocação de recursos baseada em prioridade* [1, 9, 10] são estratégias utilizadas para contornar o problema de falta de disponibilidade devido à escassez de recursos. A *superprovisão* consiste em estimar a maior carga que pode ser submetida a um sistema e, com base nessa estimativa, dimensionar os recursos do sistema de modo a suportar essa carga. Esse método é caro, pois o sistema pode permanecer a maior parte do tempo submetido a uma baixa carga, subutilizando os recursos dimensionados. Além disso, é difícil identificar corretamente essas estimativas, pois, muitas vezes, a carga submetida a um sistema é imprevisível.

A *alocação de recursos baseada em prioridade* permite que o sistema priorize o acesso e o uso de recursos conforme métricas que objetivam ganhos computacionais [1] ou ganhos financeiros [9, 10]. Essa é uma estratégia mais econômica que a *superprovisão*, pois a quantidade de recursos permanece inalterada. Um estudo realizado em [9, 10] demonstrou a viabilidade de políticas adaptativas de alocação de recursos para *sites* de comércio eletrônico. As políticas analisadas estabelecem prioridades baseadas no perfil do consumidor (freqüente ou ocasional), na duração da sessão e no valor total de compras acumuladas até o momento. O trabalho conclui em seus resultados que há um potencial de ganho em faturamento de até 29% sobre sistemas convencionais

nos momentos de pico [10]. No entanto, essa solução traz consigo um problema encontrado em sistemas com escalonamento baseado em prioridade. Esse problema é conhecido na literatura como problema de *inversão de prioridade* [14].

No nosso contexto, o problema de inversão de prioridade ocorre quando o processamento de uma requisição de alta prioridade é atrasado pelo processamento de uma requisição de baixa prioridade. O tratamento desse problema exige que uma requisição em execução seja suspensa ou abortada, para que o processamento da requisição de mais alta prioridade possa ser iniciado. Esse tratamento é normalmente realizado por um componente de software denominado escalonador de requisições. Dependendo do modo que o processamento da requisição é interrompido, este poderá continuar ou reiniciar sua execução após a conclusão do processamento da requisição de mais alta prioridade.

Além da escassez de recursos, a indisponibilidade de serviços pode também ser causada pela ocorrência de falhas. Falhas são inevitáveis, mas suas conseqüências, ou seja, o colapso do sistema, a interrupção no fornecimento do serviço ou a perda de dados, podem ser evitadas quando técnicas de tolerância a falhas são usadas de forma adequada. As técnicas de tolerância a falhas caracterizam-se pela introdução de redundância de componentes (hardware e/ou software) [8]. Redundância é normalmente introduzida pela replicação de componentes ou serviços [2, 13]. Na replicação, se uma das réplicas não está operacional, outra réplica garante que um determinado serviço seja oferecido. No entanto, replicação requer protocolos que assegurem consistência de estado entre as réplicas. Problemas de inconsistência podem acontecer devido à concorrência de operações de atualização do estado das réplicas (quando dois ou mais clientes concorrentes fazem diferentes requisições ao serviço replicado) ou devido a falhas em nodos.

Duas estratégias bastante utilizadas para se obter redundância são as técnicas de replicação passiva e ativa. Na técnica de replicação passiva, também chamada de *primary/backup* [2], existe uma réplica primária e uma ou mais secundárias. A réplica primária está sempre em execução, pronta para o processamento de requisições, e tem seu estado interno periodicamente salvo em memória estável, como por exemplo em disco. As réplicas secundárias, por sua vez, permanecem inativas e, periodicamente ou na ocorrência de falha, uma delas é promovida a primária e tem seu estado interno atualizado para o último estado interno salvo pela réplica primária anterior. Em casos de falhas, isto significa um retrocesso no estado do sistema, pois as ações realizadas pela réplica primária depois do último ponto de salvaguarda serão executadas novamente. Apesar de possibilitar a continuidade do serviço, o atraso existente nessa técnica pode não ser aceitável em alguns sistemas.

A técnica de replicação ativa [13] é uma técnica de replicação não centralizada em que todas as réplicas de um componente recebem e executam independentemente a mesma seqüência de requisições enviada por seus clientes. Desta forma, se uma réplica falhar, as outras réplicas produzirão as mesmas respostas requeridas sem o atraso de recuperação de estado existente na replicação passiva. A consistência entre as réplicas é garantida desde que as réplicas recebam as mesmas requisições na mesma ordem e produzam as mesmas saídas. Para implementar essa técnica são necessários protocolos que garantam os requisitos de *ordem* e de *acordo*, como especificado em [13]. A principal vantagem dessa técnica está em oferecer tempos de resposta aceitáveis em caso de falhas, em contra partida, essa técnica exige protocolos mais complexos.

Em resumo, o uso de prioridades, aliado à técnica de replicação ativa, pode oferecer condições a muitas aplicações para o provimento de um serviço de alta disponibilidade. Entretanto, a união dessas duas estratégias exige um tratamento especial quanto ao problema de inversão de prioridade em um processamento ativamente replicado. No contexto replicado, o tratamento é bem mais complexo que no contexto não replicado. Considerando que a chegada e o processamento de requisições ocorrem assíncronamente nas réplicas, uma réplica pode não observar os mesmos

casos de inversão de prioridade que outra. Verifica-se nesse contexto que os casos de inversão de prioridade não poderão ser tratados isoladamente nas réplicas, caso contrário, as requisições poderão ser processadas em ordem diferentes e inconsistências poderão ocorrer nos resultados das mesmas.

A solução do problema de inversão de prioridades no contexto de um grupo de processos realizando um processamento ativamente replicado foi proposta em [15]. O conceito de *inversão de prioridade em grupo* foi introduzido pela primeira vez no mesmo trabalho. Informalmente, uma *inversão de prioridade em grupo* ocorre quando casos de inversão de prioridade (*local*) são detectados em muitas réplicas.

Nesse artigo nós apresentamos o projeto e a implementação de um *framework* para suportar o desenvolvimento de aplicações de comércio eletrônico tolerantes a falhas que utilizam políticas adaptativas de alocação de recursos baseadas em prioridade. A técnica de replicação ativa é implementada para tolerar falhas. O *framework* soluciona o problema de inversão de prioridade em grupo e se baseia no estudo realizado em [15]. Mais especificamente, no nosso contexto o *framework* é uma camada de software que possibilita a comunicação entre aplicações clientes e aplicações servidoras (servidor de aplicação e banco de dados). Esse *framework* deve possuir um protocolo escalonador que permita o processamento ativamente replicado de requisições segundo as prioridade dos usuários do sistema.

O restante do artigo está organizado da seguinte forma. A Seção 2 apresenta o modelo de sistema considerado e define o problema de inversão de prioridade em grupo. A Seção 3 apresenta a arquitetura do *framework*, enquanto que a Seção 4 discute aspectos da utilização do mesmo pelas aplicações. Na Seção 5 é feita uma avaliação de desempenho do protótipo implementado. A Seção 6 conclui o artigo com os nossos comentários finais.

2. Modelo do Sistema e Definições

2.1. Modelo do Sistema

Consideramos o modelo de sistema distribuído assíncrono, onde não há limites conhecidos nos tempos de transmissão de mensagens nem nas velocidades de processos. Um sistema distribuído é formado por diversos nodos autônomos que são conectados por uma rede de comunicação. Os nodos não possuem memória compartilhada e se comunicam por troca de mensagens [8]. Adotamos o modelo de falha em que os processos podem falhar por parada [8]. Além disso assumimos que o sistema é equipado com *detectores de falhas não confiáveis* da classe $\diamond S$, conforme definidos em [5].

2.2. Inversão de Prioridade em um Grupo de Réplicas

O escalonamento de requisições orientado a prioridade define a ordem em que um conjunto de requisições é processado. Esse tipo de escalonamento é ilustrado na Figura 1.

Na Figura 1 é apresentado um escalonador com 2 listas (*Lista 1 e Lista 2*). Cada lista armazena requisições que possuem uma determinada prioridade. As requisições são representadas por $\mathbf{X_n}$, onde *n* representa um identificador único da requisição, e suas prioridades são representadas por um retângulo que, dependendo da prioridade da requisição, podem ser cinza (*prioridade 2*) ou branco (*prioridade 1*). O armazenamento de novas requisições em cada lista segue uma ordem FIFO segundo suas prioridades. O processamento ocorre da seguinte maneira: o escalonador, representado por linhas tracejadas, escolhe a lista de maior prioridade e despacha a primeira requisição da direita para esquerda da lista. Quando não houver mais requisições na lista, o

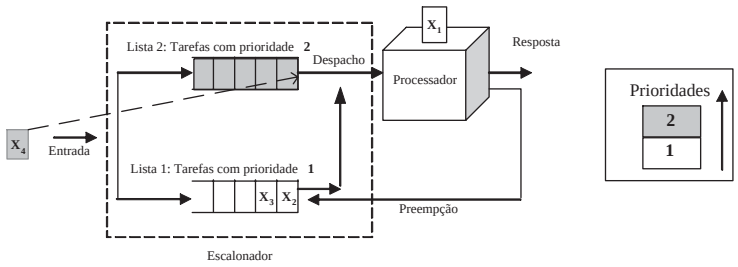

Figura 1: Inversão de prioridade

escalonador passa a escolher as requisições da lista seguinte por ordem de prioridade, e assim por diante.

Quando a entrada de requisições no escalonador ocorre dinamicamente, pode acontecer de uma requisição de prioridade alta ficar esperando a conclusão do processamento de uma outra requisição de prioridade mais baixa. No exemplo da Figura 1 vê-se que o processador está ocupado com requisição $\mathbf{X_1}$ de prioridade ***1*** quando a requisição $\mathbf{X_4}$ de prioridade ***2*** é recebida. Como *Prioridade* ($\mathbf{X_1}$) < *Prioridade* ($\mathbf{X_4}$), observa-se que o processamento da requisição $\mathbf{X_1}$ está atrasando o processamento da requisição $\mathbf{X_4}$, evidenciando portanto, um caso de inversão de prioridade.

Para tratar esse tipo de problema, é preciso um mecanismo capaz de detectar as ocorrências de inversão de prioridade, de forma a suspender ou abortar o processamento de uma requisição de baixa prioridade para dar lugar ao processamento de uma requisição de alta prioridade. Uma requisição que teve seu processamento interrompido deve ser reordenada e ter seu processamento reiniciado a partir do último estado anteriormente salvo. Seu reinício irá depender de sua posição em uma das listas de requisições pendentes.

O escalonamento em um sistema com processamento ativamente replicado exige que todas as réplicas possuam uma mesma visão sobre um conjunto de requisições. No escalonamento orientado a prioridade, essa lista precisa além de respeitar as prioridades das requisições, estar ordenada da mesma maneira em todas as réplicas. No modelo considerado não existe sincronização entre os instantes de tempo em que as requisições são recebidas, tornadas prontas para processamento ou processadas nas diferentes réplicas. Desse modo, uma réplica não pode considerar casos de inversão de prioridade local sem antes analisar o estado global das outras réplicas, pois os mesmos casos de inversão de prioridade podem não ocorrer em todas as réplicas.

A Figura 2 ilustra o problema de inversão de prioridade em um sistema com processamento ativamente replicado. Duas réplicas processam independentemente suas requisições utilizando os mesmos critérios explanados na Figura 1. No momento que a requisição $\mathbf{X_4}$ de prioridade ***2*** é escalonada na lista *2* de ambas as réplicas, o processador da réplica *B* encontra-se ocupado com o processamento de uma requisição que já foi concluído na réplica *A* (requisição $\mathbf{X_1}$ de prioridade ***1***). Embora as duas réplicas agora possuam as mesmas listas de requisições, elas não observam os mesmos casos de inversão de prioridade. A requisição $\mathbf{X_1}$ em processamento na réplica *B* possui prioridade menor que a nova requisição $\mathbf{X_4}$, ou seja, *Prioridade* ($\mathbf{X_1}$) < *Prioridade* ($\mathbf{X_4}$). Por sua vez, a réplica *A* encontra-se com o processador livre para processamento. Assim, o problema de inversão de prioridade é identificado somente na réplica *B*.

Para evitar e tratar esse problema no contexto de grupo, existem duas alternativas: realizar um retrocesso (*rollback*) da requisição já completada pela réplica *A* e em seguida reordenar as listas nas duas réplicas; ou ignorar o caso de inversão de prioridade na réplica *B* baseando-se em algumas premissas nas réplicas.

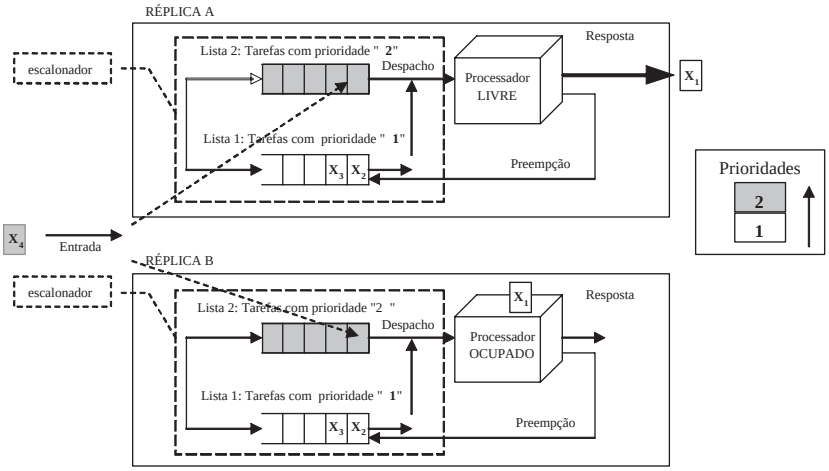

Figura 2: Inversão de prioridade em grupo

A escolha da alternativa mais apropriada irá depender de um referencial em comum entre as réplicas (*Group Execution Progress - GEP*). Esse referencial definirá como as novas requisições deverão ser ordenadas e será usado para decidir se uma requisição deverá ou não sofrer retrocesso. GEP é calculado a partir do progresso de execução individual de cada réplica (*Local Execution Progress - LEP*), ou seja, da última requisição já processada por uma determinada réplica.

O escalonamento de requisições pode seguir uma abordagem pessimista ou otimista. Na abordagem pessimista, mesmo que não ocorra casos de inversão de prioridade, ocorre um bloqueio na lista de requisições até que o acordo (ordenação total ou cálculo de GEP) seja concluído. Dessa forma mesmo que um processador esteja livre, ele precisa esperar pelo fim do acordo. Dessa forma retrocessos jamais ocorrem. Na abordagem otimista, é possível realizar antecipadamente algum processamento, antes de se obter o resultado final do escalonamento. Contudo o processamento de uma requisição poderá ser eventualmente desfeito em casos de inversão de prioridade. Como conseqüência, para garantir consistência entre as réplicas, os estados e as respostas das requisições sujeitas a retrocesso devem ser armazenados para serem entregues aos respectivos clientes quando o processamento das requisições não puder ser mais desfeitos.

O conceito de inversão de prioridade em grupo (*Group Priority Invertion - GPI*) foi introduzido pela primeira vez em [15], onde o problema de inversão de prioridade em uma única réplica (*Local Priority Inversion - LPI*) foi estendido a um grupo de processadores que realizam um processamento ativamente replicado. Em [15] foi proposto um protocolo escalonador de requisição, orientado a prioridade, que garante ordenação total, evita inversão de prioridade em grupo e segue uma abordagem otimista. Uma inversão de prioridade em grupo irá ocorrer sempre que o processamento de uma requisição replicada de alta prioridade for atrasado por atividades de menor prioridade. Assumindo um modelo de falhas por parada, o processamento de uma requisição replicada requer que no mínimo um processo gere uma resposta válida. Isso quer dizer que uma requisição não sofrerá inversão de prioridade em grupo, se for possível garantir que pelo menos uma resposta a essa requisição será gerada sem qualquer atraso.

Para o cálculo de GEP em [15], as réplicas executam um protocolo de acordo onde cada réplica propõe um valor EP (*Execution Progress*), onde EP = max(LEP, GEP). Esse protocolo utiliza uma função que é aplicada a todos os EPs obtidos. Considerando *f*, como o número máximo de réplicas que podem falhar sem afetar a disponibilidade do serviço, essa função é calculada em dois passos: primeiro, o subconjunto contendo os $f + 1$ maiores valores é determinado. Em seguida, o valor de GEP é obtido, extraindo o menor valor contido no conjunto de $f + 1$ valores. A seleção dos $f +$

1 valores oferece um avanço mais rápido de GEP e a seleção do menor valor contido no conjunto $f + 1$ garante que, no mínimo, um desses valores foi fornecido pelo módulo escalonador de uma réplica que não falhará. Dessa forma, o valor de GEP assegura que o envio de respostas aos clientes só ocorrerá quando LEP $\leq$ GEP. Assim, mesmo que ocorra uma falha em uma réplica, os clientes jamais receberão respostas inconsistentes de uma mesma requisição.

3. Arquitetura do *framework*

3.1. Estrutura Interna

A arquitetura do EROPSAR (**E**scalonador de **R**equisição **O**rientado a **P**rioridade para **S**erviços **A**tivamente **R**eplicados) é formada por 5 componentes identificados na Figura 3 pelos retângulos em cinza. Descrevemos como eles interagem uns com os outros para acessar um recurso remoto a partir de uma requisição enviada por uma aplicação cliente.

Uma *aplicação cliente* envia suas requisições e recebe seus resultados através do componente *interceptador*[1]. Antes do envio da requisição ao *interceptador*, a requisição deverá conter as seguintes informações: prioridade, recurso remoto, operações e parâmetros. O componente *interceptador* reside no lado cliente da aplicação e recebe a requisição cliente e a difunde nas réplicas de um servidor replicado (a Figura 3 ilustra apenas uma réplica, contudo a mesma estrutura se aplica às outras réplicas com setas partindo do mesmo *interceptador*). Após receber o resultado do processamento, o *interceptador* envia a primeira resposta de uma requisição ao cliente e descarta as respostas duplicadas que possam ser recebidas.

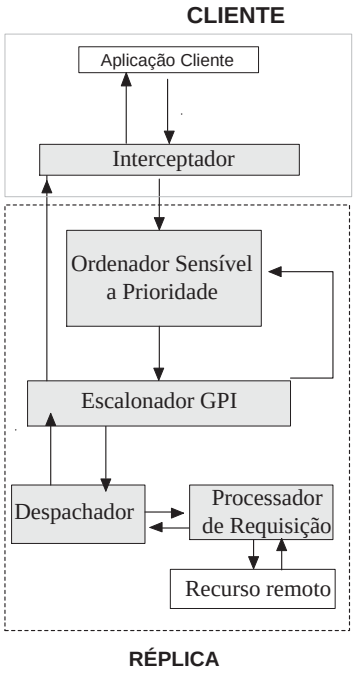

Figura 3: Modelo Estrutural do EROPSAR

O componente *Ordenador Sensível a Prioridade* reside no lado servidor da aplicação e recebe as requisições dos *interceptadores* associados aos clientes e as armazena localmente. Esse componente garante que todos os outros componentes ordenadores não falhos observem a mesma

[1]Esse componente baseia-se no *padrão arquitetural interceptador* (*Interceptor Architectural Pattern*) [12]. Esse padrão permite a adição de serviços a determinado *framework*, de forma transparente, tornando-o extensível.

seqüência de requisições. Ao receber requisições, esse componente decodifica as prioridades das requisições a partir do identificador do cliente e as ordena de acordo com uma prioridade associada no servidor. Uma vez decodificadas e reordenadas, as requisições são enviadas ao *escalonador GPI*.

O componente *Escalonador GPI* recebe do *Ordenador Sensível a Prioridade* as requisições ordenadas e as escalona de modo a evitar o problema de inversão de prioridade em grupo. Esses componentes trabalham em estrita cooperação para tratar esse problema.

O componente *despachador* recebe do *escalonador GPI* a requisição pronta de maior prioridade. Uma vez recebida, ele localiza o *processador de requisição* da aplicação e invoca um procedimento padrão no *processador de requisição* que deverá interpretar o conteúdo da requisição. O *despachador* salva o estado do *processador de requisição* antes do processamento. Esse estado poderá ser utilizado se eventualmente o processamento for interrompido pelo *escalonador GPI* quando uma inversão de prioridade em grupo for detectada. Se esse for o caso, o estado do *processador de requisição* deve ser recuperado após interrupções e eventuais retrocessos. Dessa forma uma requisição que foi interrompida será novamente iniciada a partir de um estado que antecedeu sua interrupção.

A primeira ação do componente *processador de requisição* é localizar o recurso remoto contido na requisição para então invocar as operações contidas na requisição. Em seguida o *recurso remoto* informado na requisição é acessado e realiza a operação solicitada, retornando o resultado ao *processador de requisições*. Por fim esse resultado é retornado ao *despachador*, que por sua vez envia o resultado juntamente com o estado salvo para o *escalonador GPI*.

Conforme colocado anteriormente, uma vez que o identificador do cliente é utilizado como chave para consultar a prioridade de uma requisição no servidor, assumimos que o EROPSAR utiliza o modelo de política de prioridade *client priority propagation* (prioridade propagada pelo cliente) da especificação RT-CORBA [11]. Nesse modelo de política de prioridade, a requisição é executada na prioridade requisitada pelo cliente e é codificada como parte da requisição do cliente. A atualização da prioridade no lado servidor se baseia na interface *PriorityTransform* de RT-CORBA.

3.2. Serviços de Suporte

Os serviços de suporte oferecem um nível de abstração que facilita a interação com o grupo de réplicas. Eles adicionam uma camada de distribuição ao *EROPSAR*, oferecendo acesso a recursos remotos, visão de grupo e primitivas como difusão. Esses serviços funcionam como blocos básicos na construção do protocolo de replicação ativa e do protocolo de acordo (ordenação total e valor do progresso de execução do grupo) necessários no tratamento do problema de inversão de prioridade em grupo. Descrevemos a seguir os serviços de suporte necessários para o *EROPSAR*.

O *EROPSAR* utiliza um *serviço de nomes* para registrar os recursos que o servidor deseja disponibilizar a um cliente. O cliente, por sua vez, adiciona em uma requisição o nome do recurso e a operação desejada com seus respectivos parâmetros, para que possa ser enviada ao servidor. Quando a requisição for despachada, ela deverá ser interpretada de modo a obter o nome do recurso e a operação. Em seguida, de posse dessas informações, o recurso poderá ser acessado através de uma referência obtida a partir de um *serviço de nomes*.

O projeto do *EROPSAR* se baseia em um *serviço de acordo* que precisa realizar a ordenação total de requisições e precisa calcular o *progresso de execução do grupo* para que casos de inversão de prioridade em grupo sejam detectados e tratados. Como chegada de requisições no servidor dá-se de forma contínua, o *serviço de acordo* deve permitir que várias propostas de requisições sejam realizadas mesmo após um consenso ter sido iniciado. Desse modo, muitas requisições podem ser

ordenadas em uma única execução do protocolo. Para que o valor do *progresso de execução do grupo* seja calculado, o *serviço de acordo* deverá prover um protocolo capaz de decidir um valor que é o resultado de uma dada função aplicada aos diversos valores de entrada. Esses dois acordos estão diretamente relacionados para garantir consistência nas réplicas. Devido a essa relação, se o *serviço de acordo* permitir que dois acordos sejam realizados em conjunto (em uma única execução do protocolo de consenso), o serviço poderá ser mais eficiente que se os acordos fossem executados um após o outro.

Para evitar que os processos envolvidos em um acordo fiquem indefinidamente à espera das mensagens de um outro que está muito lento ou que falhou, impossibilitando a decisão do acordo [6], o serviço de acordo é baseado em detectores de falha não confiáveis [5]. Esses detectores fornecem informações de todos os processos suspeitos de terem sofrido uma falha. [5] define várias classes de detectores, todas elas especificadas por duas propriedades básicas: abrangência (*completeness*): que define em que situações os componentes falhos serão detectados; e exatidão (*accuracy*): que limita os erros que o detector pode fazer, ou seja, limita as falsas suspeitas dos processos que não falharam. Suspeitas são implementadas utilizando-se mecanismos de temporizarão (*timeout*) assim: *i)* a detecção de uma falha real pode ficar retardada, e *ii)* um detector de falhas pode cometer erros suspeitando incorretamente de um processo que não falhou.

Dentre as várias classes de detectores, a classe chamada de $\diamond S$ é muito atrativa. Essa é a classe mais fraca de detectores de falhas que permite a solução determinística do problema de consenso em sistemas assíncronos [4], desde que a maioria dos processos não falhem. O detector $\diamond S$ tem as seguintes propriedades:

- *Strong Completeness*: em um tempo finito (*eventually*), todos os processos que falharam serão permanentemente suspeitos por todos os processos corretos.
- *Eventual Weak Accuracy*: há um instante de tempo após o qual algum processo correto não é mais considerado suspeito por nenhum processo correto.

Estas propriedades garantem que, após um determinado intervalo de tempo (intervalo finito, mas desconhecido), todos processos falhos serão considerados suspeitos, mas ao menos um processo correto não será suspeito por nenhum detector. O *serviço de detecção de falha* do *EROPSAR* utiliza essa classe de detector.

4. Desenvolvendo Aplicações Utilizando o *Framework* EROPSAR

O EROPSAR disponibiliza três classes às aplicações clientes e servidores: *RSM* implementa o módulo escalonador de requisição da camada de requisição e resposta, *Request* contém métodos e propriedades úteis para a configuração da requisição e *ClientInterceptador* permite a comunicação transparente entre cliente e servidor replicado através da interface *ClientInterceptador_interface*

Para utilizar o EROPSAR é preciso instanciar essas três classes e implementar as interfaces *PriorityTransform_interface* e *Processor_interface*, descritas anteriormente.

4.1. Configurando uma Aplicação Cliente

Os seguintes passos devem ser seguidos na construção de uma aplicação cliente:

1. Localizar a referência de um dos objetos *ClientInterceptor* disponibilizados remotamente através da interface *ClientInterceptador_interface*.
2. Criar uma instância da classe *Request*
   (a) Request request = new Request();
3. Preencher as propriedades exigidas de uma requisição:

(a) Prioridade da requisição. Ex:
    i. request.priority = ”5”;
(b) nome da interface contendo o método desejado. Ex:
    i. request.objInterfaceName = ”ObjectImpl_interface”;
(c) nome do método remoto desejado.
    i. request.objTask = ”SUM”;
(d) definir se a invocação deverá ser síncrona ou assíncrona. Ex:
    i. request.synchronous =true;
(e) definir os parâmetros de entrada e saída do método invoca. A classe *request* permite a passagem de parâmetros do tipo: *int*, *String* e *Object*. Para definir um parâmetro é preciso especificar o *nome*, o *valor* e a *direção* (parâmetro de entrada ou parâmetro de saída). Ex:
    i. request.addIntParam(”SAIDA_1”,0,request.OUT);
    ii. request.addIntParam(”Parcela1”,5,request.IN);
    iii. request.addIntParam(”Parcela2”,6,request.IN);
    iv. request.addIntParam(”SimulatedDelay”,2000,request.IN);

4. Enviar requisição para o escalonador e receber o resultado do processamento.
   (a) request = clientInterceptor.schedule(request, InvocationTimeout);
   (b) Receber o resultado da requisição.
   Se a invocação for síncrona (*request.synchronous=true*), o método retornará assim que o servidor tiver executado a requisição. O resultado da requisição poderá ser obtido de duas maneiras: *replyRequest.Reply*, para métodos com um único valor de saída e *request.getParam(”Saída_1”)*, para métodos com 1 ou mais valores de saída.
   Se a invocação for assíncrona (*request.synchronous=false*), o método retornará imediatamente após o recebimento da requisição pelas réplicas. O resultado da requisição poderá ser verificado através de *clientInterceptor.requestsPoll* que contém um *Vector* com as últimas requisições processadas e em processamento ou através de *clientInterceptor.getReply(request.ID,pTimeout)*. Quando a requisição não tiver sido ainda processada, *clientInterceptor.getReply(request.ID,pTimeout)*, retorna após *pTimeout* milisegundos, caso contrário, retorna imediatamente com o resultado do processamento.

4.2. Configurando uma Aplicação Servidor

O seguintes passos devem ser seguidos na construção de uma aplicação servidor:

1. Criar uma instância do módulo escalonador de requisição (RSM) enviando como parâmetro as propriedades do serviço (ilustrado na Figura 4 com a descrição de cada parâmetro). O parâmetro *args* contém essas propriedades.
   (a) RSM rsm = new RSM(args);
2. Criar uma instância da implementação das interfaces do *EROPSAR* e também as interfaces dos objetos da aplicação que serão acessadas remotamente.
   (a) ClientInterceptor_Impl clientInterceptor_Impl = new ClientInterceptor_Impl();
   (b) PriorityTransform_Impl priorityTransform_Impl = new PriorityTransform_Impl();
   (c) Processor_Impl processor_Impl = new Processor_Impl(sHostname, sRMIPort, sServerPort);
   (d) Criação dos objetos específicos da aplicação que devem ser acessados remotamente;
3. Registrar a referência do objeto implementado no serviço de nomes

```
EROPSAR.REGISTRY_PORT=10999                # Porta de escuta do serviço de nomes
EROPSAR.DEBUG_LEVEL=6                      # Nível de depuração para escrita no arquivo de log
EROPSAR.ReplicaDelayBeforeProcessing=0     # Gera um atraso de 0ms no processamento de todas as
                                             requisições. Útil para simular uma réplica lenta

# Política de Escalonamento exigida na aplicação, ex: EDF, FIFO, ...
EROPSAR.SchedulePolicy=CLIENT_ID_PRIORITY_MAPPING

EROPSAR.E_GAF_UNIT.NumberOfProcess=2       # Grau de replicação do servidor replicado
EROPSAR.E_GAF_UNIT.Process1=robalo:7000    # Nome e porta de escuta da réplica 1
EROPSAR.E_GAF_UNIT.Process2=traira:7100    # Nome e porta de escuta da réplica 2
EROPSAR.E_GAF_UNIT.PortToListen=7200       # Porta de escuta dessa réplica

EROPSAR.E_GAF_UNIT.FW.NumberOfProcess=2    # Número de processos com serviço de acordo
EROPSAR.E_GAF_UNIT.FW.Process1=robalo:9000 # Nome e porta de escuta do serviço de acordo na réplica 1
EROPSAR.E_GAF_UNIT.FW.Process2=traira:9100 # Nome e porta de escuta do serviço de acordo na réplica 2
EROPSAR.E_GAF_UNIT.FW.PortToListen=9200    # Porta de escuta do serviço de acordo dessa réplica

FD.NumberOfProcess=4                       # Número de detectores de falhas
FD.Process1=robalo:8500                    # Detector de falha 1
FD.Process2=traira:8600                    # Detector de falha 2
FD.PortToListen=8700                       # Porta de escuta do desse DETECTOR
FD.DELTA_DETECTOR_TIMEOUT=9000             # Intervalo inicial em ms p/ que um
                                             processo monitorado seja suspeito
                                             caso não envie "EuEstouOperante"
FD.DELTA_HEARTBEAT=3000                    # Intervalo em ms p/ que HEARTBEAT envie "EuEstouOperante"
```

Figura 4: Propriedades para configuração do serviço - Lado Servidor

(a) Naming.rebind("//"+ sHostname + ":"+ sRMIPort
+ "/ClientInterceptor_interface_"+ sServerPort , clientInterceptor_ Impl);
(b) Naming.rebind("//"+ sHostname + ":"+ sRMIPort
+ "/PriorityTransform_interface_"+ sServerPort , priorityTransform_ Impl);
(c) Naming.rebind("//"+ sHostname + ":"+ sRMIPort
+ "/Processor_interface_"+ sServerPort , processor_ Impl);

5. Avaliação de Desempenho

Todo o projeto foi implementado em JAVA (JDK 1.4.1) e o ambiente utilizado para desenvolvimento foi o JBuilder Personal 7.0. Os testes foram realizados em máquinas PC's 1.2GHz com 256Mb de RAM, com sistemas operacionais Windows XP e Linux (SuSE 8.0) conectados por uma rede Ethernet 10Mbps.

Para avaliar o desempenho do *framework*, armazenamos em um arquivo de log as seguintes marcas de tempo: do envio, da recepção no servidor, do escalonamento, do despacho, do término de processamento e do envio do resultado ao cliente. Essas marcas são utilizadas por dois experimentos. O primeiro experimento, *RoundTrip*, calcula o tempo que o *EROPSAR* leva para enviar uma requisição do cliente a um grupo de réplicas e receber o resultado do processamento de um dos membros do grupo replicado. O segundo experimento calcula o tempo de duração das operações de envio, ordenação total, despacho, processamento e retorno do resultado ao cliente.

O grau de replicação (GR) variou de 3 a 5. Em cada um dos graus de replicação utilizados foram enviadas cem requisições. Esse experimento teve como objetivo detectar a sobrecarga causada a medida que o sistema incrementa seu grau de replicação.

O gráfico da Figura 5 mostra a média obtida na medição do *RoundTrip* para cada grau de replicação utilizado. Observamos com as medições que à medida que o grau de replicação aumenta, o tempo de *RoundTrip* também aumenta.

O gráfico da Figura 6 mostra a média obtida na medição das operações de transição de estado da requisição. Observamos com as medições que as operações de envio, despacho e processamento se mantiveram constantes, independente do grau de replicação. Nas operações de ordenação, à

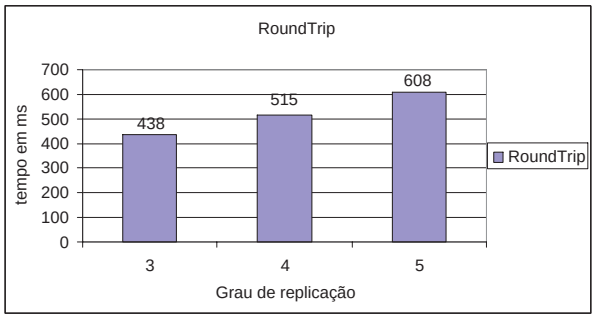

Figura 5: *RoundTrip* no *EROPSAR*

medida que o grau de replicação aumenta, mais tempo é exigido para que a operação seja concluída. Isso ocorre porque essa operação exige a realização de um protocolo de acordo que tende a ser mais lento à medida que mais processos estão envolvidos em um acordo.

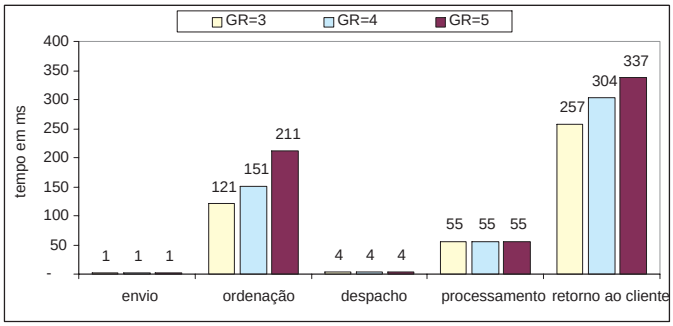

Figura 6: Duração das operações de transição de estado

A operação de retorno também possui uma tendência crescente à medida que o grau de replicação aumenta. Um outro fator que determina essa tendência crescente está relacionado às requisições completadas não entregues por ainda não serem estáveis. Esse fator depende da velocidade de execução das réplicas, portanto do valor de GEP.

6. Conclusões

Nesse trabalho projetamos e implementamos um *framework*, denominado *EROPSAR*, que viabiliza a construção de aplicações com alta disponibilidade de serviço. A alta disponibilidade é alcançada através da técnica de replicação ativa [13] associada a uma política de alocação de recursos baseada em prioridade [9, 10]. Nesse contexto, o EROPSAR contém protocolos que possibilitam: o *compartilhamento de recursos com tratamento de inversão de prioridade*, a *redundância de componentes* e a *comunicação cliente-servidor de modo síncrono e assíncrono*.

Os benefícios oferecidos no EROPSAR para prover alta disponibilidade possuem um custo: quanto mais réplicas forem utilizadas para tolerar falhas mais lento será o escalonamento de requisições. Esse atraso é causado pela necessidade das réplicas realizarem os acordos para definir uma ordem comum de processamento. Um outro custo está relacionado ao uso de prioridades no processamento ativamente replicado, pois como casos de inversão de prioridade em grupo precisam ser tratados, eventuais retrocessos podem ser necessários para garantir a consistência entre as réplicas.

Agradecimentos

Este trabalho foi parcialmente financiado pelo CNPq (processo 300646/1996-8).

Referências

[1] ALMEIDA, J., DABU, M., MANIKUTTY, A., AND CAO, P. Providing differentiated levels of service in web content hosting. In *First Workshop on Internet Server Performance* (Junho 1998), ACM.

[2] BUDHIRAJA, N., MARZULLO, K., SCHNEIDER, F., AND TOUEG, S. *Distributed Systems*, s. mullender ed. Addison Wesley, 1993, ch. 8: The Primary-Backup Approach, pp. 199–216.

[3] CARR, D. F. Don't get spiked. *Internet World 5*, 34 (Janeiro 1999), 59. Disponível por www em http://www.acm.org/technews/articles/1999-1/1208w.html (5 de novembro de 2003).

[4] CHANDRA, T., HADZILACOS, V., AND TOUEG, S. The weakest failure detector for solving consensus. *Journal of the ACM 43*, 4 (Julho 1996), 685–722.

[5] CHANDRA, T., AND TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Communications of the ACM 43*, 2 (1996), 225–267.

[6] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (1985), 374–382.

[7] GRIBBLE, S. Robustness in complex systems. In *Proceedings of the Eighth International Symposium on Hot Topics in Operating Systems (HotOS-VIII)* (2001).

[8] JALOTE, P. *Fault Tolerance in Distributed Systems*. Prentice Hall, NJ, 1994.

[9] MENASCÉ, D., ALMEIDA, V., FONSECA, R., AND MENDES, M. A. A methodology for workload characterization of e-commerce sites. In *Proceedings of the 1st ACM conference on Electronic commerce* (1999), ACM Press, pp. 119–128.

[10] MENASCÉ, D., ALMEIDA, V., FONSECA, R., AND MENDES, M. A. Resource management policies for e-commerce servers. In *Proceedings of the Second Workshop on Internet Server Performance* (Maio 1999).

[11] OMG. *Real-Time CORBA - Join Revised Submission*, document orbos/98-12-05 ed. Object Management Group, Dezembro 1998. URL: http://www.omg.org.

[12] SCHMIDT, D., STAL, M., ROHNERT, H., AND BUSCHMANN, F. *Pattern-Oriented Software Architecture*, vol. Vol. 2: Patterns for Concurrent and Networked Objects. John-Wiley & Sons, 2000.

[13] SCHNEIDER, F. *Distributed Systems*, s. mullender ed. Addison Wesley, 1993, ch. 7: Replication Management using the State-Machine Approach, pp. 169–197.

[14] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. Priority inheritance protocols: an approach to real-time synchronisation. *IEEE Transaction on Computers 39*, 9 (1990), 1175–1185.

[15] WANG, Y., ANCEAUME, E., BRASILEIRO, F., GREVE, F., AND HURFIN, M. Solving the group priority inversion problem in a timed asynchronous system. *IEEE Transactions on Computers 51*, 8 (2002), 900–915.

Validando Sistemas Distribuídos Desenvolvidos em Java Utilizando Injeção de Falhas de Comunicação por Software*

Gabriela Jacques-Silva[1†], **Regina Lúcia de O. Moraes**[2] ,
Taisy Silva Weber[1] , **Eliane Martins**[3]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul
Caixa Postal 15064 – 90501-970 Porto Alegre, RS

[2]CESET – Universidade Estadual de Campinas
Caixa Postal 456 – 13484-370 Limeira, SP

[3]Instituto de Computação - Universidade Estadual de Campinas
Caixa Postal 6176 – 13083-870 Campinas, SP

gjsilva@inf.ufrgs.br, regina@ceset.unicamp.br,
taisy@inf.ufrgs.br, eliane@ic.unicamp.br

Abstract. *Java distributed applications used in high-available systems require validated fault tolerance mechanisms, to avoid unexpected behavior during execution. Jaca is a fault injection tool to experimentally validate Java applications. In its first version, Jaca had three fault classes, based on interface faults. This work presents an expansion of Jaca's fault model to communication faults. This extension allows the use of Jaca to conduct validation experiments in distributed systems.*

Resumo. *Aplicações distribuídas em Java usadas em sistemas de alta disponibilidade exigem que mecanismos de tolerância a falhas sejam validados, para não apresentarem um comportamento inesperado no momento em que são requisitados no ambiente operacional. Jaca é uma ferramenta de injeção de falhas para validação experimental de aplicações Java. Em sua primeira versão, Jaca apresentava três classes de falhas, baseadas em falhas de interface. Este trabalho apresenta a expansão do modelo de falhas de Jaca para falhas de comunicação. Essa extensão permite o uso dessa ferramenta na condução de experimentos de validação em sistemas distribuídos.*

1. Introdução

Permitir que sistemas computacionais sejam incorporados ao cotidiano para atender serviços de missão crítica exige que os mesmos apresentem características de tolerância a falhas para corresponder a confiança depositada no comportamento correto desses serviços. Uma etapa fundamental no desenvolvimento de sistemas tolerantes a falhas é a fase de validação. Delegar a verificação do funcionamento do mecanismo de tolerância a falhas para uma situação de uso efetivo do *software* (uma falha real) pode gerar conseqüências desastrosas.

A validação pode ser tanto analítica quanto experimental, sendo estas duas formas complementares. Uma das técnicas usadas para validar experimentalmente um sistema é injeção de

*Parcialmente financiado pelos projetos ACERTE/CNPq (472084/2003-8) e DepGriFE/HP Brasil P&D
†Bolsista do Conselho Nacional de Desenvolvimento Científico e Tecnológico

falhas. Através desta técnica, são introduzidas falhas no sistema de maneira controlada e é monitorada a resposta do sistema nessas condições. Seu objetivo é testar a eficiência dos mecanismos de tolerância a falhas e avaliar a segurança de funcionamento dos sistemas, provendo uma realimentação no processo de desenvolvimento [Iyer 1995].

Sistemas distribuídos são comumente usados para oferecer alta disponibilidade para serviços de missão crítica, com isso faz-se indispensável a validação da sua dependabilidade. Uma maneira é injetar falhas no sistema de troca de mensagens, para forçar a ativação dos mecanismos de detecção de erro e recuperação do estado distribuído. Se forem usados outros modelos de falhas, como falhas de memória ou da unidade de processamento, a latência de manifestação da falha será muito grande. Para acelerar essas manifestações, as falhas são injetadas diretamente no sistema de troca de mensagens.

Este trabalho apresenta um injetor de falhas de comunicação escrito na linguagem de programação Java. Através desta ferramenta pode ser testada mais facilmente a dependabilidade em sistemas distribuídos implementados em Java, que é largamente usada no desenvolvimento de aplicações para este tipo de ambiente. A maior vantagem de injetar falhas em um nível acima da pilha de protocolos de comunicação do sistema operacional é a portabilidade, pois só dependerá de Java, portável para várias arquiteturas e sistemas operacionais.

Para desenvolver esta ferramenta estendeu-se o injetor de falhas Jaca [Martins *et al.* 2002]. Jaca é baseada no sistema de padrões para injeção de falhas [Leme *et al.* 2001] e sua implementação em reflexão computacional [Maes 1987]. Uma de suas principais características é a possibilidade de ser estendida para novos modelos de injeção de falhas. Neste trabalho, esta característica é explorada para injeção de falhas de comunicação no protocolo UDP. A próxima seção apresenta alguns mecanismos de injeção de falhas por *software*, mostrando algumas ferramentas de injeção de falhas e também o sistema de padrões existente para este tipo de ferramenta. A seção 3 apresenta Jaca e também como reflexão computacional foi usada em sua implementação. A seção 4 descreve a extensão da ferramenta, mostrando o modelo de falhas adicionado e como este foi implementado. A última seção apresenta algumas conclusões e trabalhos futuros.

2. Injeção de Falhas

Injeção de falhas é uma técnica de teste em que se procura produzir ou simular falhas e observar o sistema sob teste para verificar sua resposta nessas condições [Hsueh *et al.* 1997]. Conhecendo o modelo de falhas do sistema sob teste, podem ser criadas falhas que são injetadas no sistema de acordo com o modelo suportado. Durante e após a injeção das falhas, o sistema é monitorado para observar o efeito causado. Através deste tipo de experimento podem ser determinadas medidas de dependabilidade, como a cobertura da detecção de erros, a eficiência e o impacto no desempenho dos mecanismos de tolerância a falhas. Experimentos de injeção de falhas também podem revelar problemas de *software*, que não são encontrados com técnicas tradicionais de teste e métodos formais [Voas 1997].

A injeção de falhas pode ser aplicada a diferentes fases do ciclo de vida de um sistema, usando diferentes formas de aplicação, sendo as mais comuns injeção de falhas por simulação, por *hardware* e por *software*. A ferramenta Jaca e sua extensão se enquadram em injeção de falhas por *software*.

2.1. Injeção de Falhas por *Software*

Uma ferramenta de injeção de falhas por *software* geralmente é um trecho de código que usa todos os ganchos (*hooks*) possíveis do processador e do sistema para criar um comporta-

mento incorreto de maneira controlada [Carreira e Silva 1998]. Esta técnica pode simular falhas de *hardware*, de *software* e de interface. Interessam a esse trabalho falhas de *hardware* emuladas e injetadas por *software*. Nesse caso, são as manifestações de falhas de *hardware* (erros) que são injetadas e não as falhas propriamente ditas.

Exemplos de falhas de *hardware* são: falhas de memória, de processador, de barramento e, no caso de sistemas distribuídos, falhas de comunicação. Estas afetam mensagens que são transmitidas através de um canal de comunicação, que podem ser omitidas, alteradas, duplicadas ou entregues com atraso.

Os mecanismos que podem ser usados para injetar falhas por *software* são classificados em três categorias [Rosenberg e Shin 1993]: (i) a injeção ativa é realizada por um processo especial executado concorrentemente com os processos que fazem parte do sistema sob teste, podendo injetar falhas em ambientes onde não haja proteção contra acesso externo; (ii) a injeção por alteração de fluxo de controle é usada para alterar o comportamento funcional do sistema sob teste e, quando ativada, executa uma seqüência alternativa de instruções, de modo que a função corrente seja executada incorretamente; (iii) a injeção por alteração de código é usada para injetar falhas em áreas do programa nas quais um processo externo não pode ter acesso. A execução da aplicação sob teste é interrompida e em seu lugar é executado um código que injeta falhas no recurso do sistema especificado.

Essa alteração pode ser estática, quando feita em tempo de compilação, ou dinâmica, quando feita em tempo de execução. Na alteração estática, as falhas são introduzidas no código do programa alvo, alterando instruções do programa. A vantagem desta abordagem é que não é necessário nenhum *software* extra durante a execução para injetar falhas, porém, o modelo de falhas que pode ser injetado é limitado a falhas de *software* e conseqüências de falhas permanentes de *hardware*. Na alteração dinâmica, código extra é necessário para injetar falhas e monitorar seus efeitos e, também, é requerido um mecanismo para disparar a injeção de falhas.

Para a alteração dinâmica pode-se usar os seguintes métodos [Hsueh *et al.* 1997]: modo depuração, baseado em *time out*, exceções/interrupções e inserção de código. Jaca é baseado em exceções/interrupções, onde as falhas são disparadas por ocasião da ocorrência de alguns eventos, tais como a execução de uma determinada instrução, chamadas a rotinas de sistema ou o acesso a uma determinada posição de memória.

Um problema a ser considerado quando se usa mecanismos dinâmicos de injeção, é o fato de eles serem intrusivos, uma vez que sua execução não é transparente para o sistema alvo, alterando, muitas vezes de maneira bastante significativa, seu desempenho. Dessa forma os resultados obtidos são afetados devido à sobrecarga causada pela inserção do código do injetor de falhas. As ferramentas de injeção, além de interferirem no sistema para introduzir os erros desejados, precisam monitorar o sistema e determinar se a falha e os mecanismos de tolerância à falhas foram ativados, coletando dados que auxiliem no diagnóstico dos erros apresentados. Uma maneira de se reduzir essa interferência é usar técnicas híbridas, empregando ferramentas de *software* e de *hardware*.

Para se aplicar injeção de falhas por *software* não é preciso de *hardware* especial, não há risco de dano aos componentes e os testes podem ser mais facilmente controlados e observados. Devido a essas vantagens, injeção de falhas por *software* tem se tornado mais popular entre os desenvolvedores de sistemas tolerantes a falhas.

2.2. Ferramentas de Injeção de Falhas

Para que se possa injetar falhas é necessário dispor de ferramentas apropriadas. Essas ferramentas executam o sistema sob teste e produzem ou simulam a presença de falhas, monitorando

o sistema para verificar qual é seu comportamento. Para que se tenha sucesso nos testes, a ferramenta deve ser eficiente tanto no momento de injetar as falhas quanto ao reportar os resultados dos testes e o comportamento do sistema ao tratar as falhas injetadas.

Alguns exemplos de ferramentas de injeção de falhas de comunicação são CSFI, ORCHESTRA e ComFIRM. CSFI (*Communication Software Fault Injection*) [Carreira *et al.* 1995b] foi uma das primeiras ferramentas desenvolvidas unicamente para injeção de falhas de comunicação e seu objetivo principal era avaliar o impacto de falhas em sistemas paralelos. A versão existente de CSFI foi implementada para um sistema *transputer* T805. Outro ambiente bastante conhecido é ORCHESTRA [Dawson *et al.* 1996], desenvolvido especificamente para teste de dependabilidade de protocolos distribuídos. O mecanismo de injeção de falhas é através da inserção de uma camada na pilha de protocolos, chamada de PFI (*Protocol Fault Injection*). A ferramenta ComFIRM (*Communication Fault Injection through OS Resources Modification*) [Barcelos *et al.* 2000] propõem-se a injetar falhas apenas de comunicação. ComFIRM se situa no núcleo do sistema operacional Linux, no nível mais baixo do tratamento de mensagens pelo subsistema de rede. O código da ferramenta é inserido diretamente código do núcleo do sistema operacional, o que diminui consideravelmente a intrusão da ferramenta no sistema sob teste.

Outra classe de ferramentas é a das que permitem a extensão de seu modelo de falhas, tais como GOOFI, NFTAPE e FIDe. GOOFI (*Generic Object-Oriented Fault Injection*) [Aidemark *et al.* 2001] é uma ferramenta desenvolvida recentemente e é considerada genérica por não ser presa a nenhuma técnica específica de injeção de falhas. Desta maneira é construído um ambiente de fácil adaptação para injeção de falhas necessárias para um determinado sistema alvo. Esta ferramenta é altamente portável por ser desenvolvida em Java e usar um banco de dados compatível com a linguagem SQL. Sua principal diferença em relação a Jaca é que o seu desenvolvimento não foi baseado em um sistema de padrões de *software*. A ferramenta NFTAPE [Stott *et al.* 2000] destaca-se por ser uma ferramenta com múltiplos modelos de falhas, diversos modos de disparo para injeção de falhas e também diversos sistemas alvo. Para permitir essa multiplicidade, NFTAPE inova criando o conceito de Injetor "Leve" de Falhas (*LightWeight Fault Injector* – LWFI). Para desenvolver um novo injetor de falhas, apenas um novo LWFI precisa ser implementado, o que facilita sua expansibilidade para novos modelos de falhas. FIDe (*Fault Injection via Debugging*) [Gonçalves *et al.* 2001] é um injetor de falhas baseado nos recursos de depuração do sistema operacional Linux. Para injetar falhas, FIDe usa a chamada de sistema `ptrace()`, que intercepta a aplicação toda vez que uma chamada de sistema ocorre. A extensibilidade de FIDe se diferencia de outras ferramentas, como GOOFI e NFTAPE, pois sua capacidade de expansão está diretamente ligada a flexibilidade da chamada `ptrace()`. O modelo de falhas desta ferramenta é tão amplo quanto as chamadas de sistemas que são interceptadas por `ptrace()`.

Jaca é uma ferramenta de injeção de falhas extensível destinada à validação de aplicações orientadas a objeto desenvolvidas em Java. Sua implementação é baseada em reflexão computacional e sua arquitetura baseada em um sistema de padrões de *software*. Ao contrário de Jaca, a maioria das ferramentas descritas apresentam graves problemas de portabilidade, que vão desde o desenvolvimento para uma arquitetura específica (CSFI, para *transputers*) até o desenvolvimento para uma versão específica de núcleo de sistema operacional (ComFIRM, existente apenas para uma versão antiga do *kernel* do Linux). Além disso, por estas ferramentas terem sido criadas dentro de universidades, a maioria delas já tiveram seu desenvolvimento descontinuado, o que dificulta sua utilização.

2.3. Sistema de Padrões para Injeção de Falhas

No desenvolvimento de *software* é comum deparar-se com problemas que são recorrentes dentro de um determinado domínio de aplicações e não raro, resolvem-se esses problemas de forma similar a outras soluções já utilizadas por outros desenvolvedores. Essas soluções podem ser documentadas em forma de padrões que são independentes de linguagens de programação. Esses padrões auxiliam o desenvolvimento mais eficiente, robusto, portável e reutilizável [Schmidt 1995], facilitando o entendimento e instanciando soluções que têm sido consideradas eficientes para solucionar os problemas a que se propõem.

Baseado nessa constatação e na arquitetura proposta por Hsueh [Hsueh *et al.* 1997], foi desenvolvido um padrão para o desenvolvimento de novas ferramentas de injeção de falhas [Leme *et al.* 2001] que contempla: (i) ativação e monitoramento do sistema sob teste, (ii) coordenação dos experimentos e (iii) apresentação e armazenamento dos dados coletados.

Uma ferramenta de injeção de falhas pode ser decomposta em cinco subsistemas: (i) o *Controlador* que coordena os demais subsistemas, (ii) o *Injetor* que injeta as falhas no sistema sob teste, (iii) o *Monitor* que coleta as ocorrências relacionadas ao comportamento do sistema em presença das falhas injetadas, (iv) o *Ativador* que ativa o sistema e (v) a *Interface de Usuário* que permite que os usuários especifiquem os experimentos, acompanhem o progresso desses experimentos e obtenham os resultados.

O *Injetor*, o *Monitor* e o *Ativador* são os únicos subsistemas que podem interagir diretamente com a aplicação. Diversas instâncias do *Injetor* e do *Monitor* podem ser criadas para permitir a injeção e o monitoramento de diferentes partes do sistema sob teste durante um experimento. Além desses componentes, dois repositórios de dados são utilizados, um que armazena as falhas a serem injetadas (*Gerenciador de Falhas*) e outro que armazena os aspectos do sistema a serem monitorados (*Gerenciador de Monitoração*).

O *Injetor* e o *Monitor* por sua vez, são subsistemas que podem usar um padrão próprio, definindo um padrão de mais baixo nível. O *injetor* é composto de três componentes: o *gerenciador de injeção*, o *injetor lógico* e o *injetor físico*. O *gerenciador de injeção* controla o processo de injeção através da criação dos *injetores lógicos* de acordo com o tipo de falha especificada no *Gerenciador de Falhas* e ativa o *injetor lógico* apropriado quando chega o momento de injeção. Um *injetor lógico* é específico para uma determinada falha, mas não interage diretamente com o sistema sob teste. Isto será feito através do *injetor físico*, que irá efetivamente injetar as falhas especificadas através de sua interface, que contém operações que permitem essa interação. No caso da aplicação ou seu ambiente serem trocados, apenas esse componente deverá ser reescrito.

O padrão do Monitor é semelhante ao padrão acima descrito e por esse motivo não será aqui reapresentado.

3. Jaca – Ferramenta de Injeção de Falhas em Java

Como já citado na subseção 2.2, a principal motivação da ferramenta Jaca é auxiliar na validação de aplicações implementadas em Java. Jaca é uma evolução da ferramenta FIRE [Martins e Rosa 2000], que também usa de reflexão computacional para introduzir falhas no sistema alvo. A diferença entre as duas ferramentas é que, além de Jaca basear-se nos padrões de *software* descritos na seção 2.3, FIRE visa validar aplicações implementadas em C++. Jaca oferece mecanismos para injetar falhas de alto nível em sistemas orientados a objetos. Ao invés de afetar *bytes* de memória ou o conteúdo de registradores, Jaca afeta a interface pública de objetos, ou seja, atributos públicos, bem como parâmetros e valores de retorno de métodos públicos.

Para injetar falhas de alto nível, Jaca usa reflexão computacional, permitindo uma fácil adaptação, atendendo a especificação de qualquer sistema Java com um mínimo de reescrita, e não necessitando do código fonte do sistema sob teste. Isso acontece devido ao uso de uma plataforma de apoio a reflexão chamada Javassist [Chiba 2000], que permite a transformação de *bytecodes* em tempo de carga. Com isso, a instrumentação necessária para a injeção é introduzida no *bytecode*, podendo então ser aplicada mesmo que o código fonte não esteja disponível. Essa característica proporciona independência e portabilidade da ferramenta que pode rodar em qualquer plataforma que possa executar a máquina virtual padrão do Java. Jaca também monitora os sistemas para verificar se as respostas por ele produzidas estão dentro dos padrões esperados, mesmo em presença de falhas.

A figura 1 mostra a arquitetura da ferramenta Jaca. Como pode ser visto, sua estrutura de pacotes é semelhante à estrutura dos subsistemas do padrão para ferramentas de injeção de falhas. A funcionalidade de cada pacote segue a mesma funcionalidade descrita para cada um dos subsistemas equivalentes do padrão.

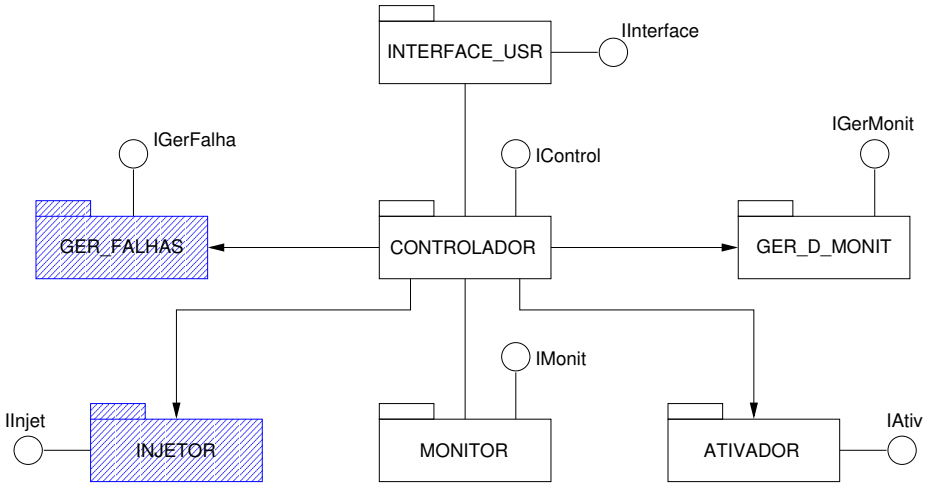

Figura 1: Arquitetura da ferramenta Jaca

3.1. Reflexão Computacional em Jaca

De acordo com Maes [Maes 1987], computação reflexiva é a atividade desempenhada por um sistema computacional quando a computação realizada é sobre a sua própria computação. A base para a construção de sistemas reflexivos é a sua divisão em dois níveis: o *nível base* e o *meta-nível*. O *nível base* é onde se encontra a implementação da aplicação. O *meta-nível* é onde as estruturas do *nível base* podem ser observadas ou ter seu comportamento modificado. Uma possível implementação destes conceitos a linguagens orientadas a objeto é o uso de *meta-objetos*.

Quando Jaca foi implementada, Javassist [Chiba 2000] foi escolhido como ferramenta de reflexão devido a sua conformidade com os requisitos da Jaca. Requisitos como portabilidade, introspecção nas classes, independência de código fonte e facilidade de programação foram possíveis com o uso de Javassist. A arquitetura de Jaca não é presa a nenhum protocolo de meta-objetos especificamente. Porém, quando um é escolhido para implementação, este será um fator limitante, já que Jaca vai estar limitada pelo poder de atuação do próprio protocolo.

O uso das classes do *toolkit* de reflexão computacional em Jaca se dá basicamente em três pacotes: `INJETOR`, `MONITOR` e `ATIVADOR`. O pacote `INJETOR` é o que faz a injeção de falhas em si, através da classe `InjetorFisico`. Para injetar de falhas, o `InjetorFisico` é implementado como um meta-objeto, sendo uma subclasse da classe

`javassist.reflect.Metaobject` de Javassist. Este `InjetorFisico` é associado a cada objeto da lista de classes, fornecida por um arquivo de configuração usado para identificar quais classes devem ser monitoradas. Desta forma os objetos especificados são interceptados sempre que há a leitura ou escrita de um atributo de classe ou então quando uma chamada de método é executada. É durante esta interceptação que o processo de injeção de falhas ocorre. O tipo de falha é buscada em um arquivo, que contém a lista de falhas que serão injetadas durante o experimento. Este arquivo é interpretado pelo gerenciador de falhas, que passa ao gerenciador de injeção, através do controlador, as informações necessárias para instanciar os injetores lógicos.

Quanto ao pacote `MONITOR`, o uso do *toolkit* Javassist é similar. O objeto associado aos objetos do nível base é o meta-objeto `SensorFisico`. Este meta-objeto faz a interceptação com o objetivo único de executar o monitoramento. Os dados obtidos são repassados ao controlador, que os registra em um arquivo de *log*. O pacote `ATIVADOR` usa das classes de Javassist para tornar reflexivas as classes que foram especificadas na configuração e também para ativar o sistema alvo. O diagrama de funcionamento de Jaca pode ser vista na figura 2.

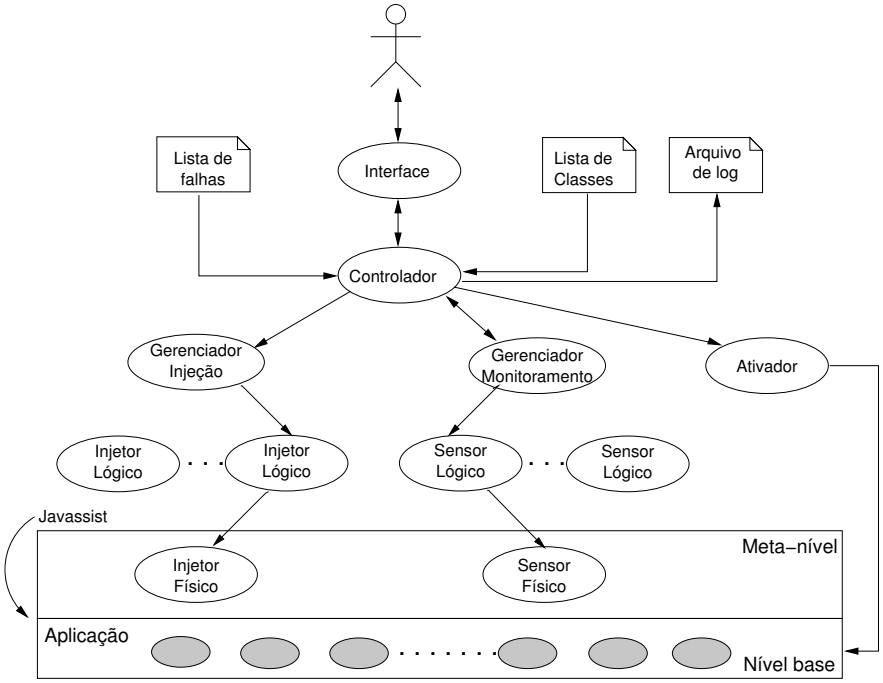

Figura 2: Diagrama de funcionamento da ferramenta Jaca

4. Extensão de Jaca para Falhas de Comunicação

A ferramenta Jaca, em sua primeira versão, injeta três tipos de falhas: falhas de atributos, falhas de parâmetro e falhas de retorno de métodos. Considerando as três classes disponíveis, a dificuldade para testar sistemas distribuídos é consideravelmente grande, uma vez que a latência para uma destas falhas se manifestar como um erro no sistema de troca de mensagens é muito alta. Para facilitar a execução de testes neste tipo de aplicações, Jaca foi estendida para modelos de falhas relativos a comunicação.

Os modelos escolhidos para a extensão são modelos para falhas que podem ocorrer no protocolo UDP (*User Datagram Protocol*) e no protocolo TCP (*Transmission Control Protocol*). Ambos os protocolos estão disponibilizados através da interface de programação de Java. A principal vantagem de oferecer um injetor de falhas de comunicação onde as falhas são injetadas em um

nível de abstração mais alto é a portabilidade. A maioria dos injetores de falhas de comunicação existentes esbarram no problema da portabilidade, pois geralmente são ligados diretamente ao sistema operacional.

A extensão do modelo de falhas de Jaca foi bastante facilitada pelo fato desta ferramenta ser baseada no sistema de padrões descrito na seção 2.3. O padrão de projeto *Injetor* já prevê a expansão do modelo. Com isso, a extensão da ferramenta passou por três fases: o estabelecimento dos modelos de falhas, a decisão do momento em que a falha deve ser injetada e a implementação de cada uma das falhas do modelo. As seções a seguir descrevem cada uma destas fases.

4.1. Modelos de Falhas

Falhas são fenômenos aleatórios, imprevisíveis e que podem levar um sistema a um estado errôneo. Se os erros não são tratados, o sistema pode apresentar defeitos. Um defeito ocorre sempre que um serviço não é prestado de acordo com a sua especificação. Dependendo do tipo de falha, ela recebe uma classificação. Em um contexto distribuído, as falhas que são considerados são baseados no modelo de falhas para sistemas distribuídos definido por Cristian [Cristian 1991].

Este modelo descreve falhas de omissão, temporização, resposta e colapso. Uma falha de omissão ocorre quando um servidor não responde a uma requisição. Uma falha é considerada de temporização quando uma resposta do servidor ocorre fora do intervalo de tempo especificado, podendo ser tanto uma resposta cedo demais quanto tardia demais. A última geralmente é associada a falhas de desempenho. A falha de resposta ocorre quando o servidor responde incorrentamente, tanto podendo o valor de retorno de uma requisição ser incorreto quanto uma transição de estado ocorrer incorretamente. Uma falha por colapso ocorre quando o servidor pára totalmente de responder. Este tipo de falha ainda pode ser classificada em subtipos, levando em consideração o estado do servidor após a sua recuperação. Observa-se que a causa específica de cada falha não interessa ao modelo, pois uma aplicação distribuída reage a falhas que afetam a troca de mensagens ou a queda de servidores para manter o sistema distribuído livre de defeitos.

O modelo de falhas para UDP é o mais detalhado. Os tipos de falhas consideradas são: omissão, temporização e colapso. São incluídas também falhas de ordenamento, que é um subtipo de falhas de temporização, e inserção de mensagens espúrias e duplicação de mensagens, ambas subtipos de falhas de resposta. A escolha por este modelo vem do fato de estes tipos de falhas serem bastante comuns em ambientes distribuídos. Além disto, nenhuma destas falhas é tratada pelo próprio protocolo UDP. Com isso, todas essas falhas podem ser diretamente percebidas pela aplicação, que deve estar preparada para sua detecção e correção.

O modelo TCP inclui apenas falhas de colapso, pois assume-se que é a única falha que pode chegar até a aplicação. Falhas de omissão, ordenamento, inserção de mensagens, duplicação e temporização são todas já mascaradas pelo protocolo. Como o objetivo desse injetor não é o teste dos mecanismos de tolerância a falhas do próprio protocolo, mas sim da aplicação que usa o protocolo, estas falhas não são consideradas.

4.2. Ativação da Falha

Uma abordagem comumente usada para ativar falhas de comunicação é disparar a falha no envio de uma mensagem ou na solicitação de uma requisição remota, no caso de uma chamada de método remoto. Considerando que deseja-se injetar falhas de comunicação na linguagem Java, a interceptação para realizar a injeção deve ser na interface de programação dos recursos de redes, como nos *sockets* TCP e UDP e no procurador do método remoto (*stub*).

No caso de *sockets* UDP, a classe que permite o envio de mensagens é `java.net.DatagramSocket`, através do método `send`. Portanto, é na execução deste

método que a interceptação deve ocorrer. Para a execução de um método ser interceptada, a classe deve ser reflexiva, ou seja, todos os objetos desta classe devem ter um meta-objeto associado. Com isso, toda execução de um método deste objeto é interceptada e assim a falha pode ser injetada ou não, baseada no arquivo de configuração de falhas. Entretanto, o carregador de classes de Javassist que torna as classes reflexivas não permite a reflexão de classes de sistema.

Para contornar este problema foi criada uma classe *wrapper* para a classe `java.net.DatagramSocket`. No caso, esta classe *wrapper* é uma subclasse da classe de sistema, que reimplementa o método `send` e pode ser reflexionada pelo carregador de classes de Javassist. A classe `java.net.DatagramSocket` pode ter subclasse pois não é declarada com a palavra chave `final`. Um exemplo simplificado de classe *wrapper* pode ser visto na figura 3. O atributo boleano é usado para decidir se a mensagem deve ser enviada ou não. Este atributo é necessário pois, no momento que um determinado método é interceptado, este deve ser executado até o seu final. Se este não for executado, a pilha de execução não é corretamente desfeita e o programa não consegue prosseguir de forma correta.

```
public class DatagramSocket_ extends java.net.DatagramSocket
{
   boolean envia;

   // Métodos para definição do atributo 'envia'

   public void send(java.net.DatagramPacket packet)
                  throws java.io.IOException
   {
      if ( envia == true )
         super.send(packet);
      else
         return;
   }
}
```

Figura 3: Classe *wrapper* para a classe de sistema

O emprego de classes *wrapper* já insere uma nova condição na ferramenta Jaca, que antes não requisitava o código fonte das aplicações para executar os experimentos de injeção de falhas. Neste caso, o usuário da ferramenta poderia usar de *scripts* para fazer a alteração do programa, redefinindo o nome das classes de `java.net.DatagramSocket` para `DatagramSocket_`. Para casos onde o código fonte não está disponível ou o usuário não deseja alterar código, pode-se alterar diretamente o *bytecode* das classes. A tabela de símbolos deve ser mudada, trocando a referência da classe de sistema para a referência da classe *wrapper*. Para isso, será desenvolvido um programa que use um *toolkit* que permita fazer alteração de *bytecodes*.

O segundo problema é relativo à ativação das falhas em programas que usam *sockets* TCP. Quando se usa *sockets* UDP, a única forma de enviar mensagens é pelo método `send` da classe `java.net.DatagramSocket`. Porém, quando se trata de *sockets* TCP, há uma vasta gama de opções para se escrever dados em um *socket*, inclusive classes que fazem operações de entrada e saída. Com isso surgem duas dificuldades: (i) a implementação de uma classe *wrapper* para cada uma das possíveis classes, e (ii) a diferenciação entre escritas em um arquivo local e em um *socket*. Tais problemas ainda estão sendo investigados, e por estas razões tal modelo ainda não foi implementado.

Está prevista também a extensão de Jaca para a injeção de falhas no protocolo RMI (*Remote Method Invocation*). Para isso, as chamadas remotas devem ser interceptadas, o que não deve apresentar dificuldades adicionais, visto que estas chamadas podem ser interceptadas durante a execução de métodos da classe *stub*. As classes *stub* são geradas pelo compilador RMI (`rmic`)

e não são consideradas classes de sistema. Portanto, podem ser reflexionadas pelo carregador de classes do Javassist.

4.3. Implementação do Modelo UDP

Apesar do protocolo UDP não ser confiável, este é comumente usado como base para a construção de mecanismos de tolerância a falhas, onde a confiabilidade requisitada é implementada nas camadas superiores. Um exemplo é o sistema de comunicação de grupo JGroups [Ban 1998], usado como bloco básico para construção de aplicações de alta disponibilidade. JGroups usa UDP como padrão na base de sua pilha de protocolos, sendo as camadas superiores da pilha que implementam a segurança de funcionamento da troca de mensagens. Neste caso, um injetor de falhas de comunicação para o protocolo UDP é fundamental da validação do funcionamento correto destas camadas.

Para adicionar um novo tipo de falha em Jaca precisam ser alterados dois pacotes: `GER_FALHAS` e `INJETOR`, destacados na figura 1. No pacote `GER_FALHAS` tem-se que adaptar o gerenciador para interpretar no arquivo de lista de falhas este novo tipo de falha adicionado. No pacote `INJETOR` inicia-se o processo de extensão pela adaptação de `GerenciadorDeInjecao`. Este é responsável pela instanciação de cada um dos injetores lógicos requisitados durante o experimento. A classe `InjetorFisico` também deve ser alterada, para estar preparada para injetar a nova falha. As duas classes que devem ser adicionadas são o injetor especializado para a nova falha e também a classe que descreve e guarda os dados da falha. Alterando e adicionando estas classes, Jaca está pronta para injetar o novo tipo de falha especificada. Até o momento foram adicionadas falhas de omissão, colapso e duplicação de mensagens.

No caso de falhas de omissão para o protocolo UDP foram criadas duas classes: `FalhaUdpOmissao` e `InjetorFalhaUdpOmissao`. A primeira é a classe que armazena os dados da falha e a segunda é o injetor lógico. Os atributos armazenados na classe `FalhaUdpOmissao` incluem o endereço IP de origem e de destino, a porta da conexão na origem e no destino, e também a taxa em que ocorrerá uma falha, caso ela seja configurada como intermitente. Se o usuário desejar configurar uma falha deste tipo é necessário adicionar no arquivo de lista de falhas a seguinte linha:

```
FalhaUdpOmissao:<padrao_de_repeticao>:<início>:<taxa_de_falha>:
<host_origem>:<porta_origem>:<host_destino>:<porta_destino>:
```

O padrão de repetição determina se a falha é transiente (`t`), permanente (`p`) ou intermitente (`i`). O início do processo de injeção também é configurável e é determinado a partir de qual mensagem (n-ésima mensagem) as falhas começarão a ser injetadas. O *socket* no qual a falha será injetada é identificado pelo par nome do *host* e porta na origem com nome do *host* e porta no destino. A configuração da origem é opcional e é interessante ser usada quando um *host* possui mais de uma interface de rede. Como um *socket* UDP não tem conexão, os endereços de destino são verificados diretamente no datagrama que está sendo enviado.

A implementação do novo modelo de falhas mostrou que o uso de uma ferramenta de arquitetura extensível é viável. A extensão de uma ferramenta pré-existente tem um custo de desenvolvimento bem menor que o projeto e a criação de uma nova ferramenta. No caso de Jaca, a extensão também foi facilitada por ser uma ferramenta baseada em um padrão de *software* bem documentado que já previa a expansão do modelo de falhas. Em Jaca, é necessário, pelo menos, a criação de duas classes para cada tipo de falha incorporada. A adaptação de outras classes é simples, como a classe de gerenciamento de falhas, que exige apenas a alteração do interpretador do arquivo de falhas.

4.4. Análise de Intrusão

Um aspecto importante de um injetor de falhas é a intrusão temporal causada pelo mesmo. Para fazer esta medição, foi montado o cenário de um cliente e um servidor, onde apenas o cliente

envia mensagens com o protocolo UDP. Os experimentos foram executados em dois computadores AMD Athlon XP 2000+ com o sistema operacional Linux e 512MB de memória RAM. A máquina virtual usada foi a disponibilizada pela Sun, em sua versão 1.4.1. Os tempos foram medidos com o utilitário `time(1)`, que imprime o tempo de CPU em modo usuário e modo sistema usado por um determinado processo. As falhas foram injetadas no cliente de forma intermitente, seguindo o modelo de omissão. A taxa de falhas escolhida foi de 20%. Foram medidos os tempos com o cliente executando com e sem o injetor de falhas. No primeiro experimento, o cliente envia ao servidor 1 milhão de mensagens, com tamanho de 512 *bytes*. No segundo, são enviadas 10 milhões. Cada experimento foi repetido cinco vezes. A tabela 1 apresenta os tempos médios obtidos em segundos.

	Experimento 1		Experimento 2	
	modo usuário	modo sistema	modo usuário	modo sistema
sem injetor	1,911	2,795	17,239	26,086
com injetor	2,260	2,462	17,580	25,341

Tabela 1: Resultados de testes de intrusão temporal em segundos

Como pode ser visto na tabela, a intrusão apresentada por Jaca estendida com falhas de omissão é baixa. Os testes realizados também indicam que a intrusão não é crescente a medida que aumenta o número de mensagens, e que o custo de instanciação de Jaca tende a diminuir quando o tempo de execução é maior. Outro dado obtido é a diferença entre o tempo de execução em modo usuário e em modo sistema. Quando o programa executa com falhas sendo injetadas, o tempo de usuário é maior, pois há o tempo de gerenciamento do injetor e o de processamento das falhas, onde o programa é interceptado a cada envio de mensagem. No entanto, o tempo de sistema é menor, pois quando uma falha de omissão ocorre não há efetivamente o envio da mensagem, evitando a troca para modo sistema e o custo do envio em si.

5. Conclusão

O artigo apresentou a extensão do injetor de falhas Jaca para modelos de falhas de comunicação. Jaca é baseada em um sistema de padrões que já previa a expansão dos modelos de falhas, o que facilitou a adição de novas falhas. Em sua primeira versão, Jaca injetava falhas de atributos, parâmetros e retorno de métodos. Com esta extensão, Jaca também pode ser usada para conduzir experimentos de testes em aplicações distribuídas, uma vez que pode atuar diretamente na interface de programação de *sockets* Java.

Jaca baseia sua implementação em reflexão computacional, que apresenta várias vantagens para implementação de injetores de falhas por *software*. A principal é a divisão clara entre a aplicação e os mecanismos de injeção de falhas e monitoramento. A escolha do protocolo de reflexão Javassist também permitiu a manutenção da portabilidade de Jaca, pois não é necessário o uso de uma máquina virtual modificada.

Trabalhos futuros incluem a incorporação de um modelo de falhas para RMI e o uso de Jaca extendido para validar experimentalmente o sistema de comunicação de grupo JGroups [Ban 1998], que vem sendo bastante usado como bloco básico de aplicações distribuídas de alta disponibilidade.

Referências

Aidemark, J.; Vinter, J.; Folkesson, P.; Karlsson, J. *GOOFI: Generic Object-Oriented Fault Injection Tool*. In Proceedings of DSN'2001. Gotemburgo, Suécia. Julho 2001.

Ban, B. *JavaGroups - Group Communication Patterns in Java*. Department of Computer Science, Cornell University. Julho 1998.

Barcelos, P. P. A.; Leite, F. O.; Weber, T. S. *Building a Fault Injector to Validate Fault Tolerant Communication Protocols*. In Proceedings of International Conference on Parallel Computing Systems. Ensenada, México. Agosto 1999.

Carreira, J.; Madeira, H.; Silva, J. G. *Assessing the Effects of Communication Faults on Parallel Applications*. In Proceedings of IPDS'95. Erlangen, Alemanha. Abril 1995.

Carreira, J.; Silva, J. G. *Why do Some (weird) People Inject Faults?* ACM SIGSOFT, Software Engineering Notes, Volume 23, Número 1, pp. 42-43. Janeiro 1998.

Chiba, S. *Load-time Structural Reflection in Java*. In Proceedings of ECOOP 2000 - Object-Oriented Programming. LNCS 1850, Springer Verlag, pp. 313-336. 2000.

Cristian, F. *Understanding Fault-Tolerant Distributed Systems*. Communications of the ACM, Volume 34, Número 2, pp. 56-78. Fevereiro 1991.

Dawson, S.; Jahanian, F.; Mitton, T. *ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementations*. In Proceedings of IPDS'96. Urbana-Champaign, Estados Unidos. Setembro 1996.

Gonçalves, L. C. R.; Rodegheri, P. R.; Manfredini, R. A.; Weber, T. S. *Testing Fault Tolerance Mechanisms in DBMS Through Fault Injection*. In Proceedings of IEEE Latin-American Test Workshop. Cancun, México. Fevereiro 2001.

Hsueh, M.-C.; Tsai, T.; Iyer, R. *Fault Injection Techniques and Tools*. IEEE Computer, Volume 30, Número 4, pp. 75-82. Abril 1997.

Iyer, R. K. *Experimental Evaluation*. In Proceedings of FTCS-25. Pasadena, Estados Unidos. Junho 1995.

Leme, N. G. M; Martins, E.; Rubira, C. M. F. *A Software Fault Injection Pattern System*. In Proceedings of PLoP 2001. Monticello, Estados Unidos. Setembro 2001.

Maes, P. *Concepts and Experiments in Computational Reflection*. In Proceedings of OOPSLA 1987. Orlando, Estados Unidos. Outubro 1987.

Martins, E.; Rosa, A. C. A. *A Fault Injection Approach Based on Reflective Programming*. In Proceedings of DSN'2000. Nova York, Estados Unidos. Junho 2000.

Martins, E.; Rubira, C. M. F.; Leme, N. G. M. *Jaca: A Reflective Fault Injection Tool Based on Patterns*. In Proceedings of DSN'2002. Washington, Estados Unidos. Junho 2002.

Rosenberg, H. A.; Shin, K. G. *Software Fault Injection and its Application in Distributed Systems*. In Proceedings of FTCS-23. Toulouse, França. Junho 1993.

Schmidt, D. C. *Using Design Patterns to Develop Reusable Object-Oriented Communication Software*. Communications of the ACM, v. 38(10), pp. 65-74. Outubro 1995.

Stott, D. T.; Floering, B.; Burke, D.; Kalbarczyk, Z.; Iyer, R. K. *NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors*. In Proceedings of IPDS'2000. Chicago, Estados Unidos. Março 2000.

Voas, J. *Software Fault Injection: Growing 'Safer' Systems*. In Proceedings of IEEE Aerospace Conference 1997, v. 2, pp. 551-561. Aspen, Estados Unidos. Fevereiro 1997.

Um Modelo para Construção de Componentes Testáveis

Camila Ribeiro Rocha[1], Eliane Martins

Instituto de Computação – Universidade Estadual de Campinas (Unicamp)
Caixa Postal 6.176 – 13.083-970 – Campinas – SP – Brazil
{camila.rocha, eliane}@ic.unicamp.br

Abstract. *A software component must be tested every time it is reused, to guarantee the quality of both the component itself and the system in which it is to be integrated. To reduce the costs of the test phase, this article proposes a model to build highly testable components, embedding monitoring mechanisms and assertions, besides an infrastructure that generates test cases from an UML specification. Our approach proposes the insertion of the mechanisms directly into the intermediate code, allowing the creation of testable COTS components.*

Resumo. *Um componente de software deve ser testado a cada reutilização para garantir tanto sua qualidade quanto a do sistema na qual é integrado. Para diminuir os custos da fase de testes, este artigo propõe um modelo para construção de um componente com alta testabilidade, no qual são embutidos mecanismos de monitoração e assertivas, além da disponibilização de uma infraestrutura capaz de gerar casos de teste automaticamente a partir de uma especificação em UML. Nossa abordagem propõe a inserção dos mecanismos diretamente no código intermediário, possibilitando a criação de componentes COTS testáveis.*

1. Introdução

O desenvolvimento de software baseado em componentes vem sendo cada vez mais utilizado atualmente. Seu principal atrativo é a possibilidade de redução do tempo e custo de desenvolvimento, através da reutilização de código. Os componentes podem ser produzidos pela própria equipe de desenvolvimento ou adquiridos de terceiros (os chamados COTS – *Commercial off the Shelf*).

A garantia da qualidade, porém, continua dependente da realização de testes. Um componente precisa ser testado tanto isoladamente quanto a cada reutilização [Weyuker 98], pois pode acontecer que um componente funcione bem em um determinado contexto mas não em outros. Por isso, é importante que o componente tenha uma alta testabilidade, para que a fase de testes não seja muito onerosa.

Foram propostas diversas técnicas para a construção de componentes testáveis [Hörnstein e Edler 2002, Ukuma 2002, Gao et. al. 2002]. Este artigo apresenta uma

[1] Camila Ribeiro Rocha recebe apoio financeiro da CAPES.

proposta de modelo para a construção de um componente testável. A estrutura deverá ser desenvolvida pelo fornecedor do componente e disponibilizada para seus usuários. O componente testável inclui mecanismos de monitoração e assertivas, além da geração automática de casos de teste a partir de sua especificação, permitindo a utilização da técnica de teste caixa preta.

Algumas destas abordagens introduzem mecanismos no componente que facilitam a fase de testes, embutindo esses mecanismos diretamente no código fonte [Hörnstein e Edler 2002, Ukuma 2002, Wang et. al. 1999]. Nossa abordagem propõe sua inserção no código intermediário, deixando separados os aspectos funcionais e os aspectos de testes e tornando opcional a permanência dos aspectos de teste no componente operacional. Além disso, a independência do código fonte desobriga o fornecedor de disponibilizá-lo para a inclusão dos mecanismos de teste, permitindo a distribuição do componente testável na forma de um COTS.

O texto está organizado da seguinte maneira: a seção 2 lista os problemas relacionados a testabilidade de componentes, e algumas abordagens para solucioná-los; a seção 3 apresenta a estrutura proposta para a construção de um componente testável; a seção 4 descreve aspectos considerados no projeto do componente testável; a seção 5 descreve sucintamente a implementação da estrutura em um estudo de caso simples, exemplificando a aplicação de um caso de teste; a seção 6 apresenta as conclusões e perspectivas futuras deste trabalho.

2. Testabilidade de Componentes

Segundo Szyperski [Szyperski 1998], “um componente de software é uma unidade de composição com interfaces especificadas através de contratos e dependências de contexto explícitas, que pode ser desenvolvido independentemente e ser utilizado para a composição de sistemas de terceiros”. Um sistema baseado em componentes é composto por componentes que interagem entre si para fornecer as funcionalidades desejadas.

A principal vantagem da utilização de componentes é a reutilização de código, que proporciona redução no tempo e custo do desenvolvimento. A reutilização deve ser acompanhada por uma fase de testes. Contudo, tanto o fornecedor quanto o usuário de um componente COTS enfrentam dificuldades na realização dos testes, que podem ser caracterizadas como falta de conhecimento [Beydeda e Gruhn 2003]. Falta de conhecimento por parte do fornecedor, que não conhece todos os diferentes contextos de utilização de um componente e por isso assume algumas hipóteses sobre seu contexto de uso, que direcionam a realização dos testes. Esse direcionamento torna imprescindível que o usuário valide o componente no seu próprio contexto de utilização, que pode não ter sido priorizado pelo fornecedor.

Falta de conhecimento também por parte do usuário, que conhece apenas a interface pública do componente, através da qual é feito o acesso às funções oferecidas. O código fonte, seja para a geração de testes (em especial os de caixa branca), seja para a depuração após a realização dos testes, não é disponibilizado. Mesmo a realização de testes de caixa preta é prejudicada quando a documentação sobre o componente é incompleta.

Por isso, é muito importante que o componente seja testável, diminuindo o custo da fase de testes. O conceito de testabilidade é relativo à facilidade e ao custo de se encontrar falhas em um *software* [Binder 1994], e está ligado à sua capacidade de ser controlado e

observado durante os testes. Aspectos como métodos BIT (do inglês *Built-in Test*), que relatam o estado do componente, e assertivas, que definem pré e pós condições para métodos, contribuem para o aumento da testabilidade de um componente.

Vários trabalhos vêm sendo elaborados para minimizar os diferentes problemas citados. Dentre eles, as abordagens *Component+* [Hörnstein e Edler 2002], *Component Test Bench* [Bundell et. al. 2000] e *Testable Beans* [Gao et. al. 2002].

A abordagem *Component+* busca facilitar a realização de testes pelo cliente do componente, propondo uma estrutura para um componente testável formada por três subcomponentes:

- Componente a ser testado (*Component BIT*), que além da implementação das funcionalidades possui mecanismos BIT, para observação de seu estado;
- Componente *Tester*, que armazena casos de teste para o *Component BIT* e possui a capacidade de realizá-los e avaliá-los, utilizando as interfaces BIT;
- *Handlers*, utilizados para implementar mecanismos de tolerância a falhas.

A proposta de [Bundell et. al. 2000] é a ferramenta *Component Test Bench* (CTB), que auxilia os fornecedores na geração de casos de testes e os usuários na aplicação dos testes de forma automatizada. A ferramenta auxilia a construção de uma especificação em formato XML (*Extensible Markup Language*) contendo a descrição das implementações do componente e um conjunto de casos de teste para cada interface. Além disso, são disponibilizadas funcionalidades como a geração automática de oráculos (resultados esperados dos casos de teste). Na entrega ao usuário, o componente é empacotado juntamente com sua especificação de testes e um módulo da ferramenta, possibilitando a realização dos testes no novo ambiente.

Ambas as abordagens têm como vantagem dispensar uma infraestrutura para a realização dos testes, já que os artefatos são empacotados junto ao componente. Esses artefatos não constituem um *overhead* ao componente operacional, pois podem ser desacoplados. Outra vantagem é permitir ao cliente a customização dos casos de teste, com a disponibilização do código fonte do componente *Tester*, no caso da abordagem *Component+*, e da especificação de testes em formato XML, na CTB.

Entretanto, a customização na abordagem *Component+* pode ser prejudicada pela dificuldade no entendimento do comportamento do componente, já que a presença da especificação não é prevista pela arquitetura. Já a CTB prevê uma especificação que traz informações sobre as implementações e interfaces do componente. Uma limitação de ambas é que a redução no custo da fase de testes é pequena para o fornecedor, pois não há geração automática de casos de teste em nenhuma das abordagens (apesar da CTB oferecer mais facilidades).

Na abordagem proposta por [Gao et. al 2002], a principal preocupação é o aumento da testabilidade. É criado o conceito de *testable bean*, um componente de *software* que incorpora mecanismos padronizados que facilitam a realização de testes e a monitoração de sua execução.

Ao componente é incorporada uma interface de testes, que possui métodos para a inicialização, execução e avaliação dos casos de teste, além de fornecer informações sobre classes e métodos do componente utilizando reflexão computacional. Também é incorporada uma interface de monitoração, que permite o rastreamento de sua execução. A

principal desvantagem do método é não oferecer facilidades para a criação de casos de teste, sendo recomendado o uso de ferramentas especializadas.

A proposta aqui apresentada é uma extensão dos trabalhos [Toyota 2000, Ukuma 2002], em que um componente testável contém, além dos mecanismos embutidos de teste e monitoração presentes na abordagem anterior, um modelo de especificação a partir do qual é possível a geração automática de testes. Diferentemente da abordagem apresentada em [Toyota 2000], em que o componente era considerado como uma única classe, aqui um componente pode ser constituído de diversas classes. Nossa abordagem estende o trabalho de [Ukuma 2002] de modo a considerar notações UML para modelagem e a inclusão de mecanismos de monitoração no componente.

Em relação às abordagens apresentadas, nossa proposta une as vantagens das três: a modularidade da *Component+*, que separa os aspectos de testes das funcionalidades; o acoplamento da especificação ao componente, como na *Component Test Bench*; a presença de funções de monitoração, como no *Testable Bean*; a possibilidade do desacoplamento dos mecanismos de teste, apresentado nas três abordagens. Outra vantagem da nossa abordagem é a geração automática de casos de teste a partir da especificação e a possibilidade de customização dos casos de teste. Naturalmente que a geração necessita de uma infraestrutura auxiliar.

De acordo com Gao et al. a construção de componentes testáveis deve levar em conta os seguintes aspectos [Gao et. al. 2002]: 1º) definir uma arquitetura e uma interface de testes que seja comum a todos os componentes testáveis; 2º) estabelecer uma forma sistemática para a geração desses componentes; 3º) definir como minimizar o *overhead* dos mecanismos embutidos e 4º) oferecer recursos que apóiem os testes desses componentes. Neste texto focaremos os pontos 1 e 3.

3. Arquitetura

O objetivo da abordagem proposta é o aumento da testabilidade de um componente para a aplicação de testes caixa preta, realizados tanto pelo fornecedor quanto pelo usuário. Para isso, é acoplada ao componente, além de mecanismos de testes e monitoração, sua especificação na forma de assertivas e de um modelo de comportamento. As primeiras servem como oráculo e o segundo é usado por uma infraestrutura capaz de gerar casos de teste de forma automática.

A arquitetura do componente é ilustrada na Figura 1. O componente testável, além do componente a ser testado e suas interfaces públicas (“Interface A” e “Interface B”), é composto por dois outros subcomponentes:

- “Tracker”, responsável pelas funcionalidades de monitoração, que implementa as interfaces ITracking (pública) e ILogTrace (visível internamente ao pacote);
- “Tester”, responsável pela inclusão de assertivas e pela disponibilização da especificação, implementa as interfaces ITesting (pública) e IAClasse_metodo1 ... IAClasse_metodoN (visível internamente ao pacote). As interfaces IAClasse_metodo serão criadas de acordo com o número de métodos públicos oferecidos pelo componente. O componente Tester também armazena o arquivo contendo a especificação do componente, construída separadamente, com a utilização da UML (*Unified Modeling Language*).

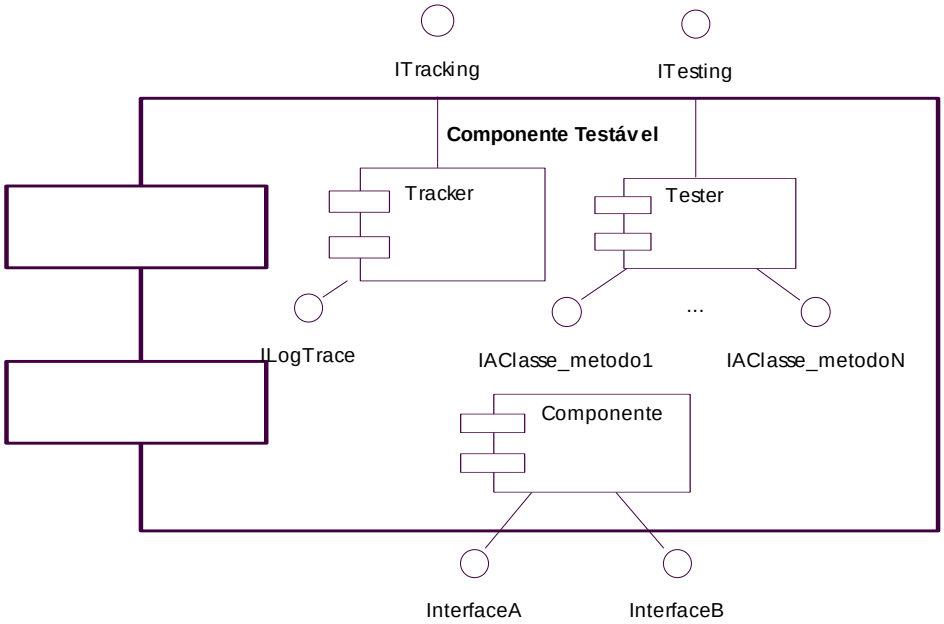

Figura 1. Estrutura interna do componente testável

3.1. Interfaces de Monitoração

Toda a interação com o componente testável é realizada através de interfaces. O subcomponente Tracker disponibiliza operações para a inserção de mecanismos de monitoração no componente. Os tipos de monitoração disponíveis são [Gao, Zhu and Shim 2000]: operacional, que rastreia as interações entre as operações do componente; de erros, que rastreia as mensagens de erro geradas durante a execução; e de estado, que monitora o valor dos atributos públicos do componente. A monitoração de performance não será disponibilizada inicialmente pois os testes serão voltados apenas para a verificação do correto comportamento do componente. A monitoração de eventos também não é considerada, já que é direcionada a um tipo específico de componentes (componentes GUI – para interface com o usuário). Entretanto, a arquitetura pode ser estendida para a incorporação desses tipos de monitoração.

A inclusão de mecanismos de monitoração é realizada no código intermediário do componente, através da interface ITracking. Após sua instrumentação, isto é, após a inclusão dos mecanismos, a execução do componente passa a ser acompanhada de acordo com o tipo de monitoração selecionado.

A interface ITracking disponibiliza as seguintes operações:

– `monitorOperacional(metodo : String, modo_chamada : char)`: o método passado como parâmetro será monitorado em relação as chamadas recebidas e realizadas (caso seu valor seja *null*, todos os métodos públicos serão rastreados). Caso o valor do parâmetro “modo_chamada” seja “e”, apenas as chamadas recebidas pelo método serão registradas no *log*; caso seja “s”, o registro será apenas das chamadas realizadas pelo método a outros métodos públicos do componente; caso seja “a”, todas serão registradas.

– `monitorEstado(atributos : String [])`: esse método habilita a monitoração dos atributos públicos do componente, que são passados como

parâmetro (caso “atributos” = *null*, todos os atributos públicos serão monitorados). O registro no *log* acontece a cada atualização.

- `monitorErro(excecao : String)`: esse método habilita a monitoração da exceção passada como parâmetro (caso “exceção” = *null*, todas as exceções serão monitoradas), registrando no *log* as informações sobre qual o tipo da exceção lançada, o método que a lançou, as classes que atravessou (através de *throw*), e a classe e método que a tratou.
- `setLog(arquivo: File)`: atribui o caminho do arquivo de *log*, onde as violações serão registradas.

A interface ILogTrace é visível apenas internamente. É invocada pelo componente sob teste após sua instrumentação, para o registro da monitoração. Disponibiliza as seguintes operações de registro no *log*:

- `incluirTraceErro(excecao : String, lançador : String)`: esse método registra o lançamento de uma exceção.
- `incluirTraceErro(excecao : String, tratador : String, sucesso : boolean)`: registra a tentativa de tratamento de uma exceção, e se o tratamento foi bem sucedido.
- `incluirTraceEstado(atributos : String [])`: registra o novo estado dos atributos.
- `incluirTraceOperacao(modo_chamada : char, chamado : String, chamador : String, parâmetros : String)`: registra a chamada de um método, além das informações sobre o método que o chamou, a hora que aconteceu a chamada, e os valores de parâmetros e de retorno.

3.2. Interfaces de Testes

O subcomponente Tester disponibiliza as interfaces ITesting, que oferece métodos para inclusão de assertivas no componente sob teste e para obtenção de sua especificação, e IAClasse_metodo1 ... IAClasse_metodoN, acessíveis apenas internamente ao componente testável e utilizadas pelo componente sob teste após sua instrumentação para verificação das assertivas. Cada interface IAClasse_metodo está relacionada a um método público do componente sob teste, sendo nomeada de acordo com a classe e método correspondentes. Por exemplo, se estiver relacionada ao método pop() da classe Pilha, será nomeada como “IAPilha_pop”.

A estrutura interna do componente Tester é ilustrada na Figura 2 e detalhada nas subseções a seguir. Foi incluída apenas uma interface IAClasse_metodo por simplicidade. O esteriótipo “+” indica visibilidade pública, e sua ausência visibilidade de pacote.

3.2.1 Interface ITesting

A interface ITesting disponibiliza as seguintes operações para inclusão de assertivas no componente sob teste: `incluirPreCondicao(metodo : String, continua: boolean)`, `incluirPosCondicao(metodo : String, continua: boolean)` e `incluirInvariante(metodo : String, continua: boolean)`. Os métodos, respectivamente, habilitam as pré-condições, pós-condições e invariantes do método passado como parâmetro (caso o valor de “método” seja *null*, as

assertivas são incluídas em todos os métodos para os quais foram especificadas). O parâmetro “continua” determina se a execução do sistema deve continuar após a violação de uma assertiva ou deve ser interrompida. Quando uma assertiva é violada, essa informação é registrada no *log*. O caminho para o arquivo de *log* é atribuído através da operação `setLog(arquivo: File)`, também oferecida pela interface ITesting.

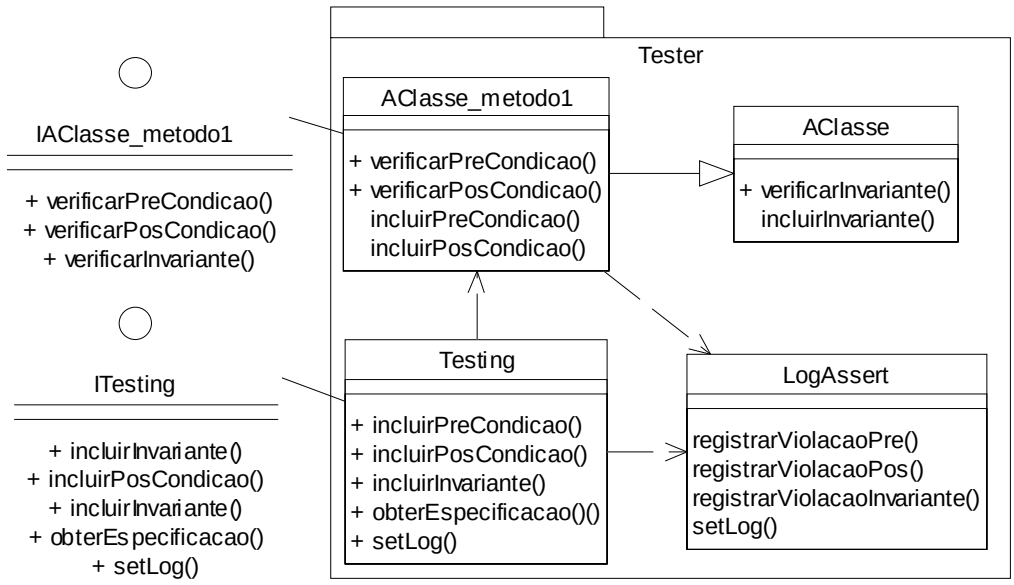

Figura 2. Estrutura interna do componente Tester

A interface ITesting é implementada pela classe Testing. As operações `incluirPreCondicao`, `incluirPosCondicao` e `incluirInvariante` delegam a inclusão dos mecanismos no componente sob teste à classe AClasse_metodo correspondente ao método solicitado. A operação `setLog` é repassada para a classe LogAssert.

Além dos métodos relativos às assertivas, a interface ITesting também oferece a operação `obterEspecificacao(): File`, que retorna o arquivo de especificação contido no componente Tester. Esse arquivo traz informações sobre a estrutura do componente, seu comportamento e as assertivas referentes a seus métodos públicos. O comportamento do componente é representado pelo diagrama de atividades da UML [Booch et. al. 2000], que contém informações sobre como os métodos interagem entre si, definindo em qual ordem eles devem ser chamados. As assertivas são registradas na linguagem OCL (*Object Constraint Language*) [Warner e Kleppe 1999], e originam as assertivas que serão embutidas no componente pelos métodos da interface ITesting.

A especificação será disponibilizada em forma de arquivo, e servirá de entrada para as ferramentas de geração automática de casos de teste e de tradução de assertivas em OCL para o formato a ser embutido no código dos métodos. A ferramenta ainda não foi construída, mas será baseada na ferramenta ConCAT, projetada por [Toyota 2000], que automatiza o processo de construção de um *driver* de teste para a classe a ser testada.

Um *driver* de teste é responsável por ativar o componente sobre teste, fornecendo as entradas e outras informações necessárias para a execução de casos de teste. A ferramenta ConCAT consiste em um *Driver Genérico* capaz de gerar um *driver* específico para o componente a partir de sua especificação [Toyota 2000]. O *driver* específico é um

conjunto executável de casos de teste, responsável por aplicar os testes e armazenar os resultados. São embutidas assertivas no código do componente que servem como oráculo para os casos de teste, isto é, determinam se uma saída obtida é correta para uma entrada determinada.

O *Driver Genérico* será capaz de obter um *driver* específico para o componente através do diagrama de atividades presente na especificação. Por gerar casos de teste a partir da especificação, a ferramenta dispensa a necessidade do código fonte para a geração dos casos de teste, e permite a aplicação de critérios de seleção de caminhos no diagrama de atividades. Além disso, como o código dos casos de teste estará disponível, o cliente poderá customizá-lo de acordo com suas necessidades.

O *Driver Genérico* não fica embutido no componente, diminuindo assim o espaço de armazenamento necessário. Apenas uma cópia deve ser mantida pelo cliente, e esta servirá para quaisquer componentes que tenham a especificação no formato exigido. Esse *driver* é independente, portanto, do componente em teste, e pretende-se que seja também independente do código em que o componente é escrito.

3.2.2 Interfaces IAClasse_metodo

As interfaces IAClasse_metodo devem ser construídas de acordo com os métodos públicos do componente sob teste. Suas operações são invocadas pelo componente sob teste após sua instrumentação, isto é, após a inclusão dos mecanismos para verificação de assertivas.

Cada método público do componente deve ter uma interface IAClasse_metodo correspondente, a qual oferece os métodos `verificarPreCondicao()`, `verificarPosCondicao()` e `verificarInvariante()`. Esses métodos são chamados pelo componente sob teste instrumentado durante sua execução, para verificar se a assertiva correspondente está sendo violada de acordo com os valores de seus parâmetros. Os parâmetros variam para cada interface, pois os métodos recebem os valores a serem verificados pelas assertivas.

As interfaces IAClasse_metodo são visíveis apenas internamente ao componente testável, e cada uma é implementada por uma classe AClasse_metodo, que por sua vez é descendente da classe AClasse. A superclasse AClasse é responsável pela implementação do método `verificarInvariante()`, já que a invariante de uma classe é a mesma para todos os seus métodos.

Caso os métodos `verificarPreCondicao()`, `verificarPosCondicao()` e `verificarInvariante()` verifiquem a violação da assertiva, o registro no *log* é realizado pelos métodos da classe LogAssert `registrarViolacaoPre (metodo : String, mensagem : String, continua : boolean)`, `registrarViolacaoPos (metodo : String, mensagem : String, continua : boolean)` e `registrarViolacaoInvariante (metodo : String, mensagem: String, continua : boolean)`. A violação é registrada no arquivo *log*, com o método no qual ocorreu e uma mensagem detalhando as causas do erro. O parâmetro "continua" indica se a execução deve continuar após a violação, e é determinado pelo método de inclusão de assertivas.

Além dos métodos de verificação de assertivas, as classes AClasse_método implementam os métodos `incluirPreCondicao()` e `incluirPosCondicao()`

(o método `incluirInvariante()` é implementado pela superclasse). Esses métodos são chamados pela classe Testing, e são responsáveis pela instrumentação do componente sob teste, incluindo respectivamente as pré, pós condições e invariantes no código intermediário do método correspondente.

4. Projeto Detalhado

A principal preocupação no projeto detalhado da estrutura do componente testável foi permitir que os mecanismos de monitoração e testes fossem desacoplados quando o componente entrasse em operação (inclusive a instrumentação inserida dentro do componente), já que eles serão utilizados apenas nas fases de teste.

Em [Gao, Zhu e Shim 2000] são apresentadas três maneiras para a inclusão de mecanismos de monitoração nos componentes (que podem ser estendidos para os mecanismos de teste): *Framework-based tracking* (monitoração baseada em *framework*), que orienta o fornecedor na adição dos mecanismos diretamente no código fonte; *Automatic code insertion* (inserção automática no código), na qual uma ferramenta adiciona os mecanismos ao código fonte do componente de forma automática; e *Automatic component wrapping* (empacotamento automático do componente), que adiciona mecanismos na forma de um *wrapper* para monitoração das interfaces externas do componente, sem necessidade do código fonte.

Como a terceira abordagem não é indicada para monitoração de erros, já que esta é altamente dependente das regras de negócio específicas da aplicação, a abordagem de inserção automática do código foi adaptada para componentes COTS, com a inclusão dos mecanismos no código intermediário do componente através de bibliotecas especializadas para sua manipulação. Outra opção para a inserção dos mecanismos seria a orientação a aspectos, usando, por exemplo, AspectJ [Kiczales et. al 2001], mas essa opção foi preterida pois seria necessário o acesso ao código fonte do componente.

A utilização de bibliotecas para manipulação do código intermediário tem como principal vantagem desobrigar o fornecedor a disponibilizar o código fonte do componente e ao mesmo tempo, oferecer ao usuário a opção de escolher os mecanismos a serem incluídos no componente para os testes e retirá-los quando preferir usar menos espaço para armazenamento. Essa inclusão pode ser realizada em tempo de carga, ou diretamente no código intermediário do componente, criando uma nova cópia instrumentada. Por ser um trabalho em andamento, a forma de inclusão ainda não foi definida.

5. Implementação

Os primeiros estudos de caso foram realizados em componentes implementados na linguagem Java. Após a compilação, é gerado um arquivo *.class* para cada classe Java. Os arquivos *.class* têm o formato de *byte code*, e são interpretados pela Máquina Virtual Java durante a execução. Assim, para a manipulação dos *byte codes* foi utilizada a biblioteca BCEL (do inglês *Byte Code Engineering Library*) [Dahm 2001].

A BCEL é uma biblioteca que permite a análise estática e a criação e modificação dinâmicas de arquivos *.class* Java. São disponibilizados vários tipos de operações, que auxiliam a modificação de uma classe [Dahm 2001]. É possível obter o código intermediário de método na forma de uma lista de instruções, que pode ser manipulada através de operações disponibilizadas pela biblioteca. A desvantagem da utilização da

biblioteca BCEL é sua complexidade, pois obriga o desenvolvedor a entender a estrutura e funcionamento do *byte codes*, mesmo que superficialmente.

Até o momento foram implementados apenas protótipos simples, mas que permitem comprovar a eficiência da biblioteca BCEL na inserção de assertivas e mecanismos de monitoração em um componente mesmo sem o código fonte, mostrando a viabilidade da arquitetura proposta.

O primeiro exemplo implementado foi o de uma Pilha de objetos do tipo *Object*. Por questão de espaço, não é possível apresentar sua especificação completa, nem detalhar sua instrumentação. Assim, serão descritos apenas os passos para a execução de um caso de teste, e qual foi o comportamento do componente instrumentado.

Como a ferramenta para a geração dos casos de teste ainda não foi implementada, foi construída a classe `Driver` para testar o comportamento do método `Pilha.pop()` quando a pilha está vazia. Para a realização do teste, o método foi instrumentado para inclusão de sua pós condição (o tamanho da pilha deve ter descrescido em uma unidade) e da invariante da classe (a pilha deve conter entre zero e cem elementos). A Figura 3 apresenta o código do caso de teste, que inicialmente requisita à interface ITesting a inclusão da pós condição e invariante, atribui o caminho do arquivo de *log*, e em seguida realiza o teste. Pode-se constatar o aumento da testabilidade do componente pois tanto para a instrumentação do componente quanto para a aplicação dos testes foi simplificada.

```
public class Driver {

  public static void main (String args[]) {
      Tester.ITesting teste = new Testing();
  //inclusão da pós condição do método pop
      teste.incluirPosCondicao ("pop", false);
  //inclusão da invariante no método pop
      teste.incluirInvariante ("pop", false);
      teste.setLog(“Pilha_vazia.log”);
      Componente.Pilha pilha = new Componente.Pilha();
      pilha.pop();
  }
}
```

Figura 3. Código fonte da classe Driver, que implementa um caso de teste

A invariante é verificada no início e no fim do método, mesmo que uma exceção seja lançada. Já a pós condição é verificada apenas ao final do método se nenhuma exceção tiver sido lançada. Como nenhum tipo de monitoração foi habilitado, a execução do caso de teste gera um *log* como ilustrado na Figura 4. Se outros casos de teste fossem executados seqüencialmente, as violações ocorridas nas assertivas continuariam sendo registradas no mesmo arquivo, assim como possíveis monitorações.

```
Invariante violada: Metodo Pilha.pop
Mensagem: Indice vetor = -1
```

Figura 4. Arquivo de *log* após a execução do caso de teste

6. Conclusões e Perspectivas

O objetivo principal deste trabalho é facilitar a realização de testes em componentes, diminuindo os custos da fase de testes. A testabilidade do componente é aumentada com a inclusão de assertivas e mecanismos de monitoração em seu código, além da disponibilização de uma infraestrutura para a geração automática de casos de teste a partir de uma especificação do comportamento do componente no formato do diagrama de atividades da UML.

A utilização da estrutura traz benefícios principalmente para o usuário do componente, que consegue testá-lo em um tempo reduzido sem prejudicar a eficiência do componente em tempo de operação. O fornecedor do componente também é beneficiado, já que os casos de teste são gerados de forma automática. A principal limitação do modelo é se tornar muito extenso caso o componente tenha um grande número de métodos públicos. O tamanho, porém, não é considerado um *overhead* em tempo de operação, já que todos os mecanismos podem ser desacoplados.

Como o trabalho está em fase inicial, as etapas realizadas foram a definição da arquitetura, de uma interface de testes e de uma maneira para minimizar o *overhead* dos mecanismos embutidos, através da inserção das assertivas e mecanismos de monitoração no código intermediário do componente. Foram implementados alguns protótipos simples, utilizando a linguagem Java e a biblioteca para manipulação de *byte codes* BCEL, que verificaram a viabilidade do modelo.

As próximas atividades do trabalho incluem a realização das duas outras etapas na construção de componentes testáveis [Gao et. al. 2002]: o estabelecimento de uma forma sistemática para a geração dos componentes testáveis e o oferecimento de recursos que apóiem os testes desses componentes. A geração dos componentes será sistematizada com a elaboração de um método para construção de componentes testáveis que possa ser acoplado a um método de desenvolvimento. Já os recursos de apoio serão formados por ferramentas que automatizem a geração e aplicação dos casos de teste e facilitem a criação dos componentes Tester e Tracker.

Referências

Beydeda, S. e Gruhn, V. (2003) “State of the art in testing components”. In: 3rd International Conference on Quality Software.

Booch, G., Rumbaugh, J. e Jacobson, I. (2000) UML: Guia do Usuário. Ed. Campus.

Bundell, G., Lee, G., Morris, J., Parker, K. (2000) “A Software Component Verification Tool”. In: Proceedings of International Conference on Software Methods and Tools.

Dahm, M. (2001) “Byte Code Engineering with the BCEL API”. Technical Report B-17-98, Freie Universität at Berlin, Institut für Informatik.

Gao, J., et. al. (2002) “On building testable software components”. In: Lecture Notes in Computer Science, V. 2255, pp. 108-121, Springer Verlag.

Gao, J., Zhu, E., e Shim, S. (2000) “Monitoring Software Components and Component-Based Software”. In: IEEE Computer Software and Application Conference (COMPSAC).

Hörnstein, J. e Edler, H. (2002) “Test reuse in CBSE Using Built-in Tests”. In: Workshop on Component-Based Software Engineering, Composing Systems from Components.

Kiczales, G., et. al. (2001) “An Overview of AspectJ”. In: Lecture Notes in Computer Science, V. 2072, p. 327-353, Springer Verlag.

Szyperski, C. (1998) Component Software: Beyond Object-Oriented Programming, Addison-Wesley.

Toyota, C. (2000) Melhoria da Testabilidade de Classes Usando o Conceito de Autoteste. Campinas: Instituto de Computação da UNICAMP. 119 p. (Dissertação, Mestrado em Ciência da Computação).

Ukuma, L. (2002) Uma Estratégia para o Desenvolvimento de Componentes de Software Autotestáveis. Campinas: Instituto de Computação da UNICAMP. 144 p. (Dissertação, Mestrado em Ciência da Computação).

Wang, Y., King, G. e Wickburg, H. (1999) “A Method for Built-in Tests in Component-based Software Maintenance”. In: IEEE International Conference on Software Maintenance and Reengineering (CSMR'99), p. 186-189.

Warmer, J e Klepper, A. (1999) “OCL: The constraint language of the UML”. Journal of Object-Oriented Programming, May.

Weyuker, E. (1998) “Testing Component-Based Software: A Cautionary Tale”, In: IEEE Software, 15(5): 54-59, September/October 1998.

STAGE: an Integrated Environment for Statistical Test Script Generation[1]

Bernardo Copstein, Flávio Oliveira, Lucas R. C. Reginato

CPTS/FACIN – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
90.616-900 – Porto Alegre – RS – Brazil

{copstein,flavio}@inf.pucrs.br, reginato@cpts.pucrs.br

Abstract. *This work describes STAGE, an integrated environment for statistical test case generation and scripting, developed at the CPTS (Software Testing Research Center). The key contributions of the system are the use of stochastic automata networks (SANs) for representing the usage model and an intermediate model (called the Interface Event-State Model) to map the abstract usage model into the implemented interface components. SANs allow modular representation of systems with complex non-deterministic behavior, minimizing the state-space explosion found in Markov chains. The use of a separate model for the implementation has the advantage of making changes in the interface and generating the script automatically without affecting the abstract usage model. We implemented the technique into the STAGE environment; the experiments indicate that the performance of test case and script generation with SANs is at least compatible with Markov-based generation.*

1. Introduction

The interest on statistical testing has increased in recent years, because of two main reasons: the growing complexity and non-determinism of the environments on which modern software is expected to operate (the Internet is just the paradigmatic and most popular example) and the higher levels of reliability required for complex mission-critical applications. Indeed, reliability assessment is crucial for systems with safety and fault-tolerance requirements. Such demands are difficult to address with traditional testing techniques. Even when one considers a more deterministic environment, the complexity of the software itself and its interface may require that the test engineer develop the test cases from a complex model describing the application. In both cases, the use of automated tools is inevitable. Such applications are simply too complex to test using only manual tests. Which poses another problem: how to create test scripts with hundreds or thousands of tests in an effective way? Moreover, how to update such scripts efficiently when there is a change in the product? Clearly, there is much to be gained from the possibility of generating the test cases automatically from a high-level model of the application, and then creating the test scripts automatically from the test cases.

Technology for statistical testing with usage models is mostly based in Markov chains. However, a Markov chain for a complex application can easily reach hundreds or thousands of states, which makes the modeling and maintenance a difficult task.

[1] This work is supported by the HP-PUCRS Cooperation Agreement 01/99 - TA 17 (CPTS project).

Moreover, in order to generate the test script, it is necessary to add implementation-related information to the usage model.

We present here STAGE (STAte-based test GEnerator), a test script generation toolkit for statistical testing, developed at CPTS (Software Testing Research Center), a cooperation project involving PUCRS and HP. The key features of the system are the use of stochastic automata networks (SANs) for representing the usage model and an intermediate model (called the Interface Event-State Model - ISEM) to map the abstract usage model into the implemented interface components. SANs allow modular representation of systems with complex non-deterministic behavior, minimizing the state-space explosion found in Markov chains. The use of a separate model for the implementation has the advantage of making changes in the interface and generating the script automatically without affecting the abstract usage model. STAGE also includes features for deterministic testing, based on finite state machines. In this way, one can combine deterministic and statistical testing, using finite-state models as an unifying concept. The choice of state-based testing is motivated by two main characteristics: (a) state diagrams are widely used in software modeling and design; (b) there are state-based formalisms that support both deterministic functional testing and statistical testing. The result is a robust and flexible tool with a unified set of modeling primitives, simplifying the task of the test engineers.

In this paper, we focus on the features introduced in STAGE for statistical testing. In section 2, we describe the issues involved in usage modeling in general and present the SAN and ISEM models. Section 3 describes the STAGE environment. Section 4 comments on related work, while in section 5 we give our conclusions so far and comment on future directions of our research.

2. State-Based Statistical Testing

Statistical Testing is a wide concept. If some aspect of a system, in any level of abstraction, could be represented using events whose occurrence rules could be represented by a statistical model, so we can do statistical testing. The major advantage of using a statistical approach for testing is that in doing so is possible to estimate parameters as, for example, reliability, what are not possible in traditional testing. One of the possible approaches for statistical testing is those based on *usage models*.

2.1 Usage Models

A usage model characterizes the operational use of a software system. Software is used by a user on a specific environment [8, 10], where the user may be a person, a hardware device, other software, or a group of users. A software use may be a working session, transaction or any service unit limited by a start/finish event.

The model structure is composed of a state set and the transitions between these states, constituting a graph. The graph nodes represent the model states, and the graph arcs represent the transitions between states. This structure describes the possible uses of the software. A probability distribution, associated with the model, describes the expected use of the software.

Usage models are represented with some system modeling formalism [8] – discrete Markov chains are the usual choice. In this formalism, the usage states of the software are mapped as chain states, and the user actions are mapped as transitions between these

chain states. There are two other special states: *invoke* – that represents the beginning of the program execution – and *terminate* – that represents the end of the program execution. These are the only start and finish states of the chain, respectively. The transition probabilities or rates represent the user action's probabilities, configuring the typical uses of the software.

The usage model may be applied at many stages of the software life-cycle, in order to refine system specification, evaluate complexity, drive the verification efforts, identify some event's frequency, project the testing chronogram, estimate software reliability, and so on. We focus here on its application to the testing phase.

When represented with Markov chains, usage models allow the test engineer – a person who has the responsibility of test's creation and management – to predict the critical paths, more susceptible to failure, concentrating the efforts in this context. This analysis is performed over the occurrence probabilities associated to each use of the software.

From the usage model analysis it is possible to extract several interesting properties, such as [11]:

- Number of software statistically typical usage paths;
- Long-run occupancy, e.g. utilization time percentage for each state;
- Mean number of events per test case;
- Mean number of events between two states, etc.

2.2. Markov Chains

A stochastic process is specified as a family of random variables defined in a probabilities space and indexed by a parameter. Usually this parameter refers to an index set of the process time or time range. If we use discrete time (ex: $T = \{1,2,3,...\}$) we will have a discrete stochastic process, but if we use continuous time (ex: $T = 0 < t < +\infty$) so the stochastic process will be continuous. A markovian process is a stochastic process where the distribution probability function assumes the main property of a markovian process, that is, the process evolution depends only upon its current state, being completely independent of the system previous states.

When the space of states of a markovian process is discrete, the process is called a *Markov Chain*. This kind of chain is classified according to a time scale as Discrete Time Markov Chains (DTMC) and Continuous Time Markov Chains (CTMC).

Using a discrete chain (DTMC) the triggering of transitions from a state to another is ruled by conditional probabilities. These probabilities are denoted by a real number in the [0;1] range and the sum of all transition probabilities from a state to another must be equal to 1.

2.3. Stochastic Automata Networks

We describe here a formal conceptualization of Stochastic Automata Networks (SANs). The treatment given here is somewhat different from the original presentation, given in the works of Fernandes [2]. The goal here is to provide a basis for understanding the approach and data model of STAGE-Test.

Given a set S of states (the local states), let A = { A_1, A_2, … A_n } be a set of *automata*, where each automaton A_i is a subset of S. A **Stochastic Automata Network (SAN)** is a structure

(G,E,Pe,Pt,I)

where:

- G = { G_1, G_2, … G_m } is a set of **global states**, such that each Gi is an element of A_1 X A_2 X … X A_n. In other words, each global state is a combination of local states of the automata.
- E = { E1, E2, … Ek } is a set of **events**. Each event is a function Ei: G → ***P***(G). In other words, each event maps global states into sets of global states. When the event is fired, the SAN can go to any element of the set specified by the function, depending on the probabilities assigned to the event (see below). Since a global state is a list of local states, the function describes, for each automaton Ai in the network, what happens in that automaton when the event is fired. Events can be classified as *local* and *synchronizing.* A local event changes the state of only one automaton; a synchronizing event changes the state of two or more automata.
- Pe = { P1, P2, … Pk } is a set of **event probability functions**, one for each event. Each function Pi: G → ℝ describes the probability of occurrence of the event at each global state.
- Pt = { Pt1, Pt2, … Ptkm } is a (possibly empty) set of **transition probability functions,** one for each pair (event, global state). As defined above, for an arbitrary event **i**, when the SAN is in a global state S and the event is fired, the SAN goes to a state S' which must be an element of Ei(S). The transition probability functions describe the probabilities of the different elements of Ei(S) being selected. Usually, Ei(S) has only one state, so the definition of these probabilities is optional.
- I ⊆ G is a (possibly empty) set of **initial states.** In the original definition, SANs do not have initial states; they are useful for the specification of usage models. Since they are optional, this definition includes the original as a special case.

Any SAN can be converted into an equivalent Markov chain [2]. The states of the Markov chain are the global states of the SAN.

Stochastic automata networks have been shown to have a number of advantages for modeling complex systems, in comparison with Markov chains [4,8]. We believe that this is the case also for statistical testing based on usage models, without loss of generality or information; indeed, any Markov chain can be mapped into a SAN [3,9]. Usage models described with SAN have some interesting characteristics [1]:

- environment requirements (a critical issue for testing) can be made explicit in the model;
- the representation is modular, improving maintainability and readability;
- an individual use is a sequence of global states in the SAN – thus, its description is more detailed, which allows for easier mapping of uses into test cases;

- as we shall see in the section 4, for complex applications the computational cost of SAN-based test case generation is smaller than Markov-based.

Figure 1: Login System

For example, let us consider an (quite simple) application consisting of just two dialogues: the first is a login dialog (fig. 1a), where the user is prompted for a username and password; if the username or password is incorrect, the application issues an error message (fig. 1b). The second is a menu (fig. 1c), where the user can only terminate the application. This application can be described by the SAN illustrated in figure 2. The network has the following structure:

A_1 = { Start, Password, Menu }

A_2 = { Waiting, POK, PnotOk }

E = { **ST, QT, S, g, f** }

ST = { (Start, Waiting) → (Password, Waiting) }

QT = { (Password, Waiting) → (Start, Waiting),

(Menu, Waiting) → (Start, Waiting),

(Menu, POK) → (Start, Waiting) }

S = { (Password, Waiting) → (Menu, POK) }

g = { (Password, Waiting) → (Password, PNotOk) }

f = { (Password, PNotOk) → (Password, Waiting) }

I = { (Start, Waiting) }

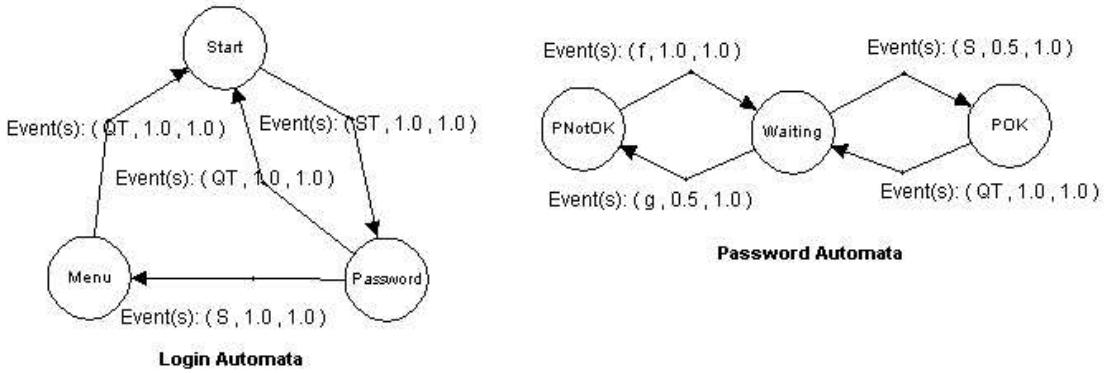

Figure 2: SAN Model

ST, **QT** and **S** are synchronizing events, while **g** and **f** are local. Note that, for simplicity, we describe the events as partial functions; if a global state does not have an image defined for some event, it means the event is not enabled at that state (for example, **ST** is enabled only at global state (Start,Waiting), which is the initial state). In figure 2, each event is represented by a triple <event identifier, event probability, transition probability>. For example, the probability of **ST** is 1.0, because it is the only possible event at the state (Start, Waiting), and its transition probability is also 1.0, since there is only one transition assigned to **ST** at that state.

2.4 Interface State-Event Model

STAGE-Test creates the test cases from the SAN usage model. These test cases form the input for STAGE-Script, which creates the scripts for automatic execution of those tests. Since the SAN is an abstract model describing usage of the application to be tested, it does not represent design and/or implementation information, which is needed for script generation. Therefore, we have an **Interface State-Event Model** (**ISEM**) associated with the SAN model. The ISEM contains the information necessary for test script generation.

Let S be a set of local states and A the set of automata. Let (G,E,Pe,Pt) be a SAN as described in section 3. An ISEM consists of a structure

(Ic,SI, EI, Fs, Fe)

Ic is a set of **interface components** (frames, buttons etc.).

SI is a set of **interface states.** Such states represent possible combinations of properties values (values of input fields, window visibility etc.) of the interface components when the system is in that state. Each combination is an interface state.

EI is a set of **interface events**, such that for each possible event in E there is a subset of interface events in EI. The interface events represent which user actions (button clicks, choices etc.) trigger the corresponding event in the model.

Fs: S $\rightarrow$ ***P***(SI) is a mapping from states to sets of interface states, such that for each local state in S there is an associated (possibly empty) subset of states in SI. It is possible having local states with no interface state, representing internal logic of the application. There is an important requirement, though: in order to generate the test script, each reachable global state must have at least one associated interface state.

Fe: E $\rightarrow$ ***P***(EI) is a mapping from events to sets of interface events. For each event in the model there must be at least one related interface event; otherwise, it is not possible to create a script for test cases with that event.

For example, in the SAN model of figure 2, the *Password* state in the **Login** automata corresponds to the pop-up window (a) in figure 1. We have two situations of interest: the fields “user” and “password” are filled with valid values and with invalid values. Therefore, we can define two interface states in the ISEM model of the application, both associated with the **Login**.*Password* state in the SAN model.

3. STAGE

STAGE (STAte-based test GEnerator) is an integrated environment for computer-supported generation of test cases and test scripts using state-based techniques. Our goal

in building this environment was to provide a framework where we can easily apply different testing techniques unified by the approach of state-based modeling. The process of designing tests for an application using STAGE consists of three steps: (a) building a model of the application to be tested; (b) generating test cases from the model; (c) generating test scripts from the test cases. The architecture of STAGE is organized in three packages, corresponding to the steps above. We call these packages STAGE-Model, STAGE-Test and STAGE-Script, respectively (figure 3). STAGE was developed in the Java language.

STAGE-Model is a toolkit for creating and editing state-based models of the application under test, using one of four types of state-based models - finite state machines, finite state machines with variables, Markov chains or SANs. We focus here on the SAN models. The core tool in the toolkit is SCE (State Chart Editor), a graphic editor where the user (normally the test engineer) defines the model structure – states, transitions, events and rates, depending on the model type. SCE can also import the states and transitions from a spreadsheet file. This is a useful feature when you want to test a new version of an application previously documented outside SCE. For example, when developing a Web site, many teams describe the pages and links in a spreadsheet. In that case, SCE reads the sheet and creates a first drawing of the model.

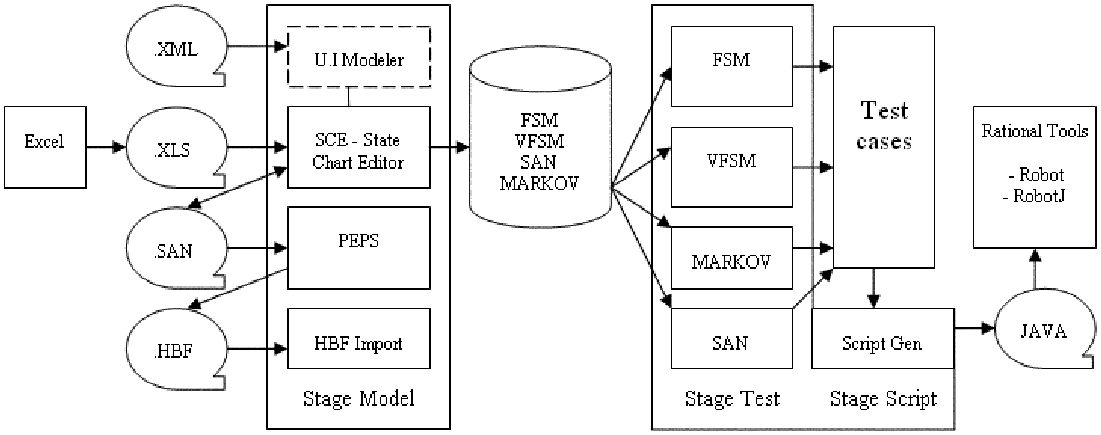

Figure 3: Architecture of STAGE

When the model is a SAN, one can perform some preliminary static analysis of the model before generating the tests, using the PEPS tool [4]. SCE exports the model into a file with the **.san** extension, containing the model description in the PEPS input format. One of the features of PEPS is to compute the equivalent Markov chain for a given SAN, and output it into a **.hbf** file. STAGE-Model may import this equivalent chain into the system using the HBF-Import tool.

The U.I. Modeler (dotted box on figure 3) is a user interface modeling tool that generates an XML description of the user interface and its relationships with the states and events in the model. This information is necessary for the script generation process in STAGE-Script. The format of the description is an implementation of the ISEM model described in the section 3. The UI Modeler is currently under implementation. At this moment the XML format is already specified but must be created using a text editor.

STAGE-Test is a toolkit for generating test cases from stated-based models built using SCE. The core tool in this case is TCG (Test Case Generator), an interactive tool where the user can create test suites. To create a test suite, the user must select a model and one of the test case generation techniques. The generated tests can be stored with the test suite on the database or exported into a text file.

For each type of model there are test case generation techniques that can be applied. For FSM or VFSM models (VFSM are properly converted to FSM before test case generation) there are two techniques available: a simple state coverage technique and the Wp algorithm [9]. For SAN and Markov models there are specific random test case generation algorithms. Each technique has some user-defined parameters. For all of them it is possible to define the maximum test case length; for Markov and SAN models, it is necessary to inform the number of test cases to be generated (the sample size).

In terms of test case generation for Markov chain models, the generation tool walks thru the model states, starting on the initial state. At each state, transitions are selected according to their probability distribution. The test case ends when the terminate state of the chain is reached or in the case that the maximum test case size is reached.

The generation of test cases based on SAN usage models works quite different of the Markov chain process. According to the models developed, each test case should start on the **ST** event and end at the **QT** event. So, the generation tool analyzes the current global state of the SAN, enumerating the candidate events to be fired, according to the current local state of each automaton, and an event is selected according to their rate distribution.

STAGE-Script is the script generation tool. It is integrated in the TCG interface and allows the user to generate automatic test scripts from the generated test cases. The current version generates scripts for the Rational playback testing tools (Robot and RobotJ) [7]. Users can generate 3 types of scripts: navigation, duration and performance. The first one is just the translation of the generated test cases to the script language. In the duration script, the user defines a time interval of test execution. The script will apply all the generated tests and, if there is time remaining, it will select tests at random to be applied until time is over. The performance script is specific for testing web servers and generates series of web page requests, simulating multiple users requests over a predefined time span. It is important to note that the script generation technology adopted applies only to systems with GUIs; that is a limitation of the current version. We are studying other execution engines and languages, in order to generate scripts for different types of applications.

3.1 Examples and experiments

Let us see an example of test script generation using a SAN model. We will use the login application presented in section 3. Figure 4 shows the main window of SCE, with the SAN model of the login application. SCE opens one window for each of the automata in the model. Note that using SCE we edit just the high-level automata. The relationships between the interface states and the SAN states are declared on a XML file as mentioned above. On figure 5 we can see a sample of the corresponding XML file.

With STAGE-Test, we create the test suite. Depending on the model type, a different set of test case generation techniques is available. Each technique has its own parameters.

Regarding SAN models, the test case generation technique is the usage-model statistical testing algorithm. In this case, for each test suite we must define the number of test cases to be generated and the maximum test case size (number of steps). Figure 6 shows the main interface of TCG with a sample of test cases. Note that each test case step is composed by a SAN global state followed by the event that triggers the state change.

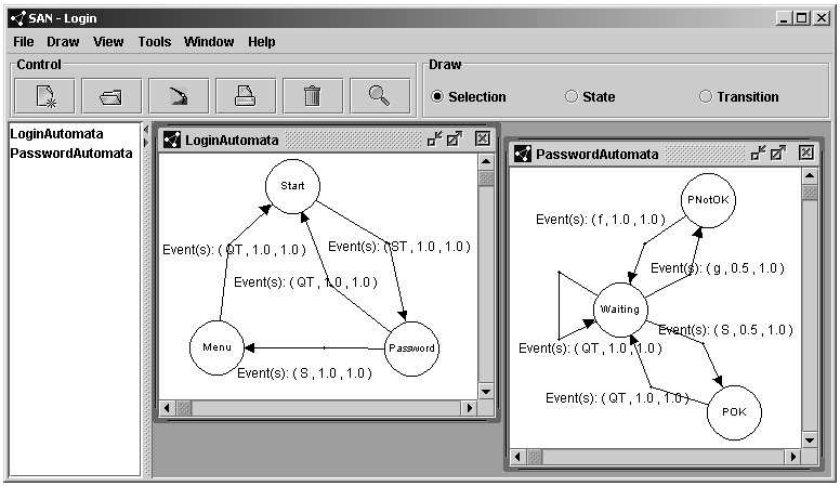

Figure 4: SCE interface

```
<isem>
    <interface_structure>
        <structure>
            <component name="MenuDialog">
                <object id="1" objectClass="JButton" text="EXIT" />
             </component>
         </structure>
        .
        .
    </interface_structure>
    <interface_states>
    <state id = "v1" modelState="Login Automata.Passoword" property="PasswordDialog">
        <components>
            <object id="Text1" value="lucas"/>
            <object id="Text2" value="123456"/>
        </components>
    </state>
    .
    .
    </interface_states>
    <interface_events>
        <event map="S">
            <transition>
                <source> valid 1 </source>
                <eventAction id="OK" action="Click" />
                <target> MenuDialog </target>
            </transition>
        </event>
    </interface_events>
</isem>
```

Figure 5: XML file ISEM model

The test script is also generated using TCG. In the current version, the generated scripts can be for Rational RobotJ or Rational Robot. Figure 7 shows a snippet of the script for testing the login application, based from the test suite. Note that it is necessary to combine the information in the test cases with the information declared on the XML file in order to generate a test script.

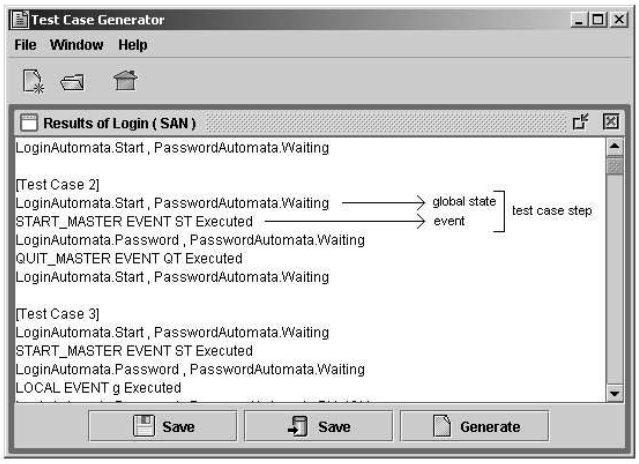

Figure 6: TCG interface

```
...
logInfo("Executing Test Number 3");
  try{
    startApp("Main");
    passwordtextText().setProperty("text", "123456");
    textText().setProperty("text", "lucas");
    OKButton().click();
    EXITButton().click();
  }catch(Exception e){
    logTestResult("Error generate = "+e.toString(), false);
  }
logInfo("Executing Test Number 4");
  try{
    ...
  }catch(Exception e){
    logTestResult("Error generate = "+e.toString(), false);
  }
...
```

Figure 7: Script for Login System

We developed a set of experiments with the environment, considering the following applications:

- the login application described above (2 automata, 9 global states),
- a Web calendar tool (5 automata, 420 global states),
- a forms-based documents editor (6 automata, 648 global states), and
- a Web-based bug tracking tool (8 automata, 9000 global states).

For each application, we developed a SAN model and generated the equivalent Markov chain, using the PEPS tool. For these models, we generated test suites from 100 to 5000 test cases, varying the maximum test case size from 10 to 40 steps. For eample, Figure 8 shows the execution times for the Calendar tool, considering 40 steps of maximum test case size both for the Markov Chain and SAN models.

The first information observed from these results is the small time necessary to the test case generation. Even the generation of 5000 test cases with 40 steps as limit stays under 8 seconds. It is possible to observe an almost linear behavior of the generation times for the Markov models. This phenomenon is probably due to the simplicity of the

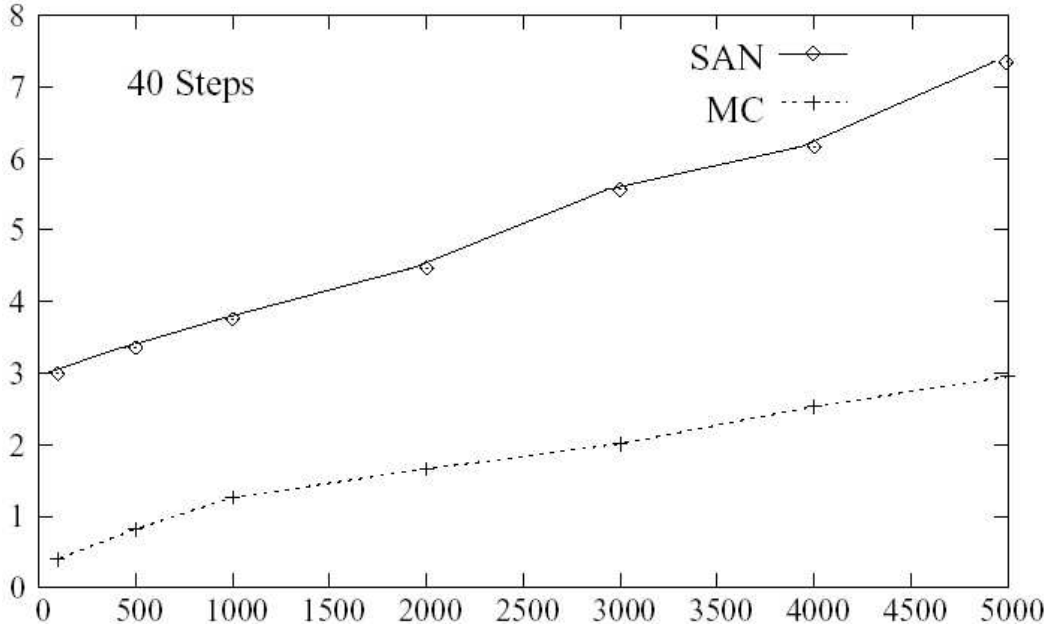

Figure 8: execution times for Calendar tool

Markov models, which describe all possible states in one single automaton. The algorithm for test case genration from the SAN models has a higher complexity, because of the synchronizing events. We observed that the number of synchronizing events in the model has an impact on the test case generation times - for example, the Calendar SAN model has a large number of synchronizing events (13 events).

4. Related Work

UMMs (Unified Markov Models) [6] are an extension of traditional Markov based models. UMM models capture the same kind of information that a SAN or a Markov chain does, but this information is represented as a set of hierarchical Markov chains. A top-level Markov chain represents the high-level states and transitions. At any level, more detail can be associated with an individual state using sub-models, creating a hierarchical model. Notice that in some of these Markov chains the sum of probabilities for transitions out from a given state may be less than one because the external destinations are omitted to keep the model simple. The implicit understanding is that the missing probabilities go to external destination (a model in a different level).

SANs are not hierarchical; instead, they allow for many different types of relatioships among system components, while keeping the description modular. From a computational viewpoint, SANs also have higher scalability: for example, it is possible to model environment aspects as, for example, number of servers and clients, with little impact in the computational effort to compute model properties. Such flexibility is useful for both statistical and load testing. Most current products in the testing industry (such as Rational TeamTest or Mercury TestRunner) do not generate test suites based on a usage model. Implemented systems use Markov chains, which impose severe limitations on the complexity of the models.

5. Conclusions and Future Work

Statistical testing is a technique with its own history of success. Nevertheless, the size of state-space and the need for automated execution have imposed restrictions on the application of statistical testing to more complex systems. STAGE is a framework that supports state-based modeling and script generation for complex systems in a modular

way. The productivity of the script development and script maintenance tasks also increased significantly; when there is a change in the interface, one only needs to edit the state-based model and re-generate the test cases and scripts. The key feature in the script generation is the mapping from the abstract model to the interface model.

A limitation of the current version of STAGE is that the ISEM model must be generated manually, using a text editor; we are already building the UI Modeler to help this task, which can be very time-consuming for complex interfaces. More than that, some parts of the ISEM model (components description) could be automatically extracted from the interface code (Java or HTML), leaving to the test engineer the task of defining states and assigning links to the high-level model (FSM, VFSM, Markov or SAN). We are currently investigating this possibility. We are also planning to incorporate state-based code analysis for object-oriented software components. This will allow, in the future, to integrate structural statistical testing into STAGE framework.

References

[1] Farina, A.G.; Fernandes, P H L; Oliveira, F.M.. "Representing Usage Models With Stochastic Automats Networks". In: *14th International Conference On Software Engineering And Knowledge Engineering* - SEKE02, 2002, Ischia, Italy.

[2] Fernandes, P. *Méthodes numériques pour la solution de systèmes Markoviens à grand espace d' états*. INPG, Grenoble, 1998. (PhD Thesis)

[3] Fernandes, P., Plateau, B. and Stewart, W.J. Efficient descriptor-vector multiplication in stochastic automata networks. *Journal of the ACM*, volume 45, no. 3, 1998, pp. 281-414.

[4] J.M.Fourneau, B.Plateau. PEPS: a package for solving complex Markov models of parallel systems. In: Proceedings of the Fourth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. Palma, Spain, 1988.

[5] Fujiwara, S.; Bochmann G. V.; Khendek F.; Amalou M. and Ghedemsi A. Test Selection Based on Finite State Model. IEEE Transactions On Software Engineering, vol 17(6):591-603, 1991.

[6] Kallepalli, C.; Tian J. - Measuring and Modeling Usage and Reliability for Statistical Web Testing. IEEE Transactions on Sotware Engineering, vol 27(11):1023-1036, 2001.

[7] Rational products. Description available at URL http://www.rational.com/products/. Accessed 9/19/2003.

[8] Sayre, K. Improved Techniques for Software Testing Based on Markov Chain Usage Models. PhD thesis. University of Tennessee, Knoxville, December 1999.

[9] Shehady, R.K.; Siewiorek, D.P.; A method to automate user interface testing using variable finite state machines Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, 24-27 June 1997. Page(s): 80 -88

[10] Trammel, C. Quantifying the Reliability of Software: Statistical Testing Based on a Usage Model. *Proceedings of the Second IEEE International Symposium on Software Engineering Standards*, Canada, August 1995

Uma Rede Overlay Tolerante a Intrusões

Rafael R. Obelheiro* **e Joni da Silva Fraga†**

Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina
Caixa Postal 476 – 88040-900 – Florianópolis – SC – Brasil
Email: rro@das.ufsc.br, fraga@das.ufsc.br

Resumo. *Este artigo apresenta ROTI, uma rede overlay tolerante a intrusões. A ROTI provê segurança para as mensagens e usa protocolos de encaminhamento de pacotes e roteamento tolerantes a faltas bizantinas. Quando da detecção de falhas, a rede overlay pode se reconfigurar, excluindo links faltosos e adicionando nós para preservar a segurança e a disponibilidade da rede.*

Abstract. *This paper introduces ROTI, an intrusion tolerant overlay network. ROTI provides message security and uses packet forwarding and routing protocols which tolerate Byzantine faults. When faults are detected, the overlay network can easily reconfigure itself, excluding faulty links while adding nodes in order to preserve network security and availability.*

1. Introdução

A crescente dependência da sociedade moderna em relação aos seus sistemas de informação impõe a esses sistemas requisitos cada vez mais severos de tolerância a faltas e segurança. Essa situação é exacerbada em sistemas ligados à Internet, que vem se tornando a cada dia que passa um ambiente mais hostil.

Os mecanismos de segurança tradicionais se concentram na prevenção de intrusões, enquanto os mecanismos de tolerância a faltas nem sempre são adequados para lidar com faltas maliciosas. Na confluência dessas duas correntes de pesquisa encontra-se a tolerância a intrusões, que se preocupa com incidentes de segurança sem depender da inviolabilidade dos componentes de segurança; ao contrário, admite-se que alguns desses componentes podem sofrer intrusões, e constrói-se o sistema de modo que ele desempenhe corretamente suas funções mesmo que essas intrusões se concretizem [9, 5, 17].

Uma característica importante em sistemas tolerantes a faltas ou a intrusões é a resiliência, que é capacidade do sistema de se ajustar a eventos adversos (como falhas de comunicação e intrusões) sem comprometer os seus requisitos de confiabilidade e segurança.

As redes *overlay*, que são redes lógicas construídas sobre redes físicas existentes, vêm sendo usadas para implementar serviços de rede com características diferenciadas, especialmente em relação ao que oferece a camada IP da Internet. Elas são interessantes

*Bolsista CAPES.
†Bolsista de Produtividade em Pesquisa do CNPq – Nível 2.

porque possibilitam que novos protocolos sejam implementados e testados em uma rede real, sem que seja necessário nenhum suporte especial por parte da infra-estrutura de rede já existente. Além disso, a flexibilidade inerente às redes *overlay* as torna ideais para a construção de serviços de rede resilientes. A literatura registra algumas propostas de redes *overlay* que atendem a requisitos de tolerância a faltas e de segurança [3, 2, 14]. Entretanto, não se conhece nenhuma proposta de rede *overlay* que seja tolerante a intrusões.

Este artigo propõe uma arquitetura de rede *overlay* tolerante a intrusões, que envolve propriedades de tolerância a faltas e de segurança. A propriedade de tolerância a faltas é mantida se entre dois nós corretos *A* e *B* do *overlay* houver pelo menos um caminho livre de faltas no *overlay* ligando os dois nós. Neste trabalho, a premissa em termos de semântica de falhas é o comportamento bizantino. As propriedades de segurança são mantidas se: (i) o conteúdo dos pacotes trocados entre *A* e *B* não puder ser examinado pelos nós intermediários (do *overlay* ou da rede física); (ii) o nó de destino *B* possa verificar que os pacotes recebidos são íntegros (não foram corrompidos em trânsito) e (iii) autênticos (foram efetivamente originados por *A*).

O restante do artigo está organizado da seguinte forma: a seção 2 apresenta os conceitos de intrusão e tolerância a intrusões, e a seção 3 apresenta conceitos de redes *overlay*. A rede *overlay* tolerante a intrusões proposta é apresentada na seção 4. A seção 5 discute trabalhos relacionados, e a seção 6 traz as conclusões e perspectivas futuras.

2. Tolerância a Intrusões

A pesquisa em segurança computacional tradicionalmente tem se concentrado em prevenir a ocorrência de violações de segurança. Devido a uma série de razões (como a aparentemente inevitável presença de vulnerabilidades), esta abordagem não tem sido muito bem sucedida. Em contrapartida, é possível supor como a abordagem de tolerância tipicamente aplicada a faltas acidentais poderia ser aplicada à segurança [17]:

- presumir que os sistemas permaneçam, até certo ponto, vulneráveis, independente das proteções que possuam;
- presumir que ataques contra componentes ou subsistemas podem acontecer, e que alguns desse ataques serão bem sucedidos;
- garantir que, a despeito destas hipóteses, o sistema como um todo permaneça seguro e capaz de desempenhar suas funções.

Uma intrusão pode então ser considerada como um evento que ocorre em um sistema, provocado por um indivíduo (ou por um processo automatizado iniciado por ele) e que resulta em, ou oportuniza, violação de uma ou mais das propriedades fundamentais da segurança (confidencialidade, integridade e disponibilidade). O conceito de intrusão pode ser, portanto, considerado um sinônimo para termos mais comumente usados em segurança computacional, tais como invasão e comprometimento de um sistema; o abuso de privilégios por parte de usuários autorizados também é uma intrusão. A tolerância a intrusões seria, por sua vez, a capacidade de um sistema de continuar a fornecer um serviço seguro a despeito de intrusões em um determinado número de seus componentes. Este conceito admite uma degradação na funcionalidade oferecida pelo sistema, desde que a sua segurança não seja violada.

3. Redes Overlay

Uma rede *overlay* é uma rede lógica construída sobre uma rede física existente [3, 1]. Redes *overlay* são interessantes porque permitem oferecer funcionalidades e qualidades de serviço diferenciadas sobre uma rede já estabelecida sem prejudicar a estabilidade e a robustez desta, causando um mínimo impacto sobre as aplicações existentes. Isto possibilita, por exemplo, que novos protocolos e idéias sejam testados com mais facilidade e possam amadurecer antes de serem transpostos para a rede real.

A figura 1 mostra um exemplo de como uma rede *overlay* se sobrepõe a uma rede física. Cada nó da rede *overlay* é também um nó da rede física. Uma conexão entre dois nós da rede *overlay* é chamada de *link* virtual, e corresponde à rota entre os respectivos nós na rede física. Cada nó é responsável por processar e rotear pacotes segundo critérios específicos da rede *overlay*; tais critérios normalmente dependem da aplicação a que a rede se destina. Cada rede *overlay* utiliza um esquema de endereçamento próprio, que pode ou não ser baseado no esquema de endereçamento da rede física. As conexões entre os nós do *overlay* são implementadas na rede física usando alguma forma de tunelamento (isto é, os pacotes da rede *overlay* são encapsulados em pacotes da rede subjacente), e não necessitam seguir nenhuma topologia predeterminada.

Assim como em uma camada de rede típica, as funções principais de uma rede *overlay* são o encaminhamento (*forwarding*) de pacotes, que determina como os elementos da rede (os roteadores) processam um pacote em trânsito para que ele chegue ao seu destino, e o roteamento, que é o processo através do qual o conhecimento sobre as diferentes rotas entre nós da rede é calculado, armazenado e disseminado.

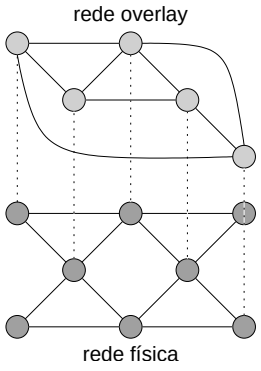

Figura 1: Rede *overlay* sobreposta a uma rede física

4. ROTI: Uma Rede Overlay Tolerante a Intrusões

Esta seção apresenta a ROTI, uma proposta de rede *overlay* tolerante a intrusões. Primeiramente coloca-se o modelo considerado e as premissas adotadas. Na seqüência, são discutidos aspectos de encaminhamento de pacotes, roteamento, detecção de falhas e reconfiguração da topologia da rede. A seção conclui com algumas considerações sobre a implementação da proposta.

4.1. Modelo e Premissas

O modelo da ROTI é mostrado na figura 2. Neste modelo, cada nó da rede *overlay* é um *host* Internet, sendo que cada *host* abriga apenas um nó *overlay*. Os nós do *overlay*

dividem-se em nós ativos e nós de reserva, inativos. A idéia é ter um *pool* de *hosts* Internet aptos a se tornarem nós da rede *overlay*. Em um dado momento, apenas um subconjunto destes *hosts* encontram-se ativos, formando a topologia corrente da rede; os demais *hosts* são considerados nós de reserva.

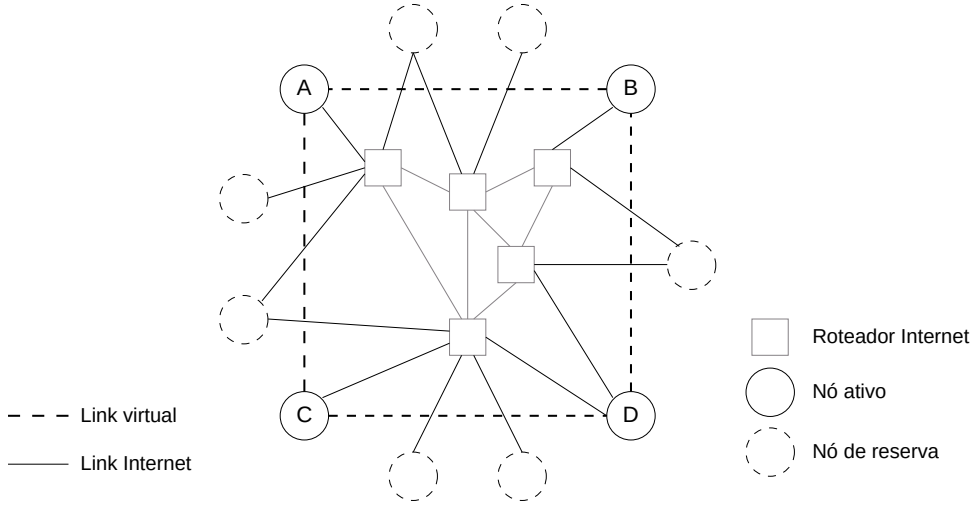

Figura 2: Modelo de rede *overlay* tolerante a intrusões

Os *links* virtuais são estabelecidos durante a ativação dos nós (a ativação de nós é discutida na seção 4.5). Em princípio, sempre que houver um *link* virtual ativo entre dois nós, a comunicação entre estes nós é feita diretamente através deste *link* virtual. A indireção, isto é, o envio de pacotes através de um ou mais nós intermediários, ocorre no caso de não existir um *link* virtual entre os nós de origem e destino. Por exemplo, na figura 2, o nó *A* tem que enviar mensagens para o nó *D* através dos nós *B* ou *C*, uma vez que não existe *link* virtual entre *A* e *D*.

Em uma dada comunicação entre nós do *overlay*, apenas os nós de origem e destino são considerados de confiança. Os demais nós, sejam eles do *overlay* ou da rede física subjacente, podem exibir comportamento bizantino, agindo sozinhos ou formando conluios. Um nó faltoso pode interceptar, modificar ou injetar pacotes, criar laços na rede, descartar pacotes de forma seletiva, rotear pacotes através de caminhos sub-ótimos ou fazer com que um caminho pareça mais curto ou mais longo do que realmente é.

Os mecanismos de detecção de falhas usados na ROTI (seção 4.4) identificam *links* virtuais com comportamento bizantino. Um *link* virtual pode exibir comportamento bizantino se pelo menos uma das três condições abaixo for satisfeita:

i. pelo menos um dos nós *overlay* que formam o *link* for faltoso;
ii. pelo menos um dos *links* da rede física usados pelo *link* virtual for faltoso;
iii. pelo menos um dos nós da rede física por onde passa o *link* virtual for faltoso.

Para aumentar a probabilidade de um caminho livre de falhas ser encontrado, é desejável que haja caminhos disjuntos entre cada par de nós da rede *overlay*, ou seja, caminhos que não compartilhem nós ou *links* tanto do *overlay* como da rede física (com exceção dos nós de origem e destino). A inexistência de caminhos disjuntos entre um determinado par de nós *overlay A* e *B* significa que existe pelo menos um nó ou *link* que, se for faltoso, pode interromper completamente a comunicação entre *A* e *B*. Na prática,

uma boa maneira de obter caminhos disjuntos é ter cada nó do *overlay* localizado em redes *multihomed* (conectadas a mais de um provedor de *backbone*). O uso de *multihoming* é recomendado já há algum tempo para combater ataques de negação de serviço (DoS—*Denial of Service*) [12]. Embora os protocolos de roteamento e encaminhamento partam da premissa de que a rede física permanece conectada, caso a rede sofra uma partição a comunicação dentro dos seus componentes conectados não é afetada negativamente, embora a capacidade de adaptação da rede (seção 4.5) possa ser comprometida.

4.1.1. Mecanismos Criptográficos

Uma infra-estrutura de chaves públicas (PKI—*Public Key Infrastructure*) é utilizada para dar suporte aos mecanismos criptográficos usados nos protocolos de roteamento e encaminhamento de pacotes. A PKI considerada segue um modelo de confiança descentralizado, baseado no SPKI (*Simple Public Key Infrastructure*) [7]. Este tipo de modelo de confiança elimina o ponto único de falhas representado pela autoridade certificadora (AC) presente nas PKIs tradicionais, com modelo de confiança centralizado.

Para poder ser admitido na rede *overlay*, um nó deve apresentar um certificado de autorização SPKI que lhe confira o direito de acesso à rede e que seja assinado por mais de um emissor através de um esquema de *threshold subjects* [8].[1] A idéia é que o comprometimento de um único emissor não possibilite a inclusão de nós forjados na rede nem a exclusão (pela autenticação incorreta) de nós válidos.

Sempre que possível, os mecanismos criptográficos adotados baseiam-se em chaves secretas compartilhadas entre cada par de nós da rede. Essa preferência deve-se a razões de desempenho. As chaves secretas são estabelecidas sob demanda através de um protocolo Diffie-Hellman [6], que utiliza as chaves públicas disponíveis através da PKI para estabelecer as chaves secretas de forma segura.

4.2. Encaminhamento de Pacotes

O encaminhamento de pacotes determina como os roteadores processam pacotes de forma a transmiti-los da origem para o destino. O caso mais simples de encaminhamento é quando existe um *link* conectando os nós de origem e destino. A dificuldade surge quando não existe uma conexão direta entre os nós comunicantes. A solução mais primitiva para o problema é usar *flooding*: o nó de origem encaminha o pacote para todos os seus vizinhos, e cada roteador encaminha o pacote para todos os seus vizinhos com exceção daquele de onde veio o pacote. Isso garante que o pacote chegue a todos os nós da rede—o que trivialmente inclui o destinatário—mas é bastante custoso, pois o número de duplicatas de um pacote varia exponencialmente com o tamanho da rede.

A solução geralmente adotada, porém, é encaminhar o pacote por um conjunto de nós intermediários para chegar ao nó de destino, o que é bem menos custoso do que disseminar pacotes por toda a rede. Este caminho através de nós intermediários é chamado de rota, e é determinado por protocolos de roteamento, que são discutidos na seção 4.3. Por enquanto, considera-se que um nó *A* que deseja se comunicar com um nó *B* pode determinar uma rota apropriada na rede *overlay* para transmitir os seus pacotes.

[1] *Threshold subjects* determinam que *k* dentre *n* principais ("sujeitos") assinem um certificado de autorização para que ele seja válido.

Na ROTI, o encaminhamento de pacotes envolve autenticação, confidencialidade e confiabilidade. A autenticação garante a origem dos dados que estão sendo transmitidos, enquanto a confidencialidade garante que nenhum nó intermediário possa inspecionar o conteúdo dos pacotes transmitidos. A confiabilidade, por sua vez, garante que os pacotes sejam entregues mesmo na presença de faltas e intrusões na rede, desde que exista um caminho livre de faltas entre os nós comunicantes. O mecanismo de encaminhamento de pacotes utilizado considera tanto faltas acidentais quanto maliciosas.

A autenticação de origem (que garante integridade) é implementada através de assinaturas digitais no nó de origem, que são verificadas em cada roteador no caminho. Um nó correto encaminha um pacote apenas se a verificação da sua assinatura for bem sucedida; caso essa verificação falhe, o nó descarta o pacote, e registra uma falha para o *link* através do qual este pacote chegou. A confidencialidade, por sua vez, é obtida através da cifragem do conteúdo dos pacotes com um algoritmo criptográfico simétrico.

A confiabilidade na comunicação é obtida através da detecção e recuperação de falhas. A detecção de falhas é baseada em reconhecimentos positivos (ACKs). A técnica de ACKs funciona bem quando a rota usada apresenta apenas perda esporádica de pacotes. Porém, quando uma rota é interrompida ou passa a apresentar uma incidência muito grande de perdas esta não pode ser mais usada. Nessa situação, diferentes estratégias de recuperação de falhas podem ser adotadas:

(a) Mudança de rota: escolhe-se uma rota alternativa para o destino. Como o protocolo de roteamento oferece um mecanismo de localização de faltas, a nova rota deve evitar os *links* reportados como faltosos.
(b) *Flooding*: caso a rota não funcione, o pacote é transmitido através de *flooding*.

A ROTI utiliza a primeira estratégia, que é menos confiável porém bem menos custosa do que a segunda.

O encaminhamento de pacotes na ROTI ocorre da seguinte maneira. Se existe um *link* virtual ativo entre os nós de origem e destino, os pacotes são transmitidos através deste *link*, e o seu recebimento é confirmado através de ACKs. Quando não existe *link* virtual entre os nós, ou quando o *link* é declarado faltoso, tenta-se obter uma rota no *overlay* entre os nós de origem e destino através do protocolo de roteamento. Se existe uma rota apropriada, os pacotes passam a ser encaminhados por esta nova rota. Neste caso, utiliza-se *source routing*, com o nó de origem especificando no cabeçalho do pacote a rota completa a ser usada. Os nós intermediários precisam então verificar a assinatura do pacote (descartando pacotes com assinatura inválida) e enviá-lo para o próximo nó da rota. Caso a rota em uso contenha um *link* que venha a ser declarado faltoso, solicita-se ao protocolo de roteamento uma nova rota. Se o protocolo de roteamento não encontrar uma rota apropriada, o nó de origem considera o nó de destino inalcançável, e sinaliza essa condição para a camada superior.

4.3. Roteamento

Sempre que um pacote não pode ser transmitido diretamente através de um *link* virtual, ele é encaminhado ao destino através de nós intermediários. Para que isso seja possível, é necessário determinar uma rota que parta do nó de origem, passe por um ou mais nós intermediários e chegue ao nó de destino. Um protocolo de roteamento é o

responsável por armazenar e disseminar informações sobre as rotas entre os nós da rede de forma a permitir a escolha da melhor rota entre uma origem e um destino específicos.

Protocolos de roteamento podem ser proativos e reativos (sob demanda). Protocolos proativos são aqueles onde os roteadores trocam informações de roteamento periodicamente, independente da utilização das rotas. Neste caso, quando surge a necessidade de encaminhar um pacote o roteador já conhece a rota para o nó de destino, e pode transmitir o pacote imediatamente. Em contrapartida, este tipo de protocolo gera um *overhead* constante em função da troca de informações de roteamento, que independe das rotas estarem sendo usadas ou não. Os protocolos reativos, por outro lado, iniciam um processo de descoberta de rota apenas quando dados necessitam ser enviados. A rota descoberta fica então armazenada em *cache* até ser descartada por falta de uso (após um determinado período) ou por mudanças na topologia da rede. Este tipo de protocolo introduz uma latência adicional na transmissão quando é necessário descobrir uma nova rota.

A rede *overlay* tolerante a intrusões utiliza um protocolo de roteamento sob demanda. Este tipo de protocolo foi escolhido visando a um melhor desempenho da rede quando não existem falhas. Como pacotes só são encaminhados através de nós intermediários quando o *link* virtual entre os nós de origem e destino não está disponível, a latência adicional imposta pelo processo de descoberta de rotas é mais interessante do que o *overhead* constante de um protocolo proativo. Além disso, os protocolos de roteamento proativos possuem algumas características que os tornam menos indicados para o modelo de faltas considerado na ROTI [15].

A estratégia de roteamento adotada na ROTI é baseada em uma proposta para roteamento seguro em redes *ad hoc* sem fio [4], adaptada para uso em redes *overlay*. Essa estratégia implementa descoberta de rotas, detecção de falhas e gerenciamento de métricas de roteamento.

O processo de descoberta de rotas funciona da seguinte maneira: um nó *A* que deseja obter uma rota para um nó *B* monta uma requisição de rota assinada, que é disseminada na rede *overlay* através de *flooding*. A requisição é propagada até *B*, onde a sua autenticidade é confirmada e uma resposta é enviada para *A*, também através de *flooding*. À medida em que a resposta vai se propagando através da rede, os nós intermediários vão calculando a rota entre *A* e *B*. Para determinar a melhor rota, cada nó intermediário soma o custo da rota até o nó anterior (presente no pacote) com o custo do *link* entre o nó anterior e si próprio; o novo custo é armazenado no próprio pacote, que é assinado pelo nó antes de ser enviado para o próximo roteador. Todos os nós intermediários verificam a validade das assinaturas tanto de requisições como de respostas com o intuito de evitar que pacotes inválidos sejam disseminados através da rede. O *flooding* é usado neste caso para garantir a máxima robustez, evitando que requisições ou respostas se percam ao passarem por um *link* faltoso.

Diferente da proposta original [4], na ROTI o nó de origem armazena em um *cache* não apenas a melhor rota para o destino mas todas as rotas disjuntas (isto é, que não compartilham nenhum nó intermediário ou *link* virtual) obtidas. Isso permite que, ao ser requisitada uma nova rota em função de faltas na rota atual, o subsistema de roteamento possa fornecer imediatamente uma rota alternativa disjunta enquanto o protocolo de descoberta de rotas é executado novamente, eliminando a latência inicial normalmente

imposta pelo processo de descoberta.

Outro ponto importante é que o nó de origem descarta as rotas que incluam *links* faltosos, conforme será detalhado na seção 4.5. Essa estratégia de evitar *links* faltosos (em oposição a técnicas de detecção e exclusão de nós faltosos) permite que a rede tolere faltas bizantinas mesmo que exista uma maioria de nós faltosos e sem que seja necessário recorrer a protocolos de acordo para decidir de forma distribuída quais nós são faltosos e devem ser excluídos da rede.

4.4. Detecção de Falhas

O principal mecanismo de detecção de falhas na ROTI são os ACKs, que detectam falhas na transmissão de pacotes. Quando um ACK não é recebido antes do seu *timeout* expirar, o nó de origem registra uma perda para a rota em uso e retransmite o pacote. Quando o número de pacotes perdidos ultrapassa um dado limiar, considera-se que há uma falha no caminho, e inicia-se um processo para localizar o *link* faltoso. Esse processo consiste em retransmitir o pacote pela mesma rota, especificando no cabeçalho que todos os nós intermediários devem enviar um ACK para o nó de origem confirmando o recebimento do pacote. O *link* situado entre o último ACK recebido e o primeiro ACK perdido é declarado faltoso. Por exemplo, na figura 3, se o nó A envia um pacote para o nó E e recebe ACKs de B e C (mas não de D ou E), ele considera o *link* (C,D) faltoso.

É importante notar que o uso de ACKs oferece uma localização imprecisa de faltas. Seja o exemplo da figura 3, onde um nó A envia um pacote para um nó E e não recebe o ACK de D antes de expirar o respectivo *timer*. Isso pode acontecer por diversas razões: o pacote é perdido (caso ilustrado na figura), o ACK é perdido, D não envia o ACK, etc. Estas razões podem ser resumidas em uma ou mais de três causas: C é faltoso, D é faltoso e/ou o *link* (C,D) é faltoso. A detecção de falhas, nesse caso, pode reportar como suspeitos tanto o *link* (C,D) como os nós C e D. É importante observar que ambos os nós têm que ser considerados suspeitos, uma vez que é impossível (usando apenas ACKs) determinar qual deles é efetivamente o faltoso caso apenas um deles o seja.

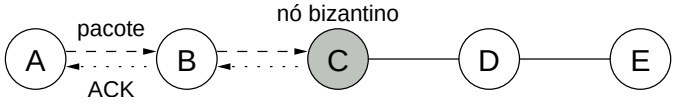

Figura 3: Detecção de falhas com ACKs

Em uma rede em que os nós e *links* podem apresentar comportamento bizantino, os ACKs podem ser manipulados pelos elementos faltosos. No exemplo da figura 3, o nó bizantino C pode deixar de transmitir ACKs do nó E em uma tentativa de incriminar o *link* correto (D,E). Para evitar que isso aconteça, a lista de nós que devem enviar ACKs para o origem e os próprios ACKs são cifrados com a chave secreta compartilhada entre cada nó do caminho e o nó de origem. Desta forma, um nó bizantino não pode verificar se um pacote que está sendo encaminhado contém um ACK ou não, o que o impede de interferir seletivamente na transmissão (o nó pode filtrar pacotes mesmo assim, mas isso acabaria por incriminar um de seus próprios *links*).

Os mecanismos criptográficos usados na ROTI também fornecem dados para a detecção de falhas. Voltando ao exemplo da figura 3, se D recebe um pacote com assinatura inválida de C ele registra uma falta para o *link* incidente (C,D), uma vez que pacotes

com assinatura inválida não devem ser propagados na rede. Certificados SPKI com assinaturas inválidas também representam uma falta do *link* por onde foram transmitidos. Faltas vinculadas aos mecanismos criptográficos não precisam ser localizadas, uma vez que só podem ser atribuídas ao *link* incidente ou ao nó vizinho.

4.5. Reconfiguração da Rede

Uma característica importante da ROTI é a sua capacidade de adaptação que consiste na reconfiguração dinâmica da topologia da rede. Existem dois aspectos envolvidos nessa reconfiguração: exclusão de *links* virtuais e ativação de nós de reserva.

Basicamente, um *link* virtual é excluído após apresentar uma determinada taxa de faltas λ_{max}. A cada falta registrada pelos mecanismos de detecção e localização de faltas, a taxa de faltas $\lambda_{I,J}$ do *link* (I,J) apontado como faltoso é analisada. Se $\lambda_{I,J} \leq \lambda_{max}$, o *link* (I,J) é posto em quarentena. Durante a quarentena, o *link* é ignorado, e qualquer rota que o contenha é descartada. Após um período t_q, ou após algum pacote válido chegar ao nó através de uma rota que passe pelo *link* em quarentena, este é reabilitado. A quarentena é usada para evitar que um *link* que esteja apenas temporariamente faltoso (devido a congestionamento na rede, por exemplo) seja excluído indevidamente por apresentar um grande número de faltas em um curto espaço de tempo.

Quando $\lambda_{I,J} > \lambda_{max}$, o *link* (I,J) é considerado permanentemente faltoso e deve ser excluído. Se (I,J) é um *link* local (ou seja, faz parte do conjunto de *links* virtuais do nó que deseja excluí-lo), ele deixa de ser usado para as comunicações deste nó: nenhum pacote é enviado através de (I,J), pacotes eventualmente recebidos através de (I,J) são descartados, e o *link* é ignorado no protocolo de roteamento. Se, por outro lado, (I,J) não for local, ele simplesmente passa a ser ignorado no protocolo de roteamento (rotas que passam por (I,J) são descartadas). Cabe ressaltar que a exclusão de um *link* virtual é uma decisão local: cada nó monitora os *links* da rede, e decide quando e quais excluir, de forma independente. Isso significa, por exemplo, que um nó correto *A* deve encaminhar pacotes originados por outros nós que passam por um *link* não local (I,J), mesmo que este *link* tenha sido excluído por *A*.

Quando um nó *A* detecta que um nó *B* ficou inacessível (isto é, todos os *links* virtuais para *B* foram removidos), ele tenta ativar um nó reserva *R* que seja topologicamente próximo a *B* (a intenção aqui é que os *hosts* ligados a *B*, ao perceber que este se tornou não funcional, migrem para um nó próximo). Para ativar o nó de reserva *R*, o nó *A* envia uma mensagem de ativação assinada `activate(R,B)` para *R*, e difunde (através de *flooding*) uma mensagem `newnode(R,B)` no *overlay*. Os nós do *overlay* que receberem `newnode(R,B)` e concordarem com a ativação de *R* também enviam `activate(R,B)` para o novo nó. Se *R* receber um mínimo de ϕ mensagens de ativação, ele tenta se tornar ativo no *overlay*. Para isso, ele tenta estabelecer *links* virtuais com ρ nós *overlay* ($\rho \leq \phi$) dentre os que lhe enviaram `activate(R,B)`. A ativação é validada com o estabelecimento do *link*, que só ocorre se *R* provar que detém pelo menos ϕ mensagens `activate(R,B)`; isso é provado incluindo as mensagens de ativação na requisição de estabelecimento de *link* virtual.

A lista de nós de reserva é instalada manualmente nos primeiros nós da rede. Essa lista é assinada pelas chaves emissoras de certificados SPKI. Quando um nó reserva é ativado, os nós que estabelecem *links* virtuais com ele enviam-lhe as suas cópias da

lista. Listas com assinatura inválida são descartadas, e são consideradas um indício de comportamento malicioso por parte de quem as envia.

4.6. Aspectos de Implementação

Para demonstrar a aplicabilidade da proposta e possibilitar a compreensão das suas implicações práticas, encontra-se em desenvolvimento um protótipo da ROTI. O *software* em cada nó do *overlay* tem a estrutura mostrada na figura 4.

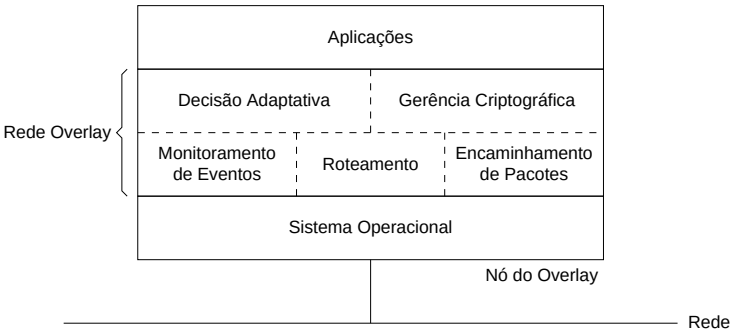

Figura 4: Arquitetura do protótipo

Os módulos de encaminhamento de pacotes e de roteamento implementam as funções apresentadas nas seções 4.2 e 4.3. Os pacotes da rede *overlay* são transmitidos na rede física como pacotes UDP, que oferecem uma comunicação não confiável com um *overhead* mínimo. Os pacotes recebidos da camada de superior (transporte) são cifrados através de um algoritmo simétrico, utilizando uma chave secreta compartilhada entre o nó de origem e o de destino.

O módulo de monitoramento de eventos é responsável pela detecção e localização de faltas, conforme descrito na seção 4.4. Este módulo baseia-se em informações de falhas fornecidas pelos demais módulos, como ACKs perdidos e assinaturas digitais inválidas.

O módulo de gerência criptográfica encarrega-se de recuperar, verificar e armazenar os certificados SPKI usados na rede *overlay*. Além disso, ele também é responsável por estabelecer e armazenar as chaves secretas compartilhadas com os outros nós da rede.

O módulo de decisão adaptativa recebe as informações do módulo de monitoramento de eventos, verifica se alguma ação de reconfiguração precisa ser tomada e determina qual seria essa ação. Em um nó de reserva, este módulo também é responsável por decidir quando o nó está apto a ser ativado e por dirigir o processo de ativação (seção 4.5).

5. Trabalhos Relacionados

Embora não haja registro na literatura de outras propostas de rede *overlay* tolerante a intrusões, diversas experiências possuem intersecção com este trabalho e podem ser usadas como referência para diferentes aspectos da arquitetura proposta.

Existem diversas experiências que utilizam redes *overlay* para oferecer serviços seguros ou tolerantes a faltas. Na RON [3] a rede *overlay* é completamente conectada, e cada nó monitora o estado dos seus *links* virtuais, redirecionando pacotes através de nós intermediários em caso de falha do *link*. A diferença crucial entre a RON e a ROTI

é que a primeira considera apenas faltas de *crash*, além de não oferecer mecanismos de segurança. O Spines [2] propõe uma rede *overlay* com características de segurança e confiabilidade. No entanto, os resultados do Spines até o momento enfatizam a comunicação confiável com alto desempenho usando ACKs *hop*-a-*hop* (em vez de fim-a-fim) [1]; os mecanismos de segurança e reconfiguração, embora mencionados em [2], não são definidos. Essa abordagem de comunicação confiável do Spines é considerada complementar à funcionalidade da ROTI, e pode ser utilizada em uma futura revisão da arquitetura.

O SecComm [11] é um serviço de comunicação ponto a ponto *survivable*, que utiliza diversidade de métodos para evitar que a quebra de um algoritmo ou chave comprometa as propriedades de mecanismos criptográficos. Isso é conseguido, por exemplo, cifrando dados múltiplas vezes (com algoritmos e/ou chaves de sessão diferentes). Assim como no caso do Spines, a abordagem do SecComm pode complementar os mecanismos criptográficos da ROTI.

A segurança no roteamento não é uma preocupação nova; Papadimitratos e Haas [15] trazem um *survey* envolvendo protocolos usados na Internet. O trabalho pioneiro sobre redes tolerantes a faltas bizantinas é a tese de Perlman [16], que combina *flooding* e roteamento *link state*; a influência desse trabalho pode ser constatada em muitas propostas atuais. O roteamento considerando faltas bizantinas adquire maior importância em redes *ad hoc* sem fio, onde a possibilidade dos nós estarem expostos a um ambiente hostil é maior do que em redes convencionais; propostas nesse contexto incluem OSDBR [4] e Ariadne [13].

A FVPN [10] é uma VPN (*Virtual Private Network*) tolerante a faltas com objetivos similares aos da ROTI. A primeira diferença entre a FVPN e a ROTI é que a primeira é projetada para uso dentro de um domínio de roteamento e utiliza informações de roteamento já disponíveis dentro desse domínio, enquanto que a ROTI não impõe qualquer restrição à topologia. Outro ponto é que a FVPN preconfigura rotas alternativas antes de utilizá-las, minimizando a latência na recuperação de falhas em troca de um possível desperdício de recursos de rede. Além disso, a FVPN usa apenas redundância na rede física, não utilizando indireção através da rede *overlay*; isso faz com que em determinadas situações a ROTI consiga entregar pacotes e a FVPN não. Entretanto, a principal diferença está na semântica de falhas: a FVPN age apenas em reação a falhas sinalizadas pelo protocolo de roteamento *link state*, o que não inclui comportamento bizantino dos elementos de rede.

6. Conclusões

Este artigo apresentou a ROTI, a primeira proposta de rede *overlay* tolerante a intrusões. Todos os mecanismos de tolerância a faltas e segurança usados na ROTI têm o objetivo de tornar a rede resiliente a faltas e intrusões. Neste contexto, destacam-se a natureza adaptativa da rede e a grande autonomia que cada nó tem para tomar suas próprias decisões.

Embora as premissas adotadas na ROTI garantam apenas de forma probabilista a resiliência da rede, é possível, através de premissas adicionais, fornecer garantias a respeito do funcionamento da rede. Por exemplo, se existirem $f+1$ caminhos disjuntos entre cada par de nós do *overlay* a rede é capaz de entregar pacotes entre dois nós corretos

mesmo que existam f faltas (na rede física ou no *overlay*). A opção por garantias probabilistas tem o objetivo justamente de evitar premissas excessivamente fortes que limitem a aplicabilidade da ROTI.

O próximo passo é, logicamente, a implementação de um protótipo da ROTI, trabalho que já se encontra em andamento. Esse protótipo permitirá a coleta de dados sobre o desempenho da arquitetura proposta e uma melhor compreensão do seu funcionamento, possibilitando a identificação de aspectos que precisam ser refinados. Além disso, os experimentos realizados com o protótipo servirão para ajustar os parâmetros ϕ, ρ, λ_{max} e t_q usados na reconfiguração da rede.

Referências

[1] Y. Amir and C. Danilov. Reliable Communication in Overlay Networks. In *Proc. Int'l Conf. on Dependable Systems and Networks*, pages 511–520, San Francisco, CA, June 2003.

[2] Y. Amir, C. Danilov, and C. Nita-Rotaru. High Performance, Robust, Secure and Transparent Overlay Network Service. In *Proc. Int'l Workshop on Future Directions in Distributed Computing*, Bertinoro (Italy), June 2002.

[3] D. G. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. 18th ACM Symp. on Operating Systems Principles*, pages 131–145, Banff, AB (Canada), Oct. 2001.

[4] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens. An On-Demand Secure Routing Protocol Resilient to Byzantine Failures. In *Proc. ACM Workshop on Wireless Security*, pages 21–30, Atlanta, GA, Sept. 2002.

[5] Y. Deswarte, L. Blain, and J.-C. Fabre. Intrusion Tolerance in Distributed Computing Systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 110–121, Oakland, CA, 1991.

[6] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, Nov. 1976.

[7] C. M. Ellison. SPKI Requirements. RFC 2692, Internet Engineering Task Force, Sept. 1999.

[8] C. M. Ellison, B. Frantz, B. W. Lampson, R. L. Rivest, B. Thomas, and T. Ylönen. SPKI Certificate Theory. RFC 2693, Internet Engineering Task Force, Sept. 1999.

[9] J. S. Fraga and D. Powell. A Fault and Intrusion-Tolerant File System. In *Proc. 3rd International Congress on Computer Security*, pages 203–218, Dublin (Ireland), Aug. 1985.

[10] J. Han, G. R. Malan, and F. Jahanian. Fault-Tolerant Virtual Private Networks within An Autonomous System. In *Proc. 21st Symp. on Reliable Distributed Systems*, pages 41–50, Suita (Japan), Oct. 2002.

[11] M. A. Hiltunen, R. D. Schlichting, and C. A. Ugarte. Building Survivable Services Using Redundancy and Adaptation. *IEEE Transactions on Computers*, 52(2):181–194, Feb. 2003.

[12] A. Householder, A. Manion, L. Pesante, G. M. Weaver, and R. Thomas. Managing the Threat of Denial-of-Service Attacks. CERT Coordination Center, Oct. 2001.

[13] Y.-C. Hu, A. Perrig, and D. B. Johnson. Ariadne: A Secure On-Demand Routing Protocol for Ad Hoc Networks. In *Proc. 8th Annual Int'l Conf. on Mobile Computing and Networking*, pages 12–23, Atlanta, GA, Sept. 2002.

[14] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proc. ACM SIGCOMM Conf.*, pages 61–72, Pittsburgh, PA, Aug. 2002.

[15] P. Papadimitratos and Z. J. Haas. Securing the Internet Routing Infrastructure. *IEEE Communications Magazine*, 40(10):60–68, Oct. 2002.

[16] R. J. Perlman. *Network Layer Protocols with Byzantine Robustness*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, Jan. 1988.

[17] P. Veríssimo, N. F. Neves, and M. Correia. Intrusion-Tolerant Architectures: Concepts and Design. DI/FCUL TR 03-05, Department of Informatics, University of Lisbon, Apr. 2003.

Diagnóstico em Nível de Sistema Baseado em Computação Evolucionária

Bogdan Tomoyuki Nassu, Aurora T. Ramirez Pozo, Elias Procópio Duarte Jr.

Departamento de Informática – Universidade Federal do Paraná (UFPR)
Caixa Postal 19.018 – CEP 81.531-990 – Curitiba – PR – Brasil

{bogdan, aurora, elias)@inf.ufpr.br

Abstract. *The size and complexity of systems based on multiple processing units asks for the employment of techniques for automatic diagnosis of these units. System-level diagnosis consists in determining which units in a system are faulty and which are fault-free. This work describes evolutionary algorithms that can be used to accomplish diagnosis. A simple and a specialized genetic algorithm, as well as variants of the PBIL and Compact GA algorithms were implemented. Experimental results show a comparison of the performances of these algorithms.*

Resumo. *O aumento no tamanho e complexidade dos sistemas compostos por múltiplas unidades de processamento torna necessário o uso de técnicas para o diagnóstico automático destas unidades. O diagnóstico em nível de sistema consiste em determinar quais unidades do sistema estão falhas e quais não estão. Este trabalho descreve algoritmos evolutivos que podem ser usados para realizar o diagnóstico. Foram implementados um algoritmo genético simples e um especializado, além de variantes dos algoritmos PBIL e AG compacto. Resultados experimentais mostram uma comparação entre o desempenho destes algoritmos.*

1. Introdução

Um sistema composto por múltiplas unidades de processamento é definido como sendo uma coleção de n unidades heterogêneas, representadas por um conjunto $U = \{u_1, ..., u_n\}$. Estas unidades podem falhar, comprometendo o funcionamento do sistema. O aumento no tamanho e complexidade de sistemas deste tipo torna necessário o uso de técnicas para o diagnóstico automático de falhas em suas unidades. O diagnóstico em nível de sistema consiste em determinar quais unidades do sistema estão falhas e quais não estão. A partir desta informação, pode-se tomar providências como a substituição ou conserto das unidades falhas.

O modelo clássico para considerar as falhas em nível de sistema é aquele apresentado em [Preparata, Metze e Chien 1967], o modelo PMC. Neste modelo, cada unidade é capaz de testar outras unidades de forma a determinar seu estado de maneira inequívoca. Em um sistema em situação de falha existe um certo subconjunto F de unidades permanentemente falhas, chamado conjunto de falhas. Assume-se que o estado das unidades em F não muda durante o diagnóstico, e que os resultados dos testes feitos por estas unidades são indefinidos, ou seja, uma unidade falha pode "mentir", afirmando que uma outra unidade falha não está falha ou vice-versa. O diagnóstico consiste na determinação do estado das unidades do sistema a partir dos testes realizados. Este

diagnóstico é feito por uma unidade central que nunca falha, para a qual os resultados dos testes feitos são enviados.

No modelo PMC, define-se um grafo direcionado G(U,*E*), no qual cada unidade $u_i \in U$ é um vértice, e cada aresta (u_i, u_j) pertence a E se e somente se u_i testa u_j.

A cada unidade $u_i \in U$, associa-se dois conjuntos:

$$\Gamma(u_i) = \{u_j : (u_i, u_j) \in E\} \text{ e}$$

$$\Gamma^{-1}(u_i) = \{u_j : (u_j, u_i) \in E\}$$

O primeiro contém as unidades u_j testadas por u_i, e o segundo as unidades uj que testam u_i. A estes conjuntos, associam-se os valores $d_{in}(u_i) = |\Gamma^{-1}(u_i)|$ e $d_{out}(u_i) = |\Gamma(u_i)|$, que definem, respectivamente, o número de unidades que testam u_i e o número de unidades testadas por u_i.

Após os testes, a cada aresta $(u_i, u_j) \in E$ associa-se um resultado de teste w_{ij}. Se a unidade u_i testa u_j como sendo falha, o valor 1 é atribuído a w_{ij}, do contrário o seu valor é 0. Dadas as asserções do modelo PMC, w_{ij} é um resultado confiável se e somente se u_i é uma unidade sem falhas. O conjunto de todos os resultados dos testes feitos é a síndrome *S* do sistema. A síndrome é definida como uma função de mapeamento *W*, definida tal que $W(u_i, u_j) = w_{ij}$. O diagnóstico é realizado pela unidade central a partir de *S*. Uma síndrome *S* é dita compatível com um conjunto de falhas *F* se, para toda aresta $(u_i, u_j) \in E$ tal que u_i não é falha, $W(u_i, u_j) = 1$ se e somente se u_j é falha. São definidos também os subconjuntos $S(u_i)$ e $S^{-1}(u_i)$ de *S* como sendo os conjuntos dos resultados dos testes feitos por u_i e os resultados dos testes feitos sobre u_i, respectivamente.

Um sistema é dito t-diagnosticável se todas as unidades falhas do sistema podem ser identificadas sem erro desde que o seu número não ultrapasse *t*. É provado que, se duas unidades não se testam mutuamente, um sistema é t-diagnosticável se $n \geq 2t + 1$ e, para toda unidade $u_i \in U$, $d_{in}(u_i) \geq t$, ou seja, se cada unidade é testada por pelo menos t outras unidades [Preparata, Metze e Chien 1967]. O diagnóstico é um problema bem conhecido [Masson, Blough e Sullivan 1996], e diversas soluções eficientes já foram desenvolvidas. Porém, o problema de se determinar o estado de todas as unidades de um sistema t-diagnosticável a partir dos resultados dos testes feitos carece de métodos eficientes. Por este motivo, em [Elhadef e Ayeb 2000] é apresentada uma solução para este problema baseada no uso de algoritmos genéticos.

Algoritmos genéticos, ou AGs, são algoritmos de busca baseados nos princípios da seleção natural e recombinação gênica [De Jong 1975], bastante usados na otimização de funções. Existem também diversas técnicas evolutivas que derivam dos AGs, e que usam a distribuição probabilística das soluções como ferramenta para guiar a busca. Entre estas, pode-se destacar o *Population-Based Incremental Learning* - PBIL [Baluja 1994, Baluja e Caruana 1995] e os AGs compactos [Harik, Lobo e Goldberg 1998].

O objetivo deste trabalho é comparar o desempenho de quatro tipos de algoritmos evolutivos quando os mesmos são usados para resolver o diagnóstico em nível de sistema. O primeiro tipo é um algoritmo genético simples, cujos resultados são usados apenas como base para a comparação. O segundo tipo é um algoritmo genético especializado, baseado naquele proposto em [Elhadef e Ayeb 2000]. Os outros dois algoritmos são implementações especializadas do PBIL e de um AG compacto, com otimizações desenvolvidas especificamente para este trabalho.

A seção 2 apresenta um algoritmo genético simples que pode ser usado para resolver o diagnóstico em nível de sistema. A seção 3 apresenta outros algoritmos evolutivos que podem ser usados para resolver este problema: um AG especializado, o PBIL e o AG compacto. A seção 4 descreve os experimentos realizados e os resultados obtidos nos mesmos, e discorre sobre tais resultados. A seção 5 traz as conclusões e considerações finais do artigo.

2. Um Algoritmo Genético Simples para Diagnóstico em Nível de Sistema

Algoritmos genéticos, ou AGs, são algoritmos de busca baseados nos princípios biológicos da seleção natural e da recombinação gênica [De Jong 1975]. Eles são bastante usados na otimização de funções. Os AGs são capazes de encontrar rapidamente regiões do espaço de busca com alto desempenho, e são resistentes a ruídos nos dados de entrada, mas como outros métodos heurísticos por vezes não conseguem encontrar uma solução ótima.

O funcionamento de um AG é baseado na sobrevivência dos indivíduos mais aptos no decorrer de uma série de gerações. Cada indivíduo, ou cromossomo, é uma possível solução para o problema que se quer resolver. Em um AG padrão, as soluções são codificadas como vetores binários de tamanho fixo, nos quais cada bit é chamado de gene e cuja interpretação varia de acordo com o problema. Cada indivíduo faz parte de uma população, que contém diversas outras soluções. A população inicial é definida de forma aleatória. Ela evolui por um determinado número de gerações, de forma que os indivíduos tendam a se aproximar da solução ótima. Para guiar esta evolução, uma função que calcula o *fitness* de cada indivíduo é definida. O *fitness* é uma medida da qualidade de uma solução, e é usado para selecionar os indivíduos sobre os quais serão aplicados certos operadores genéticos, que podem modificar características de uma solução ou combinar partes de duas soluções. Os indivíduos gerados por estes operadores não possuem necessariamente *fitness* maiores que os dos seus "pais", mas a pressão seletiva faz com que haja uma tendência nesta direção. Após a passagem de um certo número de gerações, se a solução ótima não foi encontrada, a melhor solução da população é tomada como resultado final.

Os operadores genéticos normalmente usados são a reprodução, a seleção elitista, o *crossover* e a mutação. A reprodução é simplesmente a escolha de alguns indivíduos da população que são replicados de uma geração para a outra. Os indivíduos são selecionados com base no seu *fitness*: quando mais apto é o indivíduo, maior a sua chance de sobrevivência. A seleção elitista consiste na manutenção do indivíduo com melhor *fitness* de cada geração na geração seguinte. Garantindo a permanência desta solução na população, obtém-se uma ascensão monotônica do *fitness* do melhor indivíduo através das gerações. O *crossover* consiste na combinação de características de dois indivíduos selecionados de forma aleatória da população. Assim como ocorre na reprodução, indivíduos com valores mais altos de *fitness* possuem maiores chances de serem escolhidos. Existem várias formas de se realizar o *crossover*, com resultados que variam de acordo com o problema. Uma mutação é a inversão de um bit de um indivíduo. Geralmente, a chance de ocorrer uma mutação é pequena, pois do contrário os resultados do algoritmo poderiam se tornar completamente aleatórios.

Uma diferença fundamental entre os algoritmos genéticos e outras heurísticas é o seu paralelismo [Baluja 1994]: enquanto muitas heurísticas trabalham com um único ponto no espaço de busca, em um AG cada indivíduo da população representa um ponto.

Portanto, para que o AG faça uma busca exaustiva, deve-se tomar precauções para garantir que haja uma certa diversidade genética. Quando esta diversidade é perdida, o algoritmo pode ficar preso em um grupo de soluções sub-ótimas. Dois dos mecanismos acima descritos são usados com este propósito. Um deles é a seleção probabilística dos indivíduos usados no *crossover*. Indivíduos com *fitness* baixo possuem chances menores de serem escolhidos, mas ainda assim podem ser usados e suas características podem se perpetuar em gerações futuras. O outro é a mutação, que introduz variações aleatórias que ajudam a população a escapar de ótimos locais.

Quando se trabalha com algoritmos genéticos, diversas questões precisam ser tratadas, tais como a correta definição da função que calcula o *fitness*, a certificação de que informações importantes não são perdidas por conta de escolhas aleatórias e a representação eficiente do problema. O tamanho da população também é importante: populações maiores aumentam as chances de convergência, mas também aumentam o custo computacional do algoritmo. Portanto, para que se resolva o diagnóstico em nível de sistema através do uso de um AG, certas definições devem ser feitas. As definições aqui apresentadas para a represetnação do problema e para o cálculo do *fitness* foram propostas em [Elhadef e Ayeb 2000].

2.1. Representação do Problema

No AG considerado por este trabalho, um cromossomo representa uma possível situação de falha do sistema. Cada cromossomo v é representado por uma *string* de bits, com cada gene representando o estado de uma unidade do sistema (falha ou sem falha). Assim, um sistema com n unidades é representado por uma *string* de comprimento n. No exemplo abaixo, o conjunto de unidades falhas é $F = \{u_1, u_4, u_8\}$:

(10010001)

Para o problema do diagnóstico em nível de sistema, certos cromossomos são considerados ilegais. Estes são aqueles que não satisfazem as condições necessárias de um sistema t-diagnosticável. Por exemplo, os seguintes cromossomos são ilegais para um sistema t-diagnosticável com $t = 4$:

(000000000) e (111001110)

No primeiro caso, o cromossomo é ilegal porque em uma situação de falha pelo menos uma das unidades deve estar falha, e o segundo porque o número de unidades falhas é maior que t.

2.2. Cálculo do *Fitness*

Para este problema, uma boa medida do *fitness* de uma solução em potencial é a probabilidade da mesma estar correta. Para um cromossomo ilegal, esta probabilidade é nula. Para os demais, esta probabilidade pode ser obtida gerando-se uma síndrome S, compatível com a solução v, e comparando-se esta síndrome com a síndrome S^*, obtida nos testes realizados pelas unidades do sistema. A geração da síndrome S deve ser feita de tal forma que a mesma seja idêntica a S^* se e somente se v representa a solução ótima, ou seja, a situação real de falhas do sistema.

Sejam $F(v)$ o conjunto de unidades falhas de um cromossomo v (aquelas unidades cujos genes possuem valor 1); $v[i]$ o i-ésimo gene de v e $W_S(u_i, u_j)$ o resultado do teste (u_i, u_j) em uma síndrome S. A síndrome S compatível com v é gerada segundo as regras:

1) para todo $u_i \in F(v)$ e todo $u_j \in \Gamma(u_i)$, $W_S(u_i, u_j) = W_{S*}(u_i, u_j)$; e

2) para todo $u_i \in U - F(v)$ e todo $u_j \in \Gamma(u_i)$, $W_S(u_i, u_j) = w_{ij}$.

A regra 1 diz que se uma unidade é falha em v, em S os seus testes possuem os mesmos resultados que na síndrome "real" S^*. A regra 2 diz que se uma unidade u_i não é falha em v, em S o resultado do teste $W_S(u_i, u_j)$ é 1 se u_j é falha em v, ou 0 do contrário. Desta forma, se o conjunto $F(v)$ corresponde ao conjunto de falhas F do sistema, S e S^* são idênticos e o cromossomo v é consistente com o estado real do sistema.

Por exemplo, supondo um sistema com $n = 3$ e $t = 1$ tal que a síndrome S^* tem $W_{S*}(u_1, u_2) = 1$, $W_{S*}(u_2, u_3) = 0$ e $W_{S*}(u_3, u_1) = 1$. A síndrome S gerada pelo cromossomo (001) tem $W_S(u_1, u_2) = 0$, $W_S(u_2, u_3) = 1$ e $W_S(u_3, u_1) = 1$. O cromossomo (100) gera a síndrome S com $W_S(u_1, u_2) = 1$, $W_S(u_2, u_3) = 0$ e $W_S(u_3, u_1) = 1$. Neste caso, S e S^* são idênticos, e o cromossomo (100) corresponde à solução ótima.

A função de *fitness* usada neste trabalho considera que cada gene de um cromossomo possui seu próprio *fitness*. O *fitness* do cromossomo é a soma normalizada dos *fitness* dos seus genes. A função f que calcula o *fitness* de $v[i]$, o i-ésimo gene de v, é definida como:

$$f(v[i]) = \frac{f_{out}(v[i]) + f_{in}(v[i])}{2}, \text{ onde}$$

$$f_{in}(v[i]) = \frac{\left|S^{-1}(u_i) \cap (S^*)^{-1}(u_i)\right|}{d_{in}(u_i)} \text{ e}$$

$$f_{out}(v[i]) = \begin{cases} 1, \text{se } d_{out}(u_i) = 0, \\ \dfrac{\left|S(u_i) \cap S^*(u_i)\right|}{d_{out}(u_i)} \text{ do contrário.} \end{cases}$$

$f_{in}(v[i])$ computa o número normalizado de testes efetuados sobre a unidade u_i cujos resultados na síndrome gerada S são idênticos aos seus correspondentes na síndrome S^*. Da mesma forma, $f_{out}(v[i])$ computa o número normalizado de testes efetuados pela unidade u_i com resultados iguais em S e S^*. Se u_i não efetua testes, $f_{out}(v[i]) = 1$. Assim, o *fitness* do i-ésimo bit considera a unidade u_i como testadora e testada. $f(v[i])$ pode ser vista como a probabilidade do estado de u_i em v estar correto.

O *fitness FT* do indivíduo v é a soma normalizada dos *fitness* dos seus bits, ou seja:

$$FT(v) = \frac{\sum_{i=1}^{n} f(v[i])}{n}$$

Se o cromossomo v corresponde à solução ótima, ou seja, se v representa o estado real do sistema, então $FT(v) = 1$. Portanto, a condição de término do algoritmo é a obtenção de um indivíduo cujo *fitness* é igual a 1.

No exemplo anterior, se $v = (001)$, $f(v[1]) = 0.5$, $f(v[2]) = 0$ e $f(v[3]) = 0.5$, então $FT(v) = 0.33$. Se $v = (100)$, o *fitness* de todos os cromossomos é igual a 1, e $FT(v) = 1$.

2.3. População Inicial e Operadores Genéticos

Para este AG simples, a população inicial é definida de forma completamente aleatória. Indivíduos ilegais podem ser gerados, mas seu *fitness* será igual a 0. Esta é uma abordagem ingênua, que pode ser melhorada de outras formas, como é mostrado na seção 3.1.

Os operadores genéticos utilizados também são simples. Em uma população de tamanho *p*, são selecionados *p* indivíduos que serão usados para gerar a nova população. Estes indivíduos formam o conjunto de reprodutores, e são escolhidos através de um método probabilístico conhecido como técnica da roleta. Este método define uma chance de escolha para cada indivíduo a partir do seu *fitness*. Como o sorteio é feito sempre sobre toda a população atual, cada indivíduo pode ser selecionado mais do que uma vez. O conjunto de reprodutores é então usado para gerar a nova população: uma proporção deles é reproduzida sem alterações e uma outra parte é usada na operação de *crossover*. A chance de ocorrer um *crossover* é dada por uma taxa que permanece constante durante a execução do algoritmo. Para o *crossover*, toma-se um par de cromossomos de tamanho *n* e um valor aleatório *pos* entre 1 e *n*, que define a posição onde os cromossomos são "rompidos", gerando um novo par de cromossomos. Por exemplo, tendo *pos* = 3, os cromossomos *v1* = (011|001000) e *v2* = (101|110000) são substituidos por *v1*' = (011|110000) e *v2*' = (101|001000). Por fim, cada bit de cada indivíduo da nova população tem uma pequena chance de sofrer uma mutação, que inverte o seu valor. O *crossover* e a mutação podem gerar indivíduos ilegais. Estes cromossomos terão *fitness* igual a 0.

3. Outros Algoritmos Evolutivos para Diagnóstico em Nível de Sistema

Na sessão 2, foi apresentado um algoritmo genético simples que pode ser usado para o diagnóstico em nível de sistema. Este AG pode ser melhorado através de técnicas que tiram proveito do conhecimento do domínio do problema, criando um AG especializado. Pode-se também fazer uso de outros algoritmos evolutivos, como o *Population-Based Incremental Learning* - PBIL [Baluja 1994, Baluja e Caruana 1995], e o AGs compacto [Harik, Lobo e Goldberg 1998]. Esta seção mostra como estes algoritmos podem ser usados para realizar o diagnóstico. Os indivíduos gerados pelos algoritmos descritos nesta seção possuem a mesma representação daqueles gerados pelo AG simples, e são avaliados pela mesma função de *fitness*. O AG especializado é baseado naquele apresentado em [Elhadef e Ayeb 2000], com algumas pequenas melhorias. As implementações do PBIL e do AG compacto foram criadas especificamente para este trabalho.

3.1. Um AG Especializado para Diagnóstico em Nível de Sistema

Abaixo, são apresentadas algumas técnicas que melhoram aspectos do AG usado para o diagnóstico em nível de sistema.

3.1.1. População Inicial

O AG simples cria uma população inicial completamente aleatória. Esta população inicial pode ser gerada de maneira mais eficiente, diminuindo o número total de gerações necessárias para a obtenção da solução ótima. O AG especializado melhora a inicialização aleatória usando o seguinte algoritmo para gerar cada indivíduo da população inicial.

1. Uma unidade u_i é sorteada e definida como sendo livre de falhas.

2. Unidades testadas por u_i têm o seu estado definido pelos resultados dos testes feitos por u_i. Estes resultados são obtidos na síndrome do sistema, S^*.
3. Unidades que testem u_i em S^* como sendo falha são definidas como falhas.
4. Se o número de unidades falhas for maior que t ou todas as unidades tiverem o seu estado definido como sendo sem falha, u_i não pode estar sem falhas. O cromossomo é descartado e u_i não pode mais ser sorteado no passo 1 para gerar outros indivíduos.
5. Se nem todas as unidades tiverem o seu estado definido e não há unidades falhas no sistema, uma das unidades com estado indefinido é definida como sendo falha, e as demais são consideradas sem falha. Do contrário, todas as unidades com estado indefinido são consideradas sem falha.

Esta estratégia é capaz de evitar a geração de indivíduos ilegais.

3.1.2. Mutação

No AG simples, todos os bits de um cromossomo possuem uma chance igual de sofrerem mutação. O algoritmo especializado modifica a escolha dos bits que são invertidos, usando uma estratégia adaptativa. A função de *fitness* usada define um valor de *fitness* para cada bit do cromossomo. O AG especializado pode usar este valor para determinar quais bits sofrem mutação: se o *fitness* do bit for inferior à taxa de mutação, o mesmo é invertido. Se não houverem bits para se inverter por este critério, aquele com menor *fitness* tem uma chance igual à taxa de mutação de ser invertido.

Para evitar a geração de indivíduos ilegais, são usadas duas estratégias. Sejam v um cromossomo e i o bit que será invertido. Se $|F(v)| = 1$ e $v[i] = 1$, i é a única unidade falha do cromossomo, e a inversão do i-ésimo bit irá gerar um cromossomo que não possui unidades falhas. Se isto ocorrer, o j-ésimo bit, correspondente ao testador de i com menor *fitness*, também é invertido. Se $|F(v)| = t$ e $v[i] = 0$, a inversão do i-ésimo bit irá gerar um cromossomo com mais que t unidades falhas. Neste caso, o estado da unidade falha com menor *fitness* é invertido. Desta forma, pode-se inverter o bit i sem que se gere um indivíduo ilegal.

3.1.3. Crossover

A geração de cromossomos ilegais através do *crossover* é resolvida da seguinte forma: se o *crossover* de dois cromossomos gerar um indivíduo ilegal, a operação é ignorada e o indivíduo é gerado como uma cópia do seu primeiro pai. Nos testes feitos em [Elhadef e Ayeb 2000], os melhores resultados foram obtidos quando as menores taxas de *crossover* foram usadas. Desta forma, pode-se questionar a importância do *crossover* para a evolução neste problema. Por este motivo, a implementação feita para este trabalho prevê também a possibilidade de uma taxa de *crossover* nula.

3.2. PBIL – *Population-Based Incremental Learning*

A população de um AG guarda informações relativas aos pontos do espaço de busca já visitados pelo algoritmo. Os operadores de seleção e *crossover* podem ser vistos como maneiras de fazer uso desta informação. A compreensão do papel e do funcionamento da população e dos operadores genéticos dos AGs possibilitou a criação de uma outra classe de algoritmos, que substitui a população, o *crossover* e a seleção por outra técnica: a distribuição probabilística dos genes. O PBIL (*Population-Based Incremental Learning*)

[Baluja 1994, Baluja e Caruana 1995], apresentado nesta seção, é um algoritmo evolutivo baseado neste conceito.

O PBIL cria um vetor de valores reais, que reprentam a probabilidade de cada gene de um cromossomo ter valor igual a 1. Estas probabilidades são iniciadas com um valor de 0.5 ou 50%, ou seja, cada gene tem uma chance igual de assumir um valor 0 ou 1. Com o passar das gerações, este vetor é atualizado de forma a representar indivíduos com *fitness* altos. A cada geração, uma nova população é criada a partir do vetor de probabilidades.

O PBIL é caracterizado por três parâmetros. O primeiro é o tamanho da população gerada a cada iteração do algoritmo. O segundo é a taxa de aprendizado, que diz o quanto cada bit do vetor de probabilidades é deslocado na direção das boas soluções a cada geração. O terceiro define quantos indivíduos são usados na atualização do vetor de probabilidades a cada geração. A escolha destes indivíduos pode ser estendida, de forma que o algoritmo aprenda a partir de exemplos negativos. Assim, além de ser atualizado na direção dos x melhores exemplos, o vetor de probabilidades pode ser atualizado na direção contrária aos y piores exemplos. O PBIL também pode fazer uso da mutação: uma mutação acrescenta uma pequena variação positiva ou negativa a um bit no vetor de probabilidades.

Quando se usa o PBIL para o diagnóstico em nível de sistema, o vetor de probabilidades pode ser gerado de uma forma otimizada, que exclui logo de início certos indivíduos ilegais. Isto é feito através do seguinte algoritmo, executado para cada posição i no vetor de probabilidades.

1. A unidade $u_i \in U$ é tomada como sendo livre de falhas.
2. Unidades testadas por u_i têm o seu estado definido pelos resultados dos testes feitos por u_i. Estes resultados são obtidos na síndrome do sistema, S^*.
3. Unidades que testem u_i em S^* como sendo falha são definidas como falhas.
4. Se o número de unidades falhas for maior que t ou todas as unidades tiverem o seu estado definido como sendo sem falha, a unidade u_i não pode estar sem falhas, ou seja, é certo que está falha, e o valor da posição i do vetor de probabilidades é definido como sendo 1. Do contrário, este valor é igual a 0.5.

Para evitar a geração de cromossomos ilegais a partir de um vetor de probabilidades, usa-se a seguinte estratégia: após a criação do indivíduo, se este não possui unidades falhas, a unidade cujo bit possui menor *fitness* é definida como sendo falha. Se exitirem mais que t unidades falhas, aquelas cujos bits possuirem os menores valores de *fitness* são consideradas como sendo sem falhas, até que o número de unidades falhas seja menor ou igual a t.

3.3. AG Compacto

Um outro algoritmo que, assim como o PBIL, substitui os operadores de seleção e *crossover* pela distribuição probabilística dos genes, é o Algoritmo Genético Compacto [Harik, Lobo e Goldberg 1998], ou AGc. Este algoritmo reduz de forma significativa os requisitos de memória pois, ao contrário dos outros algoritmos até aqui considerados, não tem a necessidade de manter uma população.

Assim como o PBIL, o AGc faz uso de um vetor de valores reais que representam probabilidades. No início do algoritmo, todas as posições possuem o valor 0.5, ou 50%.

Este vetor é usado para gerar de forma probabilística 2 indivíduos. Estes indivíduos são então comparados, e o vetor de probabilidades é atualizado na direção do "vencedor", ou seja, daquele que possuir *fitness* mais alto. Cada bit é atualizado independentemente: se o vencedor possui um bit em 1 e o perdedor em 0, o bit correspondente é incrementado no vetor de probabilidades. Da mesma forma, se o vencedor possui um bit em 0 e o perdedor em 1, o bit é decrementado no vetor de probabilidades. Se um bit possui valor igual no vencedor e no perdedor, seu valor não é alterado no vetor de probabilidades. Os bits do vetor de probabilidades são incrementados ou decrementados de um valor *E*. Estudos mostram que um AGc com $E = 1/n$ se comporta de forma similar a um AG simples com população de tamanho n [Harik, Lobo e Goldberg 1998 e Harik 1999].

O AGc pode ser usado para realizar o diagnóstico em nível de sistema de forma similar ao PBIL. Os procedimentos para inicialização do vetor de probabilidades e para evitar a criação de indivíduos ilegais podem ser repetidos sem alterações.

4. Experimentos

Os algoritmos apresentados nas seções 2 e 3 foram implementados para ter o seu desempenho comparado quando usados para resolver o diagnóstico em nível de sistema. Foram avaliadas algumas variações dos algoritmos, resultando em um total de seis diferentes configurações: AG simples, AG especializado com e sem *crossover*, PBIL com e sem o uso de exemplos negativos e AG compacto.

4.1. Descrição dos Experimentos

Para os experimentos foram usados grafos de teste para sistemas t-diagnosticáveis com diferentes tamanhos. Estes grafos de teste foram gerados com cada unidade $u_i \in U$ testando as unidades $u_{(i+1) \bmod n} .. u_{(i+t) \bmod n}$. Como $n \geq 2t + 1$, cada unidade será testada por t outras unidades e nenhum par de unidades se testará mutuamente. A cada teste, um conjunto de unidades falhas foi definido selecionando-se aleatoriamente n_f unidades como sendo falhas, tal que $n_f \leq t$. A partir do grafo de testes e do conjunto de unidades falhas, a síndrome S^* do sistema foi gerada, com os resultados dos testes executados pelas unidades falhas sendo definido aleatoriamente.

Para evitar possíveis problemas com arredondamentos e melhorar o desempenho dos algoritmos, a função para cálculo do *fitness* foi modificada de forma a não fazer normalizações, ou seja, as divisões foram removidas. Desta forma, pode-se trabalhar apenas com valores inteiros, e o *fitness* varia de 0 a $2nt$, ao invés de 0 a 1.

Os algoritmos foram experimentados com diferentes configurações. Foram escolhidas aquelas configurações que apresentaram o melhor desempenho médio em 10 execuções com $n = 81$ e $t = 40$ (o caso mais complexo considerado nestes experimentos). As configurações escolhidas foram:

- AG simples: população = 7, taxa de *crossover* = 0.1 e taxa de mutação = 0.01.
- AG especializado com *crossover*: população = 7, taxa de *crossover* = 0.05 e taxa de mutação = 0.01.
- AG especializado sem *crossover*: população = 7 e taxa de mutação = 0.01.
- PBIL sem uso de exemplos negativos: população = 7, taxa de atualização = 0.2, taxa de mutação = 0.08 e número de indivíduos usados na atualização = 1.

- PBIL com uso de exemplos negativos: população = 7, taxa de atualização positiva = 0.2, taxa de atualização negativa = 0.1, taxa de mutação = 0.08 e número de indivíduos usados na autalização = 1.
- AG compacto: taxa de atualização = 0.15 e taxa de mutação = 0.08.

Seja (*n, t*) um grafo de testes para um sistema *t*-diagnosticável de *n* unidades. Foram definidos os seguintes grafos de teste, com tamanho e número de falhas bastante variado: (81, 5), (81, 10), (81, 10), (81, 40), (41, 5), (41, 10), (41, 20), (21, 5), (21, 10), (11, 5) e (8, 3). Cada algoritmo foi executado 100 vezes para cada grafo de testes, e o seu desempenho médio foi computado.

4.2. Resultados dos Experimentos

As tabelas 1 a 4 apresentam os resultados obtidos nos experimentos realizados. A solução obtida corresponde àquela com melhor *fitness* após 500 gerações, ou à solução ótima, caso a mesma tenha sido obtida.

Tabela 1: Número de gerações para a obtenção da solução.

	AG Simples	AG Espec. c/ X-Over	AG Espec. s/ X-over	PBIL s/ ex. neg.	PBIL c/ ex. neg.	AGc
n = 8, t = 3	99.1	2.4	16.81	18.04	18.19	68.82
n = 11, t = 5	66.33	2.01	8.37	19.7	30.06	75.81
n = 21, t = 5	354.66	9.96	14.63	33.11	41.53	41.41
n = 21, t = 10	88.12	5.06	9.85	56.16	47.88	165.93
n = 41, t = 5	500	6.91	8.68	38.44	48.88	410
n = 41, t = 10	475.32	7.33	7.14	55.42	70.8	450
n = 41, t = 20	177.83	14.09	11.16	88.16	80.87	307.97
n = 81, t = 5	500	0.86	0.84	35.14	71.26	390
n = 81, t = 10	500	0.88	0.89	67.59	104.29	450
n = 81, t = 20	500	0.95	0.86	126.38	124.02	444
n = 81, t = 40	421,62	0.85	0.82	164.76	142.2	500

Tabela 2: *Fitness* médio da solução após no máximo 500 gerações.

	Máx. Possível	AG Simples	AG Espec. c/ X-Over	AG Espec. s/ X-over	PBIL s/ ex. neg.	PBIL c/ ex. neg.	AGc
n = 8, t = 3	48	47.34	48	47.94	48	47.98	48
n = 11, t = 5	110	110	110	109.76	110	110	110
n = 21, t = 5	210	94.5	210	210	210	210	201.12
n = 21, t = 10	420	420	420	420	420	420	420
n = 41, t = 5	410	0	410	410	410	410	398.46
n = 41, t = 10	820	113.38	820	820	820	820	763.38
n = 41, t = 20	1640	1639.38	1640	1640	1640	1640	1634.18
n = 81, t = 5	810	0	810	810	810	810	797.86
n = 81, t = 10	1620	0	1620	1620	1620	1620	1573
n = 81, t = 20	3240	0	3240	3240	3240	3240	3030.89
n = 81, t = 40	6480	6457.18	6480	6480	6480	6480	6132.8

Tabela 3: Porcentagem das rodadas que obtiveram a solução ótima.

	AG Simples	AG Espec. c/ X-Over	AG Espec. s/ X-over	PBIL s/ ex. neg.	PBIL c/ ex. neg.	AGc
n = 8, t = 3	92%	100%	97%	100%	99%	100%
n = 11, t = 5	100%	100%	99%	100%	100%	100%
n = 21, t = 5	45%	100%	100%	100%	100%	30%
n = 21, t = 10	100%	100%	100%	100%	100%	100%
n = 41, t = 5	0%	100%	100%	100%	100%	18%
n = 41, t = 10	9%	100%	100%	100%	100%	10%
n = 41, t = 20	99%	100%	100%	100%	100%	91%
n = 81, t = 5	0%	100%	100%	100%	100%	22%
n = 81, t = 10	0%	100%	100%	100%	100%	10%
n = 81, t = 20	0%	100%	100%	100%	100%	11%
n = 81, t = 40	62%	100%	100%	100%	100%	0%

Tabela 4: Tempo médio para a obtenção da solução (em segundos – 0 indica tempo inferior a 0.01 s).

	AG Simples	AG Espec. c/ X-Over	AG Espec. s/ X-over	PBIL s/ ex. neg.	PBIL c/ ex. neg.	AGc
n = 8, t = 3	0	0	0	0	0	0
n = 11, t = 5	0	0	0	0	0	0
n = 21, t = 5	0	0	0	0	0	0.25
n = 21, t = 10	0	0	0	0	0	0
n = 41, t = 5	0	0	0	0.01	0.04	5.52
n = 41, t = 10	0	0	0	0.12	0.29	6.76
n = 41, t = 20	0.71	0	0	0.96	0.93	2.23
n = 81, t = 5	0	0	0	0.76	2.06	43.58
n = 81, t = 10	0	0	0	2.48	4.87	81.05
n = 81, t = 20	0	0.02	0	10.15	12.45	116.2
n = 81, t = 40	26.31	0.85	0.21	16.3	15.74	50.16

Ao se analisar os resultados obtidos nos testes, é possível chegar a diversas conclusões sobre o desempenho dos algoritmos. O AG Simples é bastante eficiente no que diz respeito ao tempo de execução. Porém, os resultados obtidos foram os piores entre todos os algoritmos. Quando há uma grande diferença entre os valores de *n* e *t*, o algoritmo foi, na maior parte dos casos, incapaz de encontrar sequer uma solução que se aproximasse da solução ótima. Isto provavelmente ocorre porque nestes casos o número de indivíduos inválidos que é gerado é muito grande, o que dificulta a convergência dos resultados.

O AG Compacto teve os piores tempos de execução. Nos testes mais complexos, o número de gerações para a obtenção da solução foi bastante alto, mas diferente do AG Simples, o *fitness* médio das soluções se aproximou da solução ótima. Portanto, pode-se concluir que o AG Compacto teve uma convergência razoável, mas teve dificuldades para encontrar a solução ótima propriamente dita.

Na quase totalidade dos casos, o PBIL conseguiu encontrar a solução ótima após um certo número de gerações. Seu desempenho foi superado apenas pelo AG especializado. Pode-se verificar que, para este problema, o uso de exemplos negativos no aprendizado foi prejudicial, visto que na maioria dos testes a complexidade adicional não foi compensada por uma convergência mais rápida. De fato, no geral, a conversão foi mais lenta quando foram usados exemplos negativos.

O AG Especializado foi aquele que apresentou o melhor desempenho entre os algoritmos testados. Além de ter encontrado a solução ótima na grande maioria dos casos, o AG Especializado o fez com um número médio de gerações muito baixo. De fato, isto demonstra a importância de um bom algoritmo para a inicialização da população, já que em muitos casos a solução ótima foi encontrada na população inicial, ou após poucas gerações. Pode-se dizer que o AG Especializado sem *crossover* apresentou desempenho similar ao com *crossover*. Nos exemplos mais complexos, o número de gerações necessário para a obtenção da solução ótima foi bastante parecido para as duas abordagens, e a retirada do *crossover* fez o tempo médio de execução do algoritmo ser reduzido de forma discreta. Isto leva a crer que existe uma boa possibilidade de o *crossover* não ser o principal responsável pela convergência neste problema. Porém, visto que em grande parte dos exemplos a população inicial já continha a solução ótima, ou uma solução próxima a ela, o número de gerações dos testes foi bastante reduzido. Por este motivo, não é possível de se analisar de forma conclusiva o papel do *crossover* na convergência para este problema.

Os algoritmos testados podem ser ordenados de acordo com o seu desempenho nos experimentos, do melhor para o pior, da seguinte forma: AG Especializado sem

crossover; AG Especializado com *crossover*, PBIL sem exemplos negativos, PBIL com exemplos negativos, AG Compacto e AG Simples.

5. Conclusões e Trabalhos Futuros

Neste trabalho, foram apresentados algoritmos evolutivos que podem ser usados para realizar o diagnóstico em nível de sistema. Estes algoritmos foram implementados e testados, de forma a terem comparados os seus desempenhos. A partir dos resultados dos testes, pode-se dizer que a complexidade adicional de algoritmos mais sofisticados muitas vezes é compensada por uma convergência rápida dos resultados para a solução ótima. Nos casos mais complexos, esta compensação fica bastante evidente, com as abordagens mais simples (AG simples, AG compacto) não conseguindo encontrar a solução ótima ou mesmo convergir para uma boa solução, enquanto a abordagem mais complexa (AG Especializado) encontra rapidamente a solução ótima.

Entre trabalhos futuros, pode-se destacar a execução de mais experimentos que usem configurações diferentes para os algoritmos aqui apresentados ou outros algoritmos evolutivos, tais como o AG compacto extendido [Harik 1999] e o BOA [Peikan, Goldberg e Cantú-Paz 1999]. Também pode-se realizar trabalho semelhante com a aplicação de algoritmos evolutivos para o diagnóstico distribuído, no qual cada unidade do sistema realiza o diagnóstico.

Referências

Baluja, S. (1994) "Population-Based Incremental Learning: A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning", Tech. Rep. No. CMU-CS-94-163, Pittsburgh, PA, Carnegie Mellon University.

Baluja, S. & Caruana, R. (1995), "Removing the Genetics from the Standard Genetic Algorithm", Proceedings of ML-95, Twelfth International Conference on Machine Learning, A. Prieditis and S. Russel (Eds.), 1995, Morgan Kaufmann, pp. 38-46.

De Jong, K. (1975), "An Analysis of the Behavior of a Class of Genetic Adaptive Systems", University of Michigan, Tese de Ph.D.

Elhadef, M. & Ayeb, B. (2000), "Efficient Fault Identification in Diagnosable Systems: An Evolutionary Approach". University of Sherbrooke, Quebec, 2000.

Harik, G. (1999), "Linkage Learning via Probabilistic Modeling in the ECGA", IlliGAL Technical Report 99010, Urbana, IL: University off Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory.

Harik, G.R.; Lobo, F.G. & Goldberg, D.E. (1998), "The Compact Genetic Algorithm", In of Electrical, I., & Engineers, E. (Eds.), Proceedings of 1998 IEEE International Conference on Evolutionary Computation (pp. 523-528).

Peikan, M.; Goldberg, D.E. & Cantú-Paz, E. (1999), "BOA: The Bayesian Optimization Algorithm", Preceedings Genetic and Evolutionary Computation Conference 1999.

Preparata, F.P.; Metze, G. & Chien, R.T (1967), "On the Connection Assignment Problem of Diagnosable Systems", IEEE Trans. on Eletron. Comput., 16.

Masson, G.; Blough, D. & Sullivan, G (1996), "System Diagnosis", in Fault-Tolerant Computer System Design, ed. D. K. Pradhan, Prentice-Hall.

Improving Fault Tolerance to Radiation Effects in Integrated Systems

Gustavo Neuberger[1], Fernanda Kastensmidt[2], Ricardo Reis[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

[2]Universidade Estadual do Rio Grande do Sul (UERGS)
Estrada Santa Maria 2100 –Guaiba – RS – Brazil

{neuberg,reis}@inf.ufrgs.br, fernanda-lima@uergs.edu.br

Abstract. *This paper describes the radiation effects in integrated systems and discusses some techniques to mitigate these effects. The main circuits analyzed are SRAM memories and SRAM-based FPGAs. New techniques used to protect these circuits against these effects are also proposed in this work. One of the presented techniques to protect FPGAs is based on a combination of Double Modular Redundancy (DMR) with Concurrent Error Detection (CED) that can reduce overheads comparing to Triple Modular Redundancy (TMR). In the case of SRAM memories, a technique based on the Reed-Solomon Code and Hamming Code was developed, as well as a tool to generate a core for fault-tolerance that minimizes the area cost of the code. Some results are presented.*

1. Introduction

Fault-tolerance on semiconductor devices has been a meaningful matter since upsets were first experienced in space applications several years ago. Since then, the interest in studying fault-tolerant techniques to keep integrated circuits (ICs) operational in such hostile environment has increased, driven by all possible applications of radiation tolerant circuits, such as space missions, satellites, high-energy physics experiments and others [NASA 2002]. Spacecraft systems include a large variety of analog and digital components that are potentially sensitive to radiation and must be protected or at least qualified for space operation.

This paper reviews some techniques currently used to protect integrated circuits against radiation effects, and introduces new techniques developed to increase fault tolerance and/or reduce the cost involved to protection of the circuits. These techniques target fault tolerance in SRAM-based FPGAs and SRAM memories. The technique for FPGA combines time and hardware redundancy to reduce overhead comparing to the traditional Triple Modular Redundancy (TMR). It is based on duplication with comparison and concurrent error detection. The new technique proposed was specifically developed to cope with soft errors in the user combinational and sequential logic, while also reducing pin count, area and power dissipation. This technique is implemented in the high description level without modification in the FPGA architecture.

Concerning about memories, the techniques used are based on Error Detection and Correction Codes (EDAC). A popular one is Hamming code [Azumi 1975] that is a

preferred solution for implementation at circuit level due to area and performance reasons. However, it has limited error correction capability. The use of a different code, the well know Reed-Solomon code [Houghton 1997], is proposed in a hardware core. There are many different configurations of this code, such as the primitive polynomial, the number of bits to be protected, the size of the internal blocks and others. Because of this complexity, a tool able to automatically search for the best solution for a given set of parameters was developed. This tool aims to minimize the cost of the hardware implementation of this code. It generated a high level description of the code. Results show that it can be a very good alternative to Hamming code, with an efficient hardware implementation. The final implementation of Reed-Solomon code and Hamming code in a case study memory is presented and fault injection experiments have confirmed the reliability of the methods.

This paper is organized as follows. Section 2 describes the radiation effects on integrated circuits manufactured using CMOS process and specifically the effects in SRAM memories and in SRAM-based FPGA architectures. Section 3 discusses some SEU mitigation techniques that can be applied to protect circuits in general and that has been used in the past. Section 4 introduces a new high-level technique for designing fault-tolerant systems for SRAM-based FPGAs, without modifications in the FPGA architecture, able to cope with transient faults in the user combinational and sequential logic, while also reducing pin count, area and power dissipation compared to the traditional TMR. The section 5 shows a tool developed to create optimized hardware implementations of the Reed-Solomon code, an attractive alternative to the well-known Hamming code, but with more error correction capability. Section 6 presents a fault tolerant memory designed to be able to cope with all double bit upsets and a large amount of multiple bit upsets, combining Reed-Solomon and Hamming codes. The main conclusions are discussed in section 7, followed by the references.

2. Radiation Effects in Digital Circuits

The digital circuits located in the space environment are affected by the charged particles generated by the solar flares. When a charged particle hits the silicon, it can provoke a transient pulse, which can be interpreted as an internal signal [Bessot 1993]. This transient pulse can change the state of a memory cell. This phenomenon is called soft error or Single Event Upset (SEU). The consequences of SEUs are entirely device specific, and depend on the impact of the corrupted information in the system. A charged particle can also hit the combinational logic. The generated current pulse may be propagated by the logic and if latched by memory elements, it provokes a soft error (SEU) as well.

New generation of integrated circuits are becoming more sensitive to radiation effects. High-density devices require smaller feature size, this means less capacitance and hence information is stored with less charge. Lower voltage or lower power devices means that less charge or current is required to store information. Each of these effects makes the device more vulnerable to radiation and means that particles with little charge, which were once negligible, are now much more likely to produce upset or damage. As higher is the performance of a circuit more sensitive to radiation environment it is.

2.1. Radiation Effects in SRAM Memories

The most common circuit sensitive to SEU is the memory element. The memory cell is designed so that it has two stable states, one that represents a stored '0' and one that represents a stored '1'. In each state, two transistors are turned on and two are turned off (SEU target drains). A bit-flip in the memory element occurs when an energetic particle causes the state of the transistors in the circuit to reverse. This phenomenon occurs in many microelectronic circuits including memory chips and microprocessors. In a computer operating in space, for example, a bit-flip could randomly change critical data, randomly change the program, or confuse the processor to the point that it crashes.

Although SEU is the major concern in space applications, multiple bit upsets (MBU) start to be also a matter to be addressed in very deep submicron (VDSM) technologies. When a single high-energy ion passes through the silicon it can energize two or more adjacent memory cells [Reed 1997]. MBUs can be induced by direct ionization or nuclear recoil. The energy of the particle is more likely to provoke double bit upsets while multiple bit upsets are caused by an increase of the particle incident angle. Experiments in memories under proton and heavy ions fluxes have shown the probability of multiple upsets provoked by a single ion [Wrobel 2001], [Johansson 1999], [Buchner 2000].

2.2. Radiation Effects in FPGAs

In FPGAs, the upset has a peculiar effect when hit the combinational and sequential logic mapped into the programmable architecture. Let's take for example the SRAM-based FPGAs such as the Virtex® family from Xilinx that is one of the most popular programmable devices used in the market nowadays.

Virtex devices consist of a flexible and regular architecture composed of an array of configurable logic blocks (CLBs) surrounded by programmable input/output blocks (IOBs), all interconnected by a hierarchy of fast and versatile routing resources. The CLBs provide the functional elements for constructing logic while the IOBs provide the interface between the package pins and the CLBs. The CLBs are interconnected through a general routing matrix (GRM) that comprises an array of routing switches located at the intersections of horizontal and vertical routing channel. The Virtex matrix also has dedicated memory blocks called Block Select RAMs of 4096 bits each, clock DLLs for clock-distribution delay compensation and clock domain control, and two 3-State buffers (BUFTs) associated with each CLB.

Virtex devices are programmed using a bitstream, which contains all the information to configure the programmable storage elements in the matrix located in the Look-up Tables (LUT) and flip-flops, CLBs configuration cells and interconnections (figure 1). All these configuration bits are potentially sensitive to SEU and consequently they were our investigation targets.

In SRAM-based FPGA, both the user's combinational and sequential logic are implemented by customizable logic memory cells, in other words, SRAM cells, as represented in figure 1. When an upset occurs in the combinational logic synthesized in the FPGA, it corresponds to a bit flip in one of the LUTs cells or in the cells that control the routing. An upset in the LUT memory cell modifies the implemented combinational logic. It has a permanent effect and it can only be corrected at the next load of the

configuration bitstream. The effect of this upset is related to a stuck-at fault (one or zero) in the combinational logic defined by that LUT. An upset in the routing can connect or disconnect a wire in the matrix. It has also a permanent effect and its effect can be mapped to an open or a short circuit in the combinational logic implemented by the FPGA.

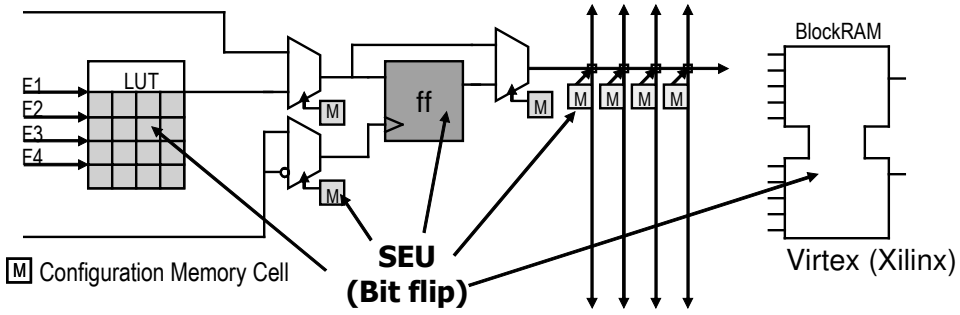

Figure 1. SEU Sensitive Bits in the CLB Tile Schematic

Radiation tests performed in Xilinx FPGAs [Fuller 2002][Carmichael 2001] show the effects of SEU in the design application and prove the necessity of using fault-tolerant techniques for space applications. A fault-tolerant system designed into SRAM-based FPGAs must be able to cope with the peculiarities mentioned in this section such as transient and permanent effects of a SEU in the combinational logic, short and open circuit in the design connections and bit flips in the flip-flops and memory cells.

3. SEU Mitigation Techniques

A SEU immune circuit may be fulfilled through a variety of mitigation techniques, including hardware, software, and device tolerance solutions. The most cost efficient approach may be an appropriate combination of SEU-hard devices and other mitigation solutions. However, the availability, power, volume, and performance of radiation-hardened devices may difficult their use. Hardware or software design also serves as effective mitigation, but design complexity may be a problem. A combination of the two may be a good select option.

Solutions to turn a logic device SEU tolerant can be implemented at different steps of the device development process. Possible solutions to design fault tolerant circuits are:

- Change of the technology process;
- Substitution of memory cells by modified ones;
- Use of Error Detection and Correction Codes (EDAC)
- Use of Triple Modular Redundancy (TMR)

3.1. Hardening by Technology

The SEU mitigation technique by technology consists in the use of a specific technology process to turn the entire device immune to radiation particles such as Silicon on Insulator (SOI) CMOS process [IBM 2000]. In this case the charged particle has much less chance to affect the device.

3.2. Hardened Memory Cells

The main idea is to provide CMOS memory cells with an appropriate feedback devoted to restore the data when it is corrupted by an ion hit. The principle is to store the data in two different locations within the cell in such way that the corrupted part can be restored. The main problem is how to organize the extra transistors used to realize the feedback that will result in new sensitive nodes, without affecting the SEU sensitivity.

The main advantages of this method are high performance (read/write time), low sensibility to temperature, technology process independence and voltage supply good SEU immunity. The main drawback is silicon area overhead. Many memory cells based on this approach have been developed in the last years, like IBM Memory Cell [Rockett 1992], NASA Memory Cell [Whitaker 1991] and HIT Memory Cell [Bessot 1993].

3.3. Hardening by EDAC

The Error Detection and Correction (EDAC) solutions [Label 1999] are examples of solutions that can be used to detect or/and correct SEUs when they occur. Some of them can achieve an acceptable level of reliability. An example is the Hamming Code technique. This approach can be used either in the circuit design or in the system level. Using Hamming code as a SEU mitigation solution in the design of a circuit, extra logic blocks are needed to code and decode the stored values such as registers and internal memory.

The use of EDAC is especially appropriated to protect high amounts of data, like in registers and memories. Some extra bits are necessary to store the redundancy bits, but the number of them is smaller than the number of data bits. But an extra circuit to encode and decode the data is needed, and it can also affect the performance of the system. Each code that can be used has a different level of protection, according to the model of faults used (single, multiple).

3.4. Hardening by TMR

Redundancy between modules, circuits or systems provides a potential mean of recovery from a SEU on a system. Autonomous or ground controlled switching from a prime system to a redundant spare may be an option, depending on spacecraft power and weight restrictions. With three identical circuits, the voter can choose the output that at least two agree upon. This approach is called Triple Modular Redundancy (TMR). The main drawbacks of this technique are the high area needed to triplicate the circuit, and that the voter must be designed in such way that no error can occur in the voter. In this case, the voter is designed using some previous techniques in the circuit or design level.

4. Protecting SRAM-based FPGAs against Soft Errors

The Triple Modular Redundancy (TMR) technique is a suitable solution for FPGAs because it provides a full hardware redundancy, including the user's combinational and sequential logic, the routing, and the I/O pads. However, it comes with some penalties because of its full hardware redundancy, such as area, I/O pad limitations and power dissipation. Many applications can accept the limitations (penalties) of the TMR approach but some cannot. Aiming to reduce the number of pins overhead of a full hardware redundancy implementation (TMR), and at the same time coping with

permanent upset effects, a new technique based on time and hardware redundancy to protect the user's combinational logic is presented, where the double modular redundancy with comparison (DWC) is combined with a time redundancy upset detection machine, able to detect upsets and to identify the correct value to allow continuous operation.

The reliability and the safety of TMR scheme compared to self-checking-based fault-tolerant schemes were discussed in [Lubaszewski 1998]. The experimental results presented that the higher the complexity of the module, the greater the difference in reliability between self-checking and TMR. In summary, the self-checking fault-tolerant scheme can achieve a higher reliability in comparison to the TMR if the self-checking overhead bound of 73% is not exceeded. The idea of using self-checking fault-tolerant scheme can be extended for FPGAs by using the duplication with comparison (DWC) method combined with concurrent error detection (CED) technique that it works as a self-checking. Figure 2 presents the scheme, called hot backup DWC-CED. The CED is able to detect which module is faulty in the presence of an upset, and consequently, there is always a correct value in the output of the scheme, because the mechanism is able to select the correct output out of two.

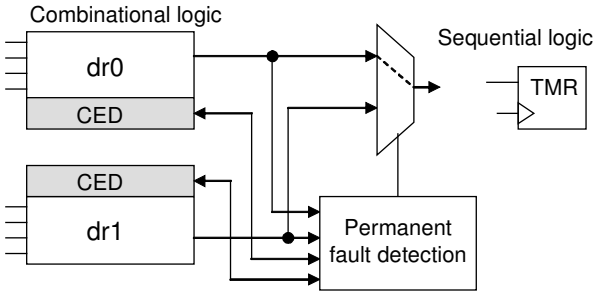

Figure 2. DWC combined with CED scheme

In the case of SEU detection in SRAM-based FPGAs, the CED must be able to identify permanent faults in the redundant modules. The CED works by finding the property of the analyzed logic block that can help to identify an error in the output in the presence of a permanent fault. There are many methods to implement logic to detect permanent faults, most solutions are based on time or hardware redundancy and they manifest a property of the logic block that is being analyzed. The CED scheme based on time redundancy recomputes the input operands in two different ways to detect permanent faults. During the first computation at time t0, the operands are used directly in the combinational block and the result is stored for further comparison. During the second computation at time t0+d, the operands are modified, prior to use, in such a way that errors resulting from permanent faults in the combinational logic are different in the first calculation than in the second and can be detected when results are compared. These modifications are seen as encode and decode processes and they depend on the characteristics of the logic block.

If an output mismatch occurs, the output register will hold its original value for one extra clock cycle, while the CED block detects the permanent fault. After this, the output will receive the data from the fault free module until the next reconfiguration (fault correction). The important characteristic of this method is that it does not incur a high performance penalty when the system is operating free of faults or with a single

fault. The method just needs one clock cycle in hold operation to detect the faulty module, and after that it will operate normally again without performance penalties. The final clock period is the original clock period plus the propagation delay of the encoders, decoders and output comparator.

This method has been named duplication with comparison combined to concurrent error detection block (DWC-CED) [Lima 2003]. The scheme is shown in figure 3. Module dr0 connects to register tr0 and module dr1 connects to register tr1. While the circuit performs the detection, the user's TMR register holds its previous value. When the free faulty module is found, register tr2 receives the output of this module and it will continue to receive this output until the next chip reconfiguration (fault correction). By default, the circuit starts passing the module dr0. A comparator at the output of dr0 and dr1 indicates an output mismatch (Hc). If Hc=0, no error is found and the circuit will continue to operate normally. If Hc=1, an error is characterized and the operands need to be re-computed by the encoder and decoder method to detect which module has the permanent fault. The detection takes one clock cycle. The encode and decode blocks implement the technique used to identify permanent faults in logic: re-computing with shifted operands (RESO) [Patel 1982].

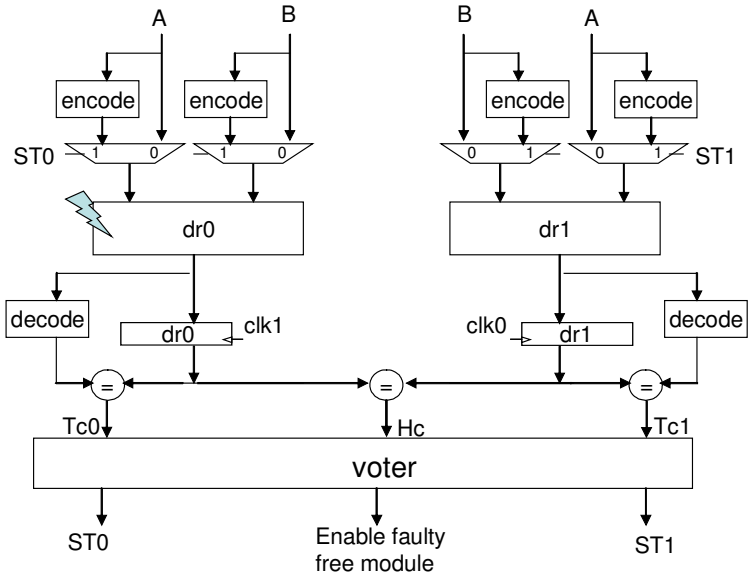

Figure 3. Fault tolerant technique based on DWC-CED for SRAM-based FPGAs

Some constraints must be observed for the perfect functioning of the technique, same as TMR: there must not be upsets in more than one redundant module, including the state machine detection and voting circuit, consequently it is important to use some assigned area constraints to reduce the probability of short circuits between redundant module dr0 and dr1. The scrubbing rate should be fast enough to avoid accumulation of upsets in two different redundant blocks. Upsets in the detection and voting circuit do not interfere with the correct execution of the system, because the logic is already triplicated. In addition, upsets in the latches of this logic are not critical, as they are refreshed in each clock cycle. Assuming a single upset per chip between scrubbing, if an upset alters the correct voting, it does not matter, as long as there is no upset in both redundant blocks.

Table 1 presents area results of 8x8 and 16x16 bits multipliers, implemented in the XCV300 FPGA using no fault tolerance technique, TMR technique and the proposed technique (DWC-CED). Results show that according to the size of the combinational logic block, it is possible to not only reduce the number of I/O pins but also area. Note that the 16x16 bits multiplier protected by TMR could not be synthesized in the prototype board that uses a Virtex part with 240 I/O pins (166 available for the user); while the same multiplier, implemented by the proposed technique could fit in the chip, and also occupy less area. In terms of performance, the TMR approach has presented an estimated frequency of 33.8 MHz, while the DMR-CED approach has presented a frequency of 26.7 MHz.

Table 1. Comparison of multiplier implementations (XCV300-PQ240)

Multipliers	Standard		TMR		DWC-CED	
Non-registered output	8x8	16x16	8x8	16x16	8x8	16x16
Total of I/O pads	32	64	96	192	66 (-31%)	130 (-32%)
Number of 4-LUTs	156	711	551	2159	425 (-23%)	1442 (-33%)
Number of ffs	0	0	0	0	34	66

5. Reed-Solomon Core Automatic Generator Tool

Error detection and correction code (EDAC) is a well-known technique to protect storage devices against transient faults because it can be implemented in a high-level design step, without changes in the mask process (low NRE cost). There are examples of SEU mitigation techniques using EDAC performed by software [Shirvani 2000], and by hardware [Redinbo 1993]. An example of EDAC is the Hamming code that is largely used to protect memories against SEU because of its efficient ability to correct single upsets with reduced area and performance overhead [Hentschke 2002]. However, Hamming code is not able to ensure reliability in Very Deep Submicron (VDSM) technology in presence of multiple bit upsets caused by the environment.

In the other hand, Reed-Solomon [Houghton 1997] is a block-based error correcting code, able to cope with multiple upsets. It has a wide range of applications in digital communications and storage. Reed-Solomon (RS) codes are used to correct errors in many systems including: storage devices, wireless or mobile communications, high-speed modems and others. RS encoding and decoding is commonly carried out in software. The RS code is based in the Finite Field Arithmetic. The main concepts of the arithmetic and the basic algorithm of the RS code can be found in [Neuberger 2004]. The code needs a polynomial to be created.

The first inconvenience of RS code in hardware implementation is the necessity of large tables of constants that usually are implemented in memory arrays. This structure presents a large area overhead for the code implementation and it cannot be easily integrated in a single chip because of its customization technology. A description of the RS code was developed based on multipliers that have completely replaced the table of constants [Neuberger 2003]. This solution has shown a good compromise in area, performance and reliability. However, for some code parameters, the high number of multipliers and the constants chosen can still present a non-optimal cost.

The main area cost of the RS code implemented in hardware lies on the size of the data block to be encoded, the generator polynomial used to create the multipliers and the size of the symbol, which implicates the set of constants to be used.

The main part of the Reed-Solomon encoding and decoding circuits are multiplications by several constants using a multiplier. To decrease the overhead in the overall circuit, the bigger effort can be spent to optimize the area used by this multiplier. The possible optimizations that can be done in the multiplier are:

- choice of the generator polynomial that produces the best multipliers,
- choice of the most appropriated constants for the multipliers.

To automatically develop an optimized RS code for the given characteristics, using the described techniques of optimization, a tool was developed. Based on the user's input, which represents a set of parameters, it creates the multipliers needed, calculates the cost associated with each one, and creates a VHDL description of the best possible hardware implementation of the code.

After the implementation of the automatic generator, some comparisons were made with previously manually implemented cores, to evaluate efficiency. The results shown in this section are based in the synthesis of the generated VHDL code in a Xilinx FPGA VirtexE V600EHQ240. The table 2 shows a comparison between codes with 7-bit symbols and 112-bit words. The first version was presented in [Neuberger 2003], and was designed manually, without the optimization techniques showed. The second version was manually generated too, but the constants to be used were carefully chosen, trying to reduce area overhead choosing constants with more zeros. These two versions were designed using the same polynomial (137). The third version was created using the automatic approach with the same polynomial. The constants chosen automatically were not the same as the ones used in the second version. The fourth version was automatically generated, but without the restriction of the polynomial to be used. The polynomial chosen this time was 131.

Table 2. Comparison between codes generated manually and automatically

	LATW 2003		Manually Optimized		Automatic 137		Automatic Free Poly	
	Enc	Dec	Enc	Dec	Enc	Dec	Enc	Dec
# 4-LUTs	215	538	140	402	134	392	129	370
Delay (ns)	14.5	47.6	13.9	30.5	13.9	30.5	13.8	30.4

The results show that a high reduction in area has occurred from the first to the second version, but the generator can achieve a more significant area reduction, and the fourth version, using the polynomial 131 had the best result from all of them. This means that the techniques presented can produce a better area result than manually implemented versions could produce, besides reducing the design time for future implementation versions.

6. Protecting Memories against Multiple Bit Upset (MBU)

An n-bit data memory can be protected against faults by using correcting code techniques based on encoder and decoder blocks and extra bits to store the data parities,

as represented in figure 4 [Neuberger 2003]. The encoder and the decoder can use any error detection and correction code. For example, in the case of using Hamming code, the memory can only support single bit upsets. The problem is that multiple upsets are likely to occur in current and in future technologies. In addition, upsets can accumulate in the memory if the correction process is not fast enough comparing to the upset flux.

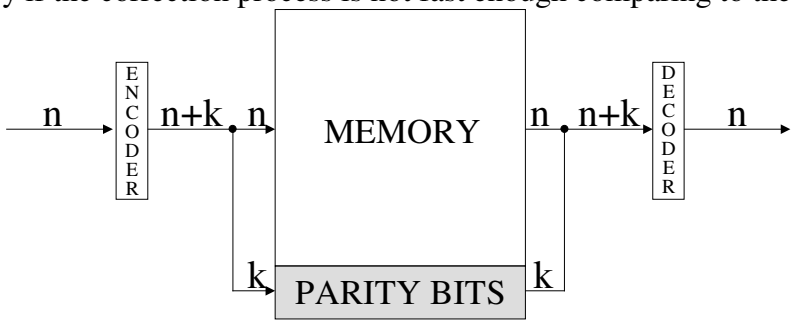

Figure 4. General schematic of a fault tolerant memory

To be able to correct multiple bit upsets, RS code must be used. The data word is divided in symbols, and each data word is a different RS coded word. For example, in a 256-rows memory, the data word uses the entire row, and each data word is divided in m symbols according to the symbol size and to the memory data size. Multiple upsets may occur in any portion of the matrix, but they are more likely to occur as double bit flips that are in the same symbol, in vertical adjacent symbols, or in horizontal adjacent symbols. The RS code can easily correct the first two types of errors, but the third one will not be corrected, because it is equivalent to errors in two different RS symbols in the same word, and the implemented RS is not capable to correct single errors in different symbols (blocks), only multiple errors in the same symbol.

A new code is needed to correct all possible double errors. The first option is the use of a Reed-Solomon code with capability to correct two different symbols. But this RS code has more than twice the area and delay overhead of the single symbol correction RS [Houghton 1997], which makes this solution inappropriate for memory architectures. The second alternative is the use of one bit protected by Hamming code between the RS symbols. The number of bits protected by Hamming will be the same of the number of symbols protected by Reed-Solomon, so this option does not significantly increases the area overhead. Figure 5 presents the insertion of Hamming code in row already coded by RS code.

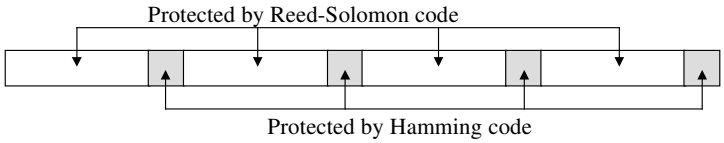

Figure 5. Schematic of a memory row protected by Reed-Solomon and Hamming

All single upsets are corrected by the code. Double bit upsets occurring horizontally or vertically are corrected by the code because each row is a different coded word and there is Hamming code protection in the RS symbols interface. Some multiple bit upsets can also be corrected. The only type of multiple upsets that can not be corrected by the method is when the hamming bit and two adjacent RS symbols are inverted. However, it is unlikely to occur in current technologies.

Once defined the codes to be used, the next step is to determine the specifications of the memory and the Reed-Solomon and Hamming codes sizes. The first memory study case is a 128-bit data memory, where a 128-bit code is needed. This number has been chosen based on the previously compared tradeoffs of RS code [Neuberger 2002]. A 7-bit symbol RS code was used to protect 112 bits of the data memory and a 16-bit Hamming code was used to protect the bits between each RS symbol. Although most memories do not have a word as large as 128 bits, it still has special interest in cache memories that are becoming larger in actual microprocessors.

The memory study case was described in VHDL and synthesized in a Virtex-E FPGA using embedded memories (BlockRAMs) and programmable complex logic blocks (CLBs). Results are presented in table 3. In the results, it is noticed that the fault tolerant memory has an area overhead that is basically the area used by the encoder and decoder blocks. Only two more BlockRAMs are needed, one to store the RS redundancy symbols and other to store the Hamming extra bits. The performance penalty in the fault tolerant memory synthesized in the FPGA is around 50%.

Table 3. Area and performance comparison between a no-protected 128-bit memory and a fault tolerant 128-bit memory based on RS and Hamming code.

	No protected Memory	Fault-tolerant Memory
# 4-LUTs	95	770
# BlockRAMs	16	18
Speed (MHz)	71	30

7. Conclusions

This work presented new techniques to increase fault tolerance to radiation in integrated circuits. The first technique present was duplication with comparison (DWC) with concurrent error detection (CED) based on time redundancy for the user's combinational logic in SRAM-based FPGAs. This technique reduces the number of input and output pins of the user's combinational logic. In addition, it can also reduce area when large combinational blocks are used. Also, a tool that generates efficient hardware implementations of the Reed-Solomon code was presented, and results show that it is a good alternative to the Hamming code. Finally, a SRAM memory that combines Reed-Solomon and Hamming code was presented, making the memory capable to tolerate all double bit upsets, that are likely to occur nowadays in latest technologies.

References

Azumi, S. and Kasami, T. (1975) "Of Optimal Modified Hamming Codes", In: Trans. Inst. Electr. Commun. Eng. Jap., vol. A58, no. 6, pp. 325-330.

Bessot, D. (1993) "Conception de Deux Points Memoire Statiques CMOS Durcis Contre L'effet des Aleas Logiques Provoques par L'environment Radiatif Spatial", These. INPG. November, 1993.

Buchner, S., Campbell, A., Meehan, T., Clark, K., McMorrow, D., Dyer, C., Sanderson, C., Comber, C. and Kuboyama, S. (2000) "Investigation of Single-Ion Multiple-Bit Upsets in Memories on Board a Space Experiment", In: Proceedings of IEEE Transactions on Nuclear Science, June 2000.

Carmichael, C., Fuller, E., Fabula, J. and Lima, F. (2001) "Proton Testing of SEU Mitigation Methods for the Virtex® FPGA", In: International Conference on Military and Aerospace Applications of Programmable Logic Devices, MAPLD, 2001.

Fuller, E. (2002) “Radiation test results of the Virtex FPGA and ZBT SRAM for Space Based Reconfigurable Computing”, In: International Conference on Military and Aerospace Applications of Programmable Logic Devices, MAPLD, 2002.

Hentschke, R., Marques, F., Lima, F., Carro, L., Susin, A. and Reis, R. (2002) “Analysing Area and Performance Penalty of Protecting Different Digital Modules with Hamming Code and Triple Modular Redundancy”, In: Symposium on Integrated Circuits and Systems Design (SBCCI), September.

Houghton, A. D. (1997) “The Engineer’s Error Coding Handbook”, London, Chapman & Hall.

IBM.(2000) “SOI Technology: IBM’s Next Advance in Chip Design”, In: http://www.ibm.com, January.

Johansson, K., Ohlsson, M., Olsson, N., Blomgren, J., and Renberg, P. (1999) “Neutron Induced Single-Word Multiple-bit Upset in SRAM”, In: IEEE Transactions on Nuclear Science, December 1999.

Label, K. (1999) “Commercial Microelectronics Technologies for Applications in the Satellite Radiation Environment”, In: http://flick.gsfc.nasa.gov/radhome.htm.

Lima, F., Carro, L. and Reis, R. (2003) “Techniques for reconfigurable logic applications: Designing fault tolerant systems into SRAM-based FPGAs”. In: Proceedings of the 40th conference on Design automation, June.

Lubaszewski, M. and Courtois, B. (1998) “A reliable fail-safe system”, In: IEEE Transactions on Computers, New York, v.47, n.2, p. 236-241, February.

NASA. (2002) “Radiation Effects on Digital Systems”, In: http://radhome.gsfc.nasa.gov/top.htm, January.

Neuberger, G., Lima, F. and Reis, R. (2002) “Designing a Reed-Solomon Core Optimized for Area and Performance”, In: Proceedings of XVII South Symposium on Microelectronics, June.

Neuberger, G., Lima, F., Carro, L. and Reis, R. (2003) “A Multiple Bit Upset Tolerant SRAM Memory”, In: Latin-American Test Workshop, February.

Neuberger, G., Kastensmidt, F. and Reis, R. (2004) “Improving the Use of Reed-Solomon Codes to Increase Fault-tolerance in Very Deep Sub-Micron Integrated Circuits”, In: Latin-American Test Workshop, March.

Patel, J. H., Fung, L. Y. (1982) “Concurrent Error Detection in ALUs by Recomputing with Shifted Operands”, IEEE Transactions on Computers, Vol. C-31, July.

Redinbo, G., Napolitano, L. and Andaleon, D. (1993) “Multibit Correcting Data Interface for Fault-Tolerant Systems”, In: IEEE Transactions on Computers, April.

Reed, R. (1997) “Heavy Ion and Proton Induced Single Event Multiple Upsets”, In: Proceedings of IEEE Nuclear and Space Radiation Effects Conference (NSREC), July 1997.

Rockett, L. (1992) “SEU Hardened Scaled CMOS SRAM Cell Design Using Gate Resistors”, In: IEEE Transactions on Nuclear Science, October.

Shirvani, P., Saxena, N. and McCluskey, E. (2000) “Software Implemented EDAC Protection Against SEUs”, In: IEEE Transactions on Reliability, September.

Whitaker, S., Canaris, J. and Liu, K. (1991) “SEU Hardened Memory Cells for CCSDS REED-Solomon Encoder”, In: IEEE Transactions on Nuclear Science, December.

Wrobel, F., Palau, J., Calvet, M., Bersillon, O., Duarte, H. (2001) “Simulation of Nucleon-Induced Nuclear Reactions in a Simplified SRAM Structure: Scaling Effects on SEU and MBU Cross Sections”, In: Proceedings of IEEE Transactions on Nuclear Science, December 2001.

Designing a Configurable Group Service with Agreement Components*

Fabíola Gonçalves Pereira Greve[1] **, Jean-Pierre Le Narzul**[2]

[1]Departamento de Ciência da Computação (DCC)
Universidade Federal da Bahia (UFBA)
Campus de Ondina, 40170-110 Bahia, Brasil

[2]GET/ENST Bretagne and IRISA - Adept
Rue de la Chataîgneraie - CS 17607
35576 Cesson-Sévigné Cedex, France

fabiola@ufba.br, JP.LeNarzul@enst-bretagne.fr

Abstract. *In the recent past years, many group toolkits, providing a support for the construction of reliable applications, have been designed. Even if the goals of their designers was similar, these toolkits differ in (i) the way the problems are tackled and (ii) the way the protocols are structured and set up in real systems. This paper presents the underlying design principles of a group system. It follows two innovative approaches which contribute to its flexible and configurable character. From an algorithmic point of view, the group primitives are implemented as instances of a generic consensus service. This choice leads to a great number of advantages: (a) the computation in the group is guaranteed as soon as a quorum of entities can communicate, (b) decoupling the group membership service from the communication service, (c) a better control of the dysfunctions in periods of strong network instability. From an architectural point of view, the elementary group protocols are regarded as autonomous agreement components, which can be combined freely for the implementation of other richer reliable services. This strategy differs from the classical ones in which the protocols are structured according to a fixed hierarchy of classes following a stack-based pattern of interaction.*

1. Introduction

For many years, the group paradigm has been recognized as a very useful abstraction for building distributed applications in various domains: cooperative work, teleconference, distributed games, etc. The group paradigm has proven also to be a natural candidate for designing fault-tolerant applications, above unreliable settings, by replicating a service in several objects located at different hosts. When used for replication, the group, represented by a collection of objects, is seen by the clients as a single logical entity; it is addressed transparently by the clients that are not aware of its composition.

Of course, the group composition can evolve along the time (members can join it, others can either leave it or crash) just as its state. The essential requirement for the survival of the group (as being a reliable object) is the common share of the same vision of events (changes on

*This work is partially supported by CNPQ/Brazil grant number 40.80.86/03-2.

composition, delivery of messages, crashes) by all its members; that requires the coordination of its actions by the means of agreement algorithms.

Given the importance of the group model, many efforts were carried out during this last decade to understand the problems related to the implementation of such a paradigm. These efforts led to various theoretical results and the development of many platforms. The set of works realized at the Cornell university – ISIS [1], HORUS[2], ENSEMBLE[3] and SPINGLASS[4] – are one of the most significant examples. It is important to notice that the statement of meaningful specifications, as well as the design and implementation of a group service are far from being commonplace [5, 6]. These difficulties are due mainly to the impossibility results regarding the agreement problems to which one is confronted during the implementation of the functionalities of the paradigm in asynchronous environments [7, 8].

One notices that various practical systems have the disadvantage of not specifying under which conditions the properties which they are supposed to implement are assured [5]. This is essential, considering the negative results regarding the resolution of agreement problems. It is thus necessary to clarify which is the behavior of the system when the assumptions established to guarantee its termination are not satisfied. The use of the consensus as building block for the construction of group services provides a concrete response to this need [9]. The precise characterization of the liveness and safety properties ensured by these solutions allows an exhaustive control of the behavior of the applications in the occurrence of network dysfunctions. Besides, theoretical solutions founded on the consensus support the construction of well structured protocols [10]. Our interest in this work is to benefit from the modular nature of the consensus based solutions to build extensible and adaptable systems.

The great majority of the group toolkits adopt a layer-based way of interaction between the elementary group protocols [2, 11]. If, on one hand, the structure imposed by a stack-based organization defines a way of developing an abstraction which is clear and easy to understand, on the other hand, the need to accomodate the functionality of the abstractions in layers that can only communicate with their adjacent layers imposes a very restrictive design model. Moreover, the flexibility of a stack-based architecture is rather limited and it does not allow for a real adaptation to the application needs or to the various qualities of the communication medium. Driven by these considerations, we address in this article a solution based on the use of the component technology. We believe that the properties of reusability, configurability and composability exhibited by this technology is of great interest for building a library of flexible reliable abstractions.

This paper is dedicated to a presentation of the underlying design principles of a component-based group system called EDEN [12]. The design is driven by the recent theoretical results regarding the realization of an agreement in an asynchronous environment. Moreover, it advocates the use of the component technology to struture the protocol classes. In such an approach, a reliable application will be designed as the composition of several independent components, connected via a communication platform and cooperating by well-defined interfaces. In EDEN, these components are part of the ADAM [13, 14] library and are structured using EVA [15], an event based architecture.

The paper is composed of five sections. Section 2 characterizes a group service. Section 3 presents briefly some of the group toolkits. Then, Section 4, the core of this work, discusses the many design alternatives proposed previously and compares them with the EDEN issues. Section 5 presents EDEN. Finally, Section 6 concludes the paper.

2. Group Service

A group is a collection of related processes, considered as a single logical entity. This paradigm covers two basic services, namely, the *group membership* service and the *group communication* service.

Group Membership. The group membership is in charge of providing, to the processes members, the current composition of the group. This is a major attribute of the state of the group (the other attributes of the state of the group are related to the application service or computation which the group is supposed to provide). The composition of the group evolves according to the will of the processes to join it or to leave it, and to the occurrence of processe crashes or communication channel failures. The current group composition is usually named a view.

The group membership problem has been introduced and solved for the first time by Cristian [16] in the context of synchronous distributed systems. The work on the membership problem in asynchronous systems has been pioneered by the Isis system [17]. Unfortunately, its specification was incomplete [5]. The asynchronous group membership problem was later proved to be impossible to solve without additional assumptions (on the detection of crashed processes) [8]. The existing difficulties to specify and solve this problem are directly related to the impossibility result regarding reliable detection of the failures in the asynchronous model [18, 7]. Since real failures combined with eventual wrong suspicions can carry out to a division of the group in sub-groups, two approaches for the management of the composition appeared: *primary partition systems* and *partitionable systems*.

Primary Partition x Partitionable Systems. A primary component membership service ensures that at any time the group is implemented by a single view. From a user perspective, it means that the membership service ensures the total ordering on the set of the views. A partitionable membership service allows different views of the same group to coexist (concurrent views). This means that the set of views perceived by the user is partially ordered. In that case, the processes of each view behave as if they were the only ones that are currently implementing the group. In each one of these components, the service (or at least a part of this one) must continue to be assured. Possibly, when the communication is restored, the group membership service carries out a mechanism of fusion of views in order to restore the group in a consistent state (taking into account the various states of the broken components).

Group Communication. The aim of a group communication service is to provide application processes with communication primitives that are well-suited to the group computing paradigm. The main primitive offered by this service is a *reliable multicast* facility allowing a process to send messages to all group members in an atomic manner. Usually, ordering guarantees (e.g., total order, causal order, fifo order) are associated with this multicast primitive [17]. This originates a number of other primitives. The fundamental ones for the implementation of a replicated service are (i) total order broadcast (also known as *atomic broadcast*), which ensures that all messages are delivered in the same order by all the group members and (ii) *view synchronous multicast* which aims at coordinating message delivery and installation of new views.

2.1. Group Protocols as Agreement Problems

Most of the reliable distributed abstractions of interest in group computing (atomic broadcast, membership management, view synchrony, leader election, ...) are agreement problems. With regard to all these problems, processes belonging to a same group have, from time to time, to

reach an unanimous decision. For example, in the atomic broadcast problem, due to the fact that messages are not received in the same order by all the processes, determining the delivery order can be viewed as an agreement problem. To construct this common global knowledge, all (non-crashed) members of the group have to repeatedly reach new agreements to unanimously order the new arrived messages.

The Consensus Problem. The agreement problems family can be characterized by a single abstraction, namely the *consensus* problem. Informally, this one can be defined in the following way. Each process proposes a value and all correct processes have to decide the same value, which has to be one of the proposed values. Consensus is the "greatest common denominator" among agreement problems and this is practically and theoretically very important. From a practical point of view, this means that any solution to consensus can be used as a building block on top of which solutions to particular agreement problems can be designed. From a theoretical point of view, this means that an agreement problem cannot be solved in systems where consensus cannot be solved.

Unfortunately, the consensus problem is actually impossible to solve in a deterministic way in asynchronous distributed systems when even a single process may crash [7]. Fortunately, to circumvent this negative result, several approaches have been investigated. One of them is based on the concept of unreliable failure detectors proposed by Chandra and Toueg [9]. The role of a failure detector service is to maintain a list of correct processes and suspected processes. Of course, due to the asynchronism of the system, it is impossible to write perfect failure detectors. But, if we consider that (i) any process that crashes is eventually suspected (called "completness property"), (ii) there is a time after which there is a correct process that is no longer suspected (called "accuracy property"), and (iii) a majority of processes within the group never fails, then it is possible to solve the consensus and some other agreement problems.

3. Group Toolkits

In a recent past (ten years), many group toolkits have been implemented. All of them allow to build reliable applications based on the group paradigm. We have classified these toolkits in families, which correspond more or less to the research laboratories where the projects were carried out.

- Isis [1], Horus [2], Ensemble [3] and Spinglass [4] from Cornell university. These systems have been used in the following toolkits, Orbix, Electra [19], AQuA [20] for a fault-tolerant Corba implementation.
- Totem [21] from the California university of Santa Barbara and Transis [22] from Hebrew university, Jerusalem.
- Phoenix [23] , Bast [11], OGS [24] from EPFL, Lausanne.
- Consul [25], Coyotte [26], Cactus [27] from Illinois university, Urbana.
- EDEN [12] and ADAM [28] developed conjointly by INRIA-IRISA, ENST Bretagne at Rennes and UFBA, Salvador.

The first group system generations (Isis, Totem, Transis, Consul) especially brought initial answers to the problems arising when reliability must be ensured in an asynchronous environment. Thereafter, the majority of them evolved to more modular and flexible versions (HORUS, Ensemble, Coyotte, GARF, BAST) providing full dependable issues (fault-torerance, security and real-time constraints). Further versions fulfill better the requirements of the modern applications: distributed objects support, deployment in large scale, adaptability, mobility (Spinglass, AQuA, ETERNAL, OGS, Cactus).

4. Design Choices

Even if the designers of the group systems originally had the same goal, their toolkits differ in a considerable manner (i) in the way problems are tackled and (ii) in the structuration of the protocols. In the rest of this section, we will describe some of the main characteristics of these systems in order to justify the design choices used in the development of EDEN. We are in particular interested by (i) their behavior in face of false suspicions and (ii) the way their protocols are structured.

4.1. How to React to False Suspicions

Previously, we discussed the impossibility of detecting with precision process crashes. In case of a scenario in which erroneous suspicions remain, two main approaches for the progression of computation in the group are distinguished. In a first strategy, named *progression by safe group*, the computation within the group requires the participation of all correct processes. In a second strategy, named *progression by long-lived group*, the evolution of computation requires only the participation of a subset of the correct processes (a quorum). Let us explain the characteristics and implications of each one of them.

Evolution by Safe Group The approach by safe group consists in keeping only correct processes inside the group. If some process is suspected to be faulty, it should be removed from the group view in order for the computation to take place. It means that the management of the communication depends on the management of the group membership and that it must trust the failure detector. To avoid execution blocking, the information generated by the latter is used to remove from the group view the processes suspected to be faulty, and this even if such a suspicion proves to be abusive. In order to avoid too frequent errors, the suspicion timeouts associated with the failure detectors must be well tuned.

To our knowledge, all the partitionable systems follow this strategy: Totem, Transis, Horus and their derivatives. In this context, the consequences of erroneous suspicions could be the bursting of the group, its partitioning in several minority components or in the worst case, the loss of the primary component. In the case of the Isis primary partition system, since the activity of the processes expelled from the group is stopped, this can carry out to a collective suicide.

Evolution by Long-Lived Group The approach by long-lived group is less conventional and more innovative. It consists in decoupling, as much as possible, the services provided by the group and authorizing, as soon as possible, the progression of computation even in the presence of failures.

The use of failure detectors in the design of agreement protocols enables such an approach. The adoption of the theoretical solutions founded on unreliable failure detectors authorizes the evolution of computation as soon as a majority of correct processes are able to communicate. Thus, as soon as the communication between the partitions is established (real or virtual partitions are they) and the contact between a quorum of processes is re-established, computation progresses.

One can thus benefit from the advantages related to the use of the consensus and the unreliable failure detectors as building blocks to authorize decisions even if the group is not in a safe state. Message ordering takes place even in the occurence of suspicions. View changes are carried out only with a majority of participants. This model allows the independent implementation of group protocols. The failure detectors timeouts can be adjusted in order (i) to speed the decisions up, which is appropriate for ordering messages or (ii) to delay them, which is appropriate for view changes so that a safe state is reached. This choice is application dependent. For example, in the passive replication style, view changes are imperative only when the coordinator crashes. By using

this approach, one avoids frequent or unsuitable changes. The communication within the group is finally completely uncoupled from the dynamic management of its group membership. The benefits are: effectiveness, flexibility and adaptability. The EDEN system follows this approach of long-lived group. To our knowledge, besides EDEN , only the family of systems developed in Lausanne (Phoenix, Bast, OGS) presents such a characteristic.

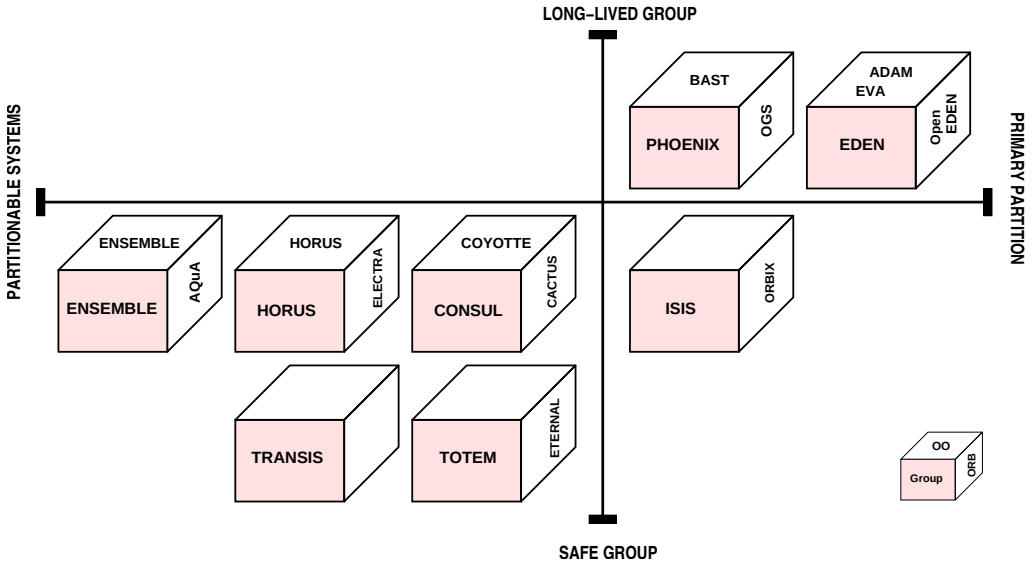

Figure 1: The group systems by type of evolutionary approach and group membership partition

Figure 1 classifies the families of group services according to the type of evolutionary approach (safe group or long-lived) and the type of group partition (primary or partitionable). In the figures which follow (this one included), the families are represented by cubes. Each face of the cube shows one of the elements of the family. The face group indicates the element which represents the core protocols, face OO indicates the element representing the object design criteria and face ORB represents the platform resulting from the integration of the group service in a distributed object architecture.

4.2. How to Structure the Protocols

We showed along this paper in what the theoretical solutions founded on the consensus support the construction of well structured and independent protocols. This approach also allows us to identify important abstractions (consensus, failure detectors, reliable broadcast, atomic broadcast, etc.) from which other reliable abstractions can be created (e.g. a replication service). Our interest now is to benefit from the modular nature of these solutions to build systems which are reusable, flexible, extensible and adaptable.

Protocol Classes. The object-oriented design and its advantages were appropriately used in the construction of Horus, Coyotte and Bast. Table 1 shows a summary of important principles introduced by these systems in the design of flexible reliable abstractions. In all these systems, the flexibility is reached by considering the “elementary reliable abstractions” as *protocol classes* from which other abstractions can be built.

Protocol Patterns. Horus and Coyotte primarily use the mechanism of inheritance to carry out protocol compositions. Bast goes further by introducing the concept of *protocol patterns* which are

domain-specific design patterns that describe how to compose protocol objects. Bast synthesizes much knowledge already established in the field of distributed systems and it identifies some important protocol patterns which are recurring in the modeling of fault tolerant applications. The `active replication` pattern and the `distributed agreement` pattern are good examples.

	Abstractions		
Group System	*Protocol Classes*	*Protocol Patterns*	*Protocol Components*
Horus	•		
Coyotte	•		
Bast	•	•	
Eden/Adam	•	•	•

Table 1: Strategies to structure protocols

Composable Protocol Stack. Horus introduces the concept of *composable protocol stack*, where each specific functionality (atomic broadcast, reliable broadcast, group membership, etc.) is implemented by a micro-protocol which can be placed above another to form a stack. This establishes a layer-based architecture in which each layer uses the services of the adjacent lower layer to provide an extended service to the adjacent upper layer, up to the last layer which provides the functionality required. The composition of the stack can be done at run time in a variety of ways to meet the exact application requirements. This way of "composition" has further been adopted by the great majority of group toolkits (Bast, Coyotte, OGS, etc.). Figure 2 shows the representation of such a strategy to implement a consensus-based replicated service.

Figure 2: Composable protocol stack

Layer Based Interaction. The Horus platform adopts a communication pattern in which an outgoing message goes through all the layers in a top-down manner, whereas an incoming message enters through the bottom layer and has to be pushed through the layers in a bottom-up way. Information is passed to lower layers using procedure calls, whereas information from lower layers is delivered to higher layers using call-backs which are installed dynamically by the higher layers when they start their execution. The same approach can be found in Bast, OGS and many other toolkits.

Event Based Interaction. Instead of using a linear pattern of communication, Coyotte represents the interactions between the protocol classes into a graph. In Coyotte the communication

between each micro-protocol is done in a dynamic way by the emission and the reception of events.

4.3. Component Oriented Design

As a layer-based architecture is frequently employed to specify communication protocols (OSI/ISO, TCP/IP, etc.), it could seem natural to keep on designing reliable protocols in the same way. However, if on one hand the layer-based design defines a simple discipline of development, on the other hand the need to accommodate the protocols functionality into layers that can only communicate with their adjacent layers imposes a restrictive model. Rather than being specified as a collection of well defined functionality layers, micro-protocols are normally presented as a description of the cooperating entities that together implement the protocol's functionality, and the interactions among them [29].

From a performance viewpoint, the layer-based style of communication yields penalties due (arbitrary delays can be introduced, the growth of the message size). This happens because information must cross all the stack to arrive at the destination. Of course, ad hoc optimizations (as for example, short cuts between non-consecutive layers) can allow alternative interaction paths and also enable a reduction on the protocol's latency [30]. Nevertheless, even with such improvements, a static stack of protocols is still not flexible enough.

To avoid the drawbacks of a layer-based architecture while keeping the advantages associated to the decomposition of the problem into well defined and independent basic services, we advocate the use of a *component oriented design*. In such an approach, the reliable service is designed as several independent components, connected via a communication platform and cooperating by well-defined interfaces. Thus, we propose to transform the "protocol classes" into *protocol components*. Instead of using a fixed hierarchy of classes to design a particular reliable abstraction, we propose to break this hierarchy into several independent entities, each one representing a reliable abstraction materialized by a component. Communication between them is not restricted to a stack, they undertake the shape of a graph. Such a flexibility allows components to be freely combined in order to create reliable services even more complex.

With such a strategy, the relationship between micro-protocols is no more made by implicit specializations. They are explicit and can be done in an asynchronous way. The functionalities of the reliable semantics can thus be implemented by a certain number of autonomous distributed objects interacting via the production and the consumption of events. In this way, events that are produced by suppliers must be routed by an event channel middleware to their corresponding consumers. Distributed algorithms are normally designed as reactive entities to the occurrence of events (messages, exception notifications, etc.). So, an event-based pattern of interaction is a simple and elegant way to fill in the gap between the specification of the protocol and its implementation.

ADAM is a library of agreement protocols, which is built according this component oriented design. In the implementation, the hierarchy of "protocol classes" (figure 2) – common in Horus, Bast and OGS – is completely broken in order to derivate the "protocol components". The EDEN group system is thus the result of the composition of some of the ADAM components.

Figure 3 represents the group systems according to their design strategy. We classify them in *composable protocols* (those designed by following OO concepts) and *non composable protocols*. Among composable protocols, we distinguish those which interact by layers and those which interact by events. Let us notice that, as in Coyotte, we propose to structure the protocols as a graph of entities. However, differently from it, we provide a library of components (ADAM) ready to be used by protocol developers in the implementation of reliable services. To our knowledge, Coy-

otte does not provide such a mechanism. As in BAST, we base ourselves on the `distributed agreement` pattern to capture the recurring structure of various classical agreement protocols.

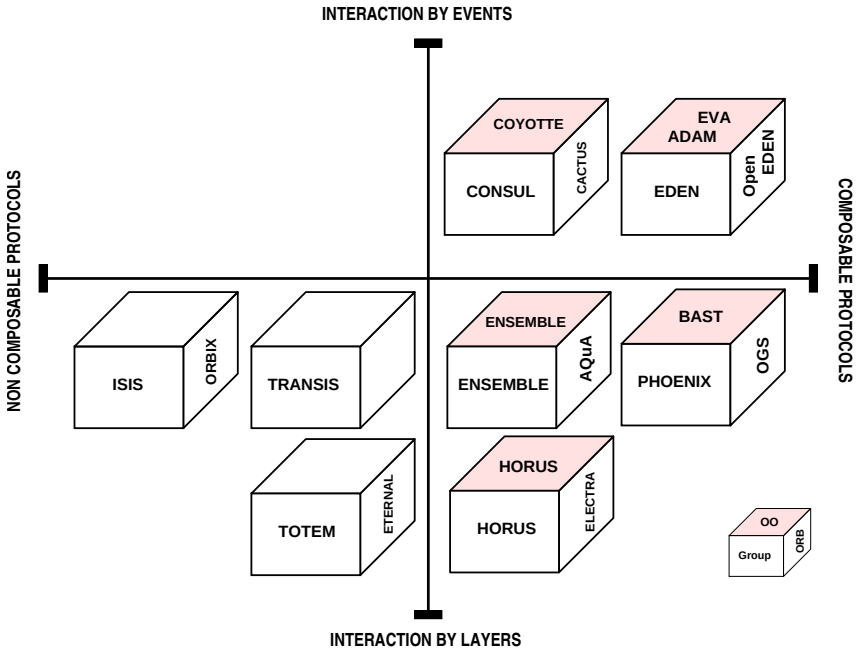

Figure 3: Group services and their design strategies

5. The EDEN Group Service

EDEN is a configurable group service based on a library of agreement components, called ADAM. ADAM [13, 14] is implemented above EVA [15], an event-based framework for developing distributed abstractions and high-level communication protocols.

ADAM is a component-based library of agreement abstractions, used to build reliable programming toolkits. The central element of the ADAM library is the GENERIC AGREEMENT COMPONENT (GAC) [29]. It implements a generic and adaptative fault-tolerant consensus algorithm that can be customized to cope with the characteristics of the environment. Moreover, thanks to a set of versatile methods, its behavior can be tuned to fit the exact needs of a specific agreement problem. A range of fundamental ADAM components are implemented as specializations of this GAC component. By composing some of the ADAM agreement components, we have built a group communication logic. An ACTIVE REPLICATION service has been designed and forms the heart of the EDEN system. It is mainly based on the ATOMIC BROADCAST, GROUP MEMBERSHIP and GENERIC AGREEMENT components. Figure 4 shows the EDEN components and their relationships.

The framework EVA implements a publish-subscriber communication environment to structure entities composing high-level protocols. In this architecture, protocols are regarded as a number of cooperating objects (entities) that communicate via an event-channel. Applications built on top of EVA are composed of a set of cooperating components that communicate with each other via the production and consumption of special types of objects called events. Further, a particular component is itself formed by cooperating entities (*e.g.* passive and active objects, sub-components, etc.) which also communicate via a similar event channel mechanism. Each component has an event channel manager that is responsible for routing the events produced by

its supplier entities to all of its consumer entities that have register to receive a notification for the particular event.

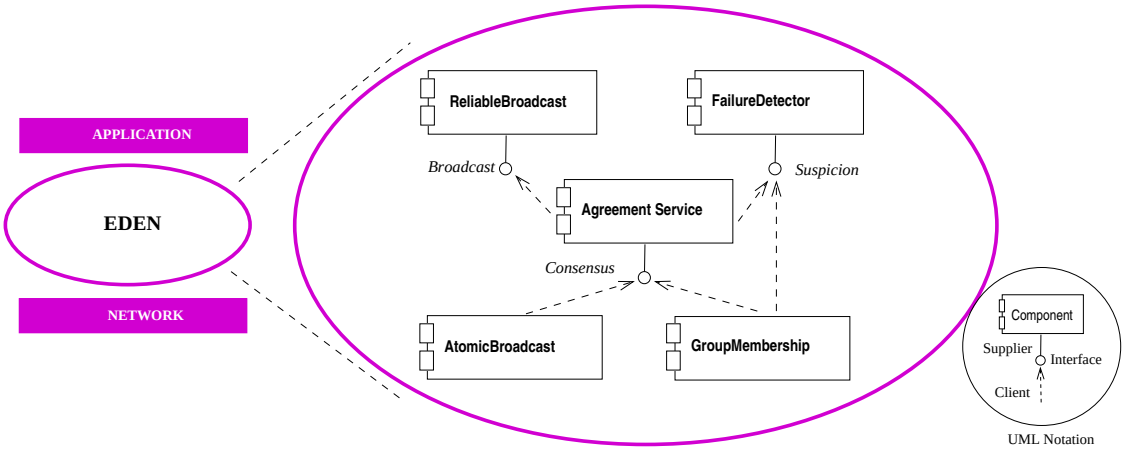

Figure 4: The EDEN Components

The ADAM library currently includes the most important components for a reliable distributed programming. Some of them are:

- GENERIC AGREEMENT COMPONENT (GAC). It lies at the core of the ADAM library. It implements a generic fault tolerant consensus algorithm [29]. To be operational, it has to be instantiated through the definition of a concrete agreement component (e.g. group membership, atomic broadcast). Indeed, it was designed to solve multiple agreement problems at the same time.
- GROUP MEMBERSHIP MANAGEMENT (GM) It is in charge of managing the changes on the group composition [31]. It will be composed with GAC in order to get the agreement on three sets of objects: (i) those which are authorized to *join* the group (ii) those which are authorized to *leave* the group (iii) those which are suspected to be faulty. The GM component interacts with a FAILURE DETECTOR component to take into consideration crashed members.
- ATOMIC BROADCAST (AC) It is implemented by a set of AC components that are in charge of receiving the requests broadcasted by external clients and to deliver them in the same order. For this it will interact and make use of the service provided by the GAC component.
- VIEW SYNCHRONY To install a new view, all the non-crashed members of the previous view must have delivered all the messages ordered during the previous view. The VIEW SYNCHRONY is in charge of ensuring this property. To achieve this goal, no additional agreement is necessary. Indeed, strong synchronizations have to be implemented to ensure that ATOMIC BROADCAST and GROUP MEMBERSHIP observe their decisions and progress in a consistent way. In practice, the installation of the new view is postponed till all the synchronization constraints are satisfied.

6. Conclusion

This paper was dedicated to present the design issues of the EDEN group system. It was conceived by following two innovative approaches which contribute to its flexible and adaptable character. From an algorithmic point of view, it is based in theoretical results that precisely characterizes the liveness and safety properties ensured by the protocols. This allows better control of the

behavior of the applications in the occurrence of dysfunctions. Moreover, it advocates the use of a component oriented design to implement each elementary reliable abstraction (atomic broadcast, group membership, etc.). In such an approach, a reliable application will be designed as the composition of several independent components, connected via a communication platform and cooperating by well-defined interfaces.

References

[1] K. Birman, *Reliable Distributed Computing with the ISIS Toolkit*, ch. Virtual Synchrony. Los Alamitos, CA: IEEE Computer Society Press, 1994.

[2] R. van Renesse, K. Birman, and S. Maffeis, "Horus: a flexible group communication system," *Communications of the ACM*, vol. 39, pp. 76–83, Apr. 1996.

[3] M. Hayden, *The Ensemble System*. PhD thesis, Cornell University, 1998.

[4] K. P. Birman, R. van Renesse, and W. Vogels, "Spinglass: Secure and scalable communications tools for mission-critical computing," in *International Survivability Conference and Exposition. DARPA DISCEX-2001*, (Anaheim, California), June 2001.

[5] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg, "On the formal specification of group membership services," Tech. Rep. TR95-1534, Depto of Computer Science, Cornell University, Aug. 1995.

[6] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study," *ACM Computing Surveys*, vol. 33, pp. 427–469, Dec. 2001.

[7] 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.

[8] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the impossibility of group membership," in *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, (New York, USA), pp. 322–330, ACM, May 1996.

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

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

[11] B. Garbinato, *Protocol Objects & Patterns for Structuring Reliable Distributed Systems*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998.

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

[13] F. Greve, , M. Hurfin, J.-P. L. Narzul, M. Xiaojung, and F. Tronel, "A library of agreement components for reliable distributed programming," in *Workshop on Communication Abstractions for Distributed Systems, with ACM ECOOP - 17th European Conference on Object-Oriented Programming*, (Darmstadt, Germany), 2003.

[14] F. Greve, , M. Hurfin, and J.-P. L. Narzul, "Adam: une bibliothèque de composants d'accord pour la programmation d'applications fiables," in *Actes des Journées Composants*, (Lille, France), Mar. 2004.

[15] F. Brasileiro, F. Greve, M. Hurfin, J.-P. L. Narzul, and F. Tronel, "Eva: an event based framework for developing specialised communication protocols," in *NCA 2001: IEEE International Symposium on Network Computing and Applications*, pp. 108–119, Feb. 2002.

[16] F. Cristian, "Reaching agreement on processor group membership in synchronous distributed systems," *Distributed Computing*, vol. 4, pp. 175–187, Apr. 1991.

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

[18] K. M. Chandy and J. Misra, "How processes learn," *Distributed Computing*, vol. 1, pp. 40–52, 1986.

[19] S. Maffeis, "Electra—making distributed programs object-oriented," in *Proc. of the Usenix Symp. on Experiences with Distributed and Multiprocessor Systems*, (San Diego, CA (USA)), pp. 143–156, 1993.

[20] Y. Ren, *AQUA: A Framework for Providing Adaptive Fault Tolerance to Distributed Applications*. PhD thesis, University of Illinois, Urbana, 2001.

[21] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos, "Totem: a fault-tolerant multicast group communication system," *Communications of the ACM*, vol. 39, pp. 54–63, Apr. 1996.

[22] D. Dolev and D. Malki, "The transis approach to high availability cluster communication," *Communications of the ACM*, vol. 39, pp. 64–70, Apr. 1996.

[23] C. Malloth, *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 1996.

[24] P. Felber, *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998.

[25] S. Mishra, L. Peterson, and R. Schlichting, "Consul: a communication substrate for fault-tolerant distributed programs," *Distributed Systems Engineering Journal*, vol. 1, no. 2, pp. 87–103, 1993.

[26] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu, "Coyote a system for constructing fine-grain configurable communication services," *ACM Transactions on Computer Systems*, vol. 16, Nov. 1998.

[27] M. A. Hiltunen and R. D. Schlichting, "The cactus approach to building configurable middleware services," in *Proc. of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Oct. 2000.

[28] F. Greve, , M. Hurfin, J.-P. L. Narzul, M. Xiaojung, and F. Tronel, "A library of agreement components for reliable distributed programming," in *Workshop on Communication Abstractions for Distributed Systems, with ACM ECOOP - 17th European Conference on Object-Oriented Programming*, (Darmstadt, Germany), 2003.

[29] 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.

[30] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable, "Building reliable, high-performance communication systems from components," in *Proc. of the* 17^{th} *ACM Symposium on Operating Systems Principles (SOSP'99)*, (Charleston, USA), pp. 80–92, Dec. 1999.

[31] F. Greve, M. Hurfin, M. Raynal, and F. Tronel, "Primary component asynchronous group membership as an instance of a generic agreement framework," in *ISADS'2001: 5th International Symposium on Autonomous Decentralized Systems*, pp. 93–100, Mar. 2001.

GroupPac 3: Estendendo o FT-CORBA para Gerenciamento e Replicação Ativa*

Alysson Neves Bessani[1†], Lau Cheuk Lung[2] ,
Eduardo Adílio Pelinson Alchieri[1‡], Joni da Silva Fraga[1 §]

[1]DAS - Departamento de Automação e Sistemas
UFSC - Universidade Federal de Santa Catarina
Campus Universitário, Caixa Postal 476 - CEP 88040-900 - Trindade - Florianópolis - SC

[2]PPGIA - Programa de Pós-Graduação em Informática Aplicada
PUC-PR - Pontifícia Universidade Católica do Paraná
R. Imaculada Conceição, 1155 - Prado Velho - CEP 80215-901 - Curitiba -PR

{neves,alchieri,fraga}@das.ufsc.br, lau@ppgia.pucpr.br

Resumo. *O FT-CORBA da OMG adota nas suas especificações quatro técnicas de replicação para tolerância a faltas de objetos: Sem Estado; Passiva Fria; Passiva Morna; e, por fim, a Replicação Ativa. Apesar da replicação ativa estar prevista no FT-CORBA, seus mecanismos não foram devidamente especificados devido à falta de definições para comunicação de grupo. Neste artigo, propomos um conjunto de estensões a estas especificações para melhorar o suporte da arquitetura à replicação ativa e ao gerenciamento de replicação.*

1. Introdução

O padrão FT-CORBA surgiu de um esforço da OMG (*Object Management Group*) no sentido de especificar uma arquitetura para objetos distribuídos tolerantes a falhas [Object Management Group, 2002]. Este padrão introduz suporte a tolerância a faltas na arquitetura CORBA fazendo uso de técnicas de replicação de objetos. Este suporte é construído a partir de um conjunto de objetos de serviço usados no gerenciamento de objetos de aplicação replicados. A tolerância a faltas fornecida usando os serviços definidos pelo padrão está fundamentada em premissas de falhas de parada (*crash*).

A especificação FT-CORBA define quatro tipos de replicação para os grupos de objetos: **Sem Estado** (estilo de replicação em que o estado dos objetos não é alterado pelas requisições); **Passiva Fria** (replicação com apenas um objeto ou réplica primária que grava periodicamente informações de estado em meio persistente); **Passiva Morna** (funciona de maneira semelhante ao anterior com a diferença de que os *checkpoints* da réplica primária são difundidos entre as réplicas secundárias); e, por fim, a **Replicação Ativa** (onde todas as réplicas executam as requisições do cliente).

O único tipo de replicação que é previsto e não tem seus mecanismos necessários especificados é a replicação ativa. A especificação atual do FT-CORBA permite que as abstrações definidas para a gestão de replicações possam ser também usadas com réplicas ativas, desde que suportadas em suas necessidades de comunicação por ferramentas proprietárias[1]. Assim, as requisições são difundidas, através de difusão atômica

*Realizado com recursos do CNPq (projeto número 401802/2003-5).
†Bolsista PGI/CNPq.
‡Bolsista PIBIC/CNPq.
§Bolsista de Produtividade em Pesquisa do CNPq - Nível 2.
[1]Proprietárias no sentido de não serem baseadas em padrões abertos.

[Hadzilacos and Toueg, 1994, Défago et al., 2000], a todos os objetos membros do grupo (réplicas ativas) usando meios externos ao ORB. Neste caso, se considerarmos as necessidades de suporte de uma replicação ativa, do ponto de vista conceitual, diríamos que o FT-CORBA fornece os mecanismos para controle de *membership* do grupo formado pelas réplicas ativas e a ferramenta proprietária fornece o suporte para comunicação de grupo. Alguns pontos que a especificação da OMG não deixa claro são como integrar essa ferramenta proprietária na arquitetura CORBA e qual a arquitetura de implantação dos serviços definidos, entre outros.

Este trabalho apresenta um conjunto de estensões propostas à arquitetura FT-CORBA para um melhor suporte a replicação ativa e gerenciamento de grupos de réplicas. São ao todo três novos mecanismos que visam melhorar a arquitetura de tolerância a faltas definida para o padrão CORBA, sem, no entanto, ferir qualquer aspecto já definido nas especificações atuais. O primeiro mecanismo a ser apresentado é a definição, gravação e recuperação de grupos em arquivos XML [Bray et al., 2004]. Este mecanismo visa definir um modelo para a representação de grupos de réplicas FT-CORBA e dar suporte a gravação do estado de um grupo (propriedades e estados das réplicas) para posterior recuperação. O segundo mecanismo definido, chamado célula de tolerância a faltas, é um pequeno servidor que visa dar suporte às réplicas. Grande parte dos serviços da especificação atual são ativados dentro deste servidor. A última (e mais importante) contribuição deste trabalho é um *framework* para a integração de ferramentas de comunicação de grupo na arquitetura FT-CORBA visando dar suporte à replicação ativa. Através deste *framework* é possível o uso de qualquer sistema de comunicação de grupo na arquitetura FT-CORBA.

As estensões propostas foram implementadas no sistema GROUPPAC [Lung et al., 2001]. Este sistema é uma implementação completa do padrão FT-CORBA desenvolvida pelo grupo de sistemas distribuídos do LCMI/DAS/UFSC. Inicialmente desenvolvido para dar suporte a sistemas de larga escala, o GROUPPAC foi uma das primeiras implementações do padrão FT-CORBA desenvolvidas, e ainda hoje é uma das poucas disponibilizadas livremente (*http://grouppac.sf.net*).

O texto está organizado da seguinte forma: na seção 2 temos uma breve descrição dos principais aspectos das especificações FT-CORBA. A seção 3 apresenta nossas estensões para gerenciamento de grupos, seguida pela seção 4, onde a célula de tolerância a faltas tem seu conceito e arquitetura apresentados. Na seção 5 é apresentado detalhadamente o *framework* de integração de sistemas de comunicação de grupo ao FT-CORBA. Nesta seção são descritos ainda os *plugins* de integração disponíveis atualmente para o GROUPPAC. Finalmente, a seção 6 relata alguns experimentos similares da literatura e a seção 7 apresenta as conclusões do trabalho.

2. As Especificações FT-CORBA

A arquitetura **FT-CORBA** [Object Management Group, 2002] define os serviços, em nível de *middleware*, responsáveis por prover as funcionalidades básicas para aplicações tolerantes a faltas através da replicação de objetos CORBA. Esses serviços estão divididos em três módulos básicos:

- **Gerenciamento de Replicação (SGR):** Este é o serviço responsável pelo ciclo de vida dos grupos. Duas funcionalidades são oferecidas no mesmo: Gerenciamento de Propriedades, onde as características funcionais da replicação (tipo de replicação usada, número mínimo de réplicas, etc.) são definidas, e o Gerenciamento de Grupos, que oferece mecanismos para a criação de membros (através de Fábricas de Objetos) e para o controle de *membership* dos grupos definidos;

- **Gerenciamento de Falhas (SGF):** É o serviço responsável pela detecção, notificação, análise e diagnóstico de falhas. Este serviço trabalha em conjunto com o SGR para que este último mantenha um *membership* sempre atualizado dos grupos;
- **Gerenciamento de Recuperação e Logging (SRL):** O SRL é responsável pela consistência de estados das réplicas. Este serviço define mecanismos para a recuperação de réplicas e do próprio grupo. Entre estes mecanismos está um suporte para construção de *logs* usados no armazenamento de requisições enviadas ao grupo. Este serviço também fornece mecanismos para atualização de membros através de *checkpoints*. Além disso também é responsabilidade do SRL atuar na recuperação de réplicas faltosas através da atualização de seus estados.

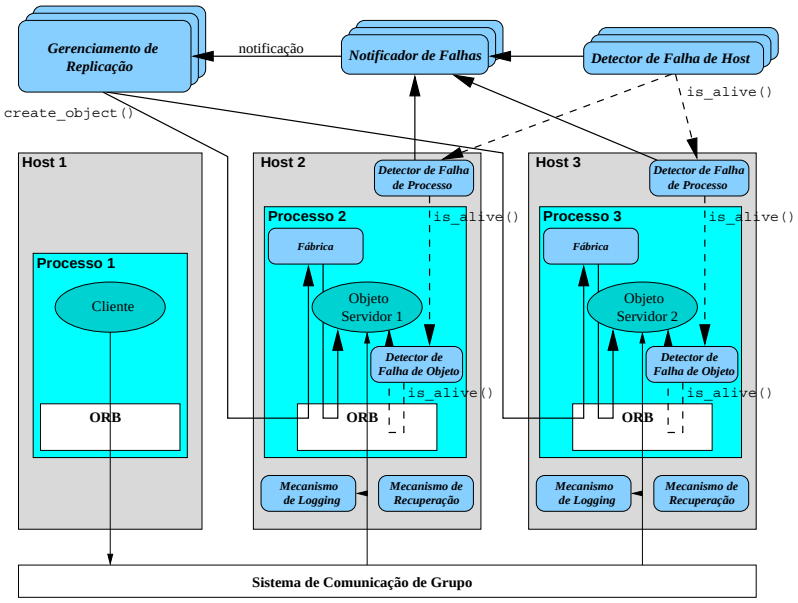

Figura 1: Arquitetura FT-CORBA.

Na figura 1, que ilustra a arquitetura FT-CORBA, não é explicitado suporte dos serviços FT-CORBA no *host* 1. Isto evidencia que, para o cliente, a replicação do servidor de aplicação é completamente transparente, a não ser pelo tratamento da referência de grupo (ver adiante). A criação de membros do grupo é feita através do gerenciador de replicação (SGR), que delega esta funcionalidade chamando, no *host* correspondente, o método remoto `create_object()` do objeto fábrica que é o responsável local pela criação e ativação de novos membros do grupo. Ainda nesta figura, os *hosts* 2 e 3 contêm membros do grupo de réplicas criados a partir das fábricas ativas nos mesmos.

A detecção de falhas no FT-CORBA (SGF) segue uma hierarquia, de acordo com três níveis de abstrações: objeto, processo e *host*. Em cada processo que mantém membros de um grupo existe um detector de falhas de objetos. Este detector verifica periodicamente, através da chamada de método `is_alive()`, se os objetos em seu processo ainda estão em atividade. Para nível de *host* é definido o detector de falhas de processos que faz o monitoramento dos processos do *host*. Por fim, em nível de sistema, completando a hierarquia, ocorre a detecção de falhas em *hosts*. Os detectores de falhas de *hosts* enviam, periodicamente, requisições aos detectores de falhas de processos (ativando também o método `is_alive()` destes últimos), para verificar se os mesmos e, por conseqüência, seus *hosts* ainda estão ativos.

Quando qualquer um dos detectores presentes no sistema observam uma parada em um objeto, processo ou *host*, esta é sinalizada ao notificador de falhas (ver figura 1), que a repassa ao SGR para que este atualize o *membership* do grupo. Se a falha ocorre no

membro primário de uma replicação passiva, cabe ao mecanismo de recuperação escolher um novo primário do grupo de objetos e atualizar seu estado de tal forma que este possa continuar processando requisições do ponto onde o objeto primário faltoso parou. A fim de obter uma melhor confiabilidade nos sistemas que usam suporte FT-CORBA, os objetos de serviço (gerenciador de replicação, notificador de falhas e detector de falhas de *host*) devem ser replicados.

Conforme já citado, a especificação FT-CORBA não define um suporte de comunicação de grupo, indispensável na replicação ativa: não existem interfaces e nem tampouco um protocolo padronizado para tal. As especificações apenas sugerem que se utilize qualquer suporte proprietário para este fim. Na figura 1, este suporte de comunicação aparece como camada subjacente, abaixo da camada de *middleware* e, portanto, dos objetos de serviço FT-CORBA.

Em relação à interoperabilidade com grupos onde objetos são mantidos por diferentes ORBs que implementam o FT-CORBA, a OMG criou um formato padronizado para a referência de grupo de objetos: IOGR (*Interoperable Object Group Reference*). A referência IOGR consiste basicamente em um conjunto de perfis IIOP (concretização do GIOP sobre TCP/IP) que identificam cada membro do grupo sendo o membro primário definido a partir da presença da *tag* `TAG_PRIMARY` em seu perfil.

3. Gerenciamento: Definição e Persistência de Grupos em XML

O gerenciador de replicação definido nas especificações FT-CORBA tem um componente para o gerenciamento de propriedades dos grupos. Estas propriedades são sempre definidas durante a criação de um grupo, e algumas delas podem ser alteradas durante o ciclo de vida do mesmo. A gestão destas propriedades é feita através da interface `PropertyManager`, e serve como um bom ponto de entrada para mecanismos de gerência mais amigáveis como interfaces gráficas e programas que carregam arquivos que definem grupos.

O GROUPPAC têm uma ferramenta de administração que serve para a criação de grupos de réplicas e sua gestão. A interface desta ferramenta é apresentada na figura 2.

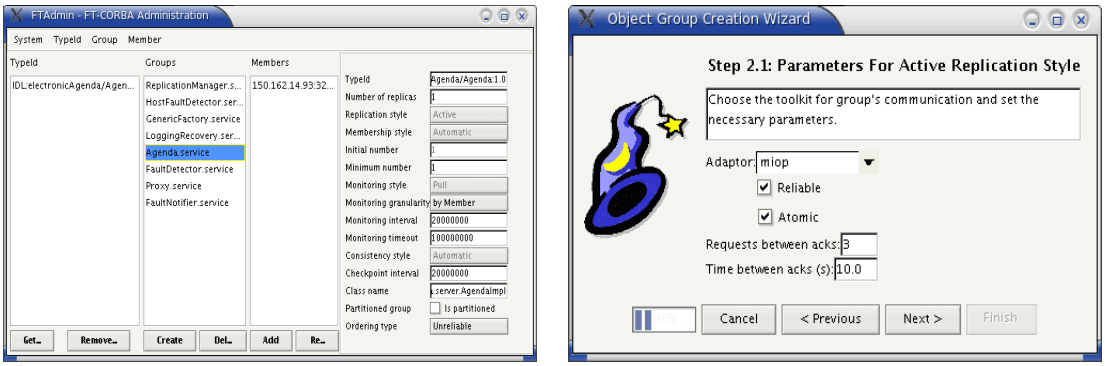

Figura 2: Ferramenta de administração e *wizard* de criação de grupo.

A criação de grupos via esta ferramenta é feita passo a passo através do preenchimento de um conjunto de formulários, o que nem sempre é agil. Portanto definiu-se um mecanismo para a especificação de grupos em XML que, além de permitir a criação de grupos dos mais diferentes tipos (permitindo a especificação de todas as propriedades definidas pela OMG mais algumas estensões), permita a gravação de um grupo em arquivo para sua posterior recuperação. Nesta gravação temos o armazenamento dos membros (localização, estado e se é o primário) e das propriedades do grupo.

O mecanismo de gerência de grupos implementado é definido em três módulos básicos:

- **XML Schema:** Um XML Schema [Fallside, 2001] define um conjunto de regras para a definição de um documento XML. Estas regras descrevem a estrutura esperada do documento, sendo portanto semelhante a uma gramática. Em nossa implementação optou-se pela utilização de XML Schema ao invés de DTD (*Document Type Definition*) [Bray et al., 2004] devido ao maior poder de expressão do primeiro;
- **Interpretador:** Este módulo define duas classes usadas para a leitura, validação e interpretação do arquivo XML que descreve um grupo. Uma vez as informações interpretadas, a criação do grupo é feita através da interação com o gerenciador de replicação;
- **Capturador de Estado:** Este componente é responsável pela captura do estado do sistema (número de membros, suas localizações e seus estados). Por ser executado apenas em tarefas administrativas, espera-se que a aplicação seja parada antes de se realizar uma gravação do estado do grupo.

A figura 3 apresenta o XML para a criação de um grupo.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <ft-service name="Agenda" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4                           xsi:noNamespaceSchemaLocation="ft-schema.xsd">
5
6    <description>Simple replicated service for ft-corba</description>
7
8    <type-id>IDL:electronicAgenda/Agenda:1.0</type-id>
9
10   <replication-style type="Active">
11      <property name="adaptor-factory"
12                value="br.ufsc.das.grouppac.scg.impl.umiop.UMIOPAdaptorsFactory"/>
13      <property name="reliable" value="true"/>
14      <property name="atomic" value="true"/>
15      <property name="n" value="3"/>
16      <property name="t" value="10"/>
17   </replication-style>
18
19   <membership-style type="Automatic">
20     <initial-number>1</initial-number>
21      <minimum-number>1</minimum-number>
22      <class-name>electronicAgenda.server.AgendaImpl</class-name>
23   </membership-style>
24
25   <monitoring-style type="Pull">
26      <granularity>by member</granularity>
27      <interval>2</interval>
28      <timeout>10</timeout>
29   </monitoring-style>
30
31   <consistency-style type="Automatic">
32      <interval>2</interval>
33   </consistency-style>
34</ft-service>
```

Figura 3: Especificação em XML de um grupo de réplicas para o FT-CORBA.

Nesta figura temos alguns pontos que devem ser destacados. Em primeiro lugar, o grupo é descrito por um elemento raiz `ft-service` identificado por um nome (neste caso “Agenda”). Dentro deste elemento temos vários outros elementos importantes, como por exemplo o `replication-style` (linhas 10-17). Este elemento define o tipo de replicação (no exemplo é ativa) e pode conter propriedades a serem usadas como critério na criação do grupo. Em nosso exemplo, definimos a fábrica de adaptadores usada pelo SCG (ver seção 5) e uma série de propriedades a serem passadas para este adaptador.

Os demais elementos da figura 3 dizem respeito a outras propriedades definidas no padrão FT-CORBA, apresentadas no capítulo 23 das especificações [Object Management Group, 2002].

4. CTF: Célula de Tolerância a Faltas

A fim de agrupar todos os elementos necessários para a ativação de um objeto tolerante a faltas foi criado um pequeno servidor capaz de oferecer todo o suporte para uma réplica funcionar corretamente. Este servidor, chamado **célula de tolerância a faltas** (CTF), é uma espécie de *container* onde as réplicas convivem. A figura 4 apresenta sua arquitetura.

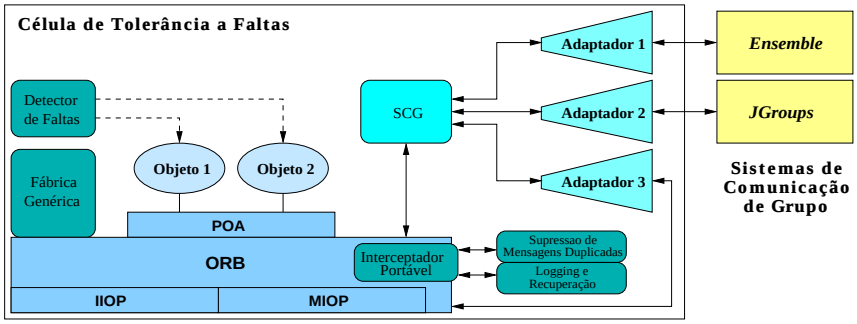

Figura 4: Arquitetura da Célula de Tolerância a Faltas.

Nesta figura estão representados os vários elementos necessários em um processo que mantenha réplicas de grupos. O elemento central da CTF é o ORB, e sobre ele são ativados todos os demais elementos. Cada célula CTF suporta no máximo uma réplica de cada tipo[2]. Estas réplicas são sempre criadas pela fábrica genérica, que utiliza as capacidades de reflexão computacional do Java para a execução desta tarefa. Uma vez criadas, as réplicas são monitoradas pelo detector de falhas (de objeto) que verifica o funcionamento das mesmas, reportando erros ao notificador de faltas.

A comunicação com a réplica pode se dar de duas formas, dependendo do tipo de replicação usado no grupo. Se usado o modelo de replicação ativa ou ativa com votação, a comunicação é feita através do *framework* SCG, que permite a integração da arquitetura FT-CORBA a sistemas de comunicação de grupo[3] que possam prover serviços de difusão atômica [Hadzilacos and Toueg, 1994], um requisito básico para a replicação ativa [Schneider, 1990]. Se o estilo de replicação do grupo for sem estado, passiva ou passiva morna, as mensagens são trocadas via IIOP, usando apenas mecanismos padrão do ORB. Note-se que em todos os tipos de replicação a mensagem passa pelo interceptador portável, que executa duas tarefas: *logging* das mensagens e supressão de mensagens duplicadas, evitando que um objeto execute a mesma requisição duas vezes.

Vale ressaltar que nem todos os elementos apresentados na figura 4 são ativados de uma única vez. Quando o SCG é necessário (replicação ativa e ativa com votação) os serviços de *logging* e recuperação não são (já que eles só são úteis em replicação passiva), e vice versa.

[2]O tipo corresponde ao tipo da interface, definido em IDL, chamado nas especificações de *type id*.

[3]Podem ser construídos adaptadores que se baseiam apenas em serviços de comunicação do ORB, como o apresentado em [Bessani et al., 2004]

5. SCG: Integração de Ferramentas de Comunicação de Grupo ao FT-CORBA

O modelo de replicação ativa, proposto para o FT-CORBA, baseia-se na idéia de um grupo fechado de réplicas acessado por clientes leves através do mecanismo de comunicação usual do CORBA (chamada de métodos remotos via IIOP). A especificação FT-CORBA também prevê a utilização de ferramentas de comunicação de grupo proprietárias para difusão de mensagens com diferentes níveis de qualidade de serviço [Object Management Group, 2002]. Entretanto, estas mesmas especificações não definem como estas ferramentas de comunicação de grupo serão integradas na arquitetura FT-CORBA.

Diante disto foi criado o *framework* de **suporte de comunicação de grupo** (SCG). O SCG é parte integrante do GROUPPAC e define um mecanismo padrão através do qual a infra-estrutura FT-CORBA interage com diferentes ferramentas de comunicação de grupo visando prover serviços de comunicação com as propriedades necessárias para a replicação ativa. Através deste mecanismo é possível a integração de qualquer suporte de comunicação de grupo à arquitetura do FT-CORBA através da construção de adaptadores que são integrados ao GROUPPAC através de *plugins*. A figura 5 apresenta este modelo.

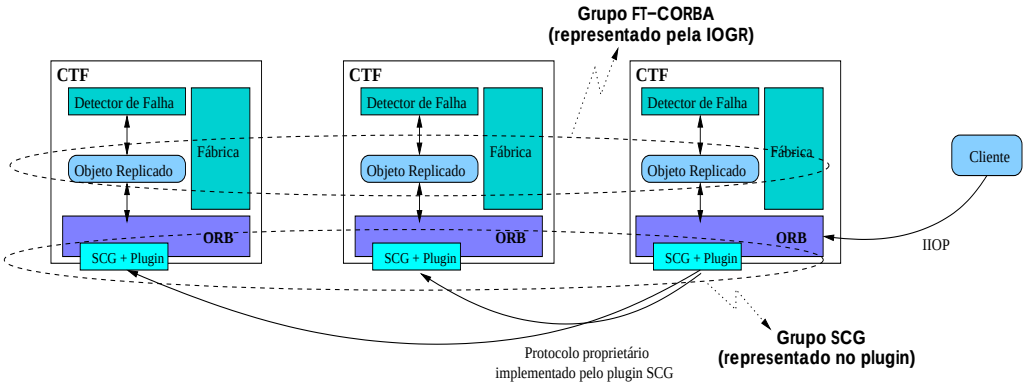

Figura 5: Replicação Ativa no GROUPPAC.

Nesta figura, o cliente, de posse da referência de grupo (IOGR), envia uma requisição ponto a ponto (através do IIOP) a um dos objetos listados na IOGR. O receptor desta requisição serve de ponte para acesso aos protocolos de comunicação de grupo (encapsulados nos *plugins* e ferramentas de comunicação de grupo). Estes protocolos são executados atendendo ao requisito de que todas as réplicas do serviço executem a mesma sequência de requisições [Schneider, 1990]. As respostas, se houverem, são enviadas de volta à "ponte", que retorna o resultado ao cliente[4]. Portanto, é através destes elementos "ponte" e de *plugins* que o grupo fechado definido pelo FT-CORBA fica acessível a clientes IIOP. Caso a réplica "ponte" falhe em algum ponto desse processamento, um *timeout* ocorre no cliente que reenvia a requisição a outro objeto cuja referência está contida na IOGR.

O SCG foi concebido através da instanciação de vários padrões de projeto [Gamma et al., 1995] utilizados em três componentes:

- ***Proxy* de Requisições:** Este componente recebe todas as requisições enviadas ao grupo as repassa ao *plugin* de comunicação de grupo. Sua implementação é baseada no conceito de servidores genéricos, utilizando os mecanismos

[4]Se o tipo de replicação for ativa, a primeira resposta devolvida por uma réplica é retornada ao cliente. Já no caso da replicação ativa com votação, todas as respostas de réplicas não faltosas são recebidas para então o resultado ser computado e devolvido.

de DSI (*Dynamic Skeleton Interface*), previsto nas especificações CORBA [Object Management Group, 2002];

- **Invocador de Requisições:** Este é o componente responsável por passar as requisições recebidas do *plugin* de comunicação de grupo para a réplica do grupo invocado. A implementação deste componente emula um *stub* para a invocação dos métodos da réplica sem o *overhead* do ORB;
- **Suporte ao *Plugin* de Comunicação de Grupo:** Este componente compreende as classes e interfaces de suporte a *plugins* de ferramentas de comunicação de grupo. Basicamente, temos um mecanismo para a carga dinâmica de *plugins* e algumas interfaces usadas como base na implementação destes.

A figura 6 apresenta um pequeno diagrama em UML com as diversas classes e interfaces que compõem o *framework*. Nesta figura estão representadas as três interfaces que devem ser implementadas para a concretização de um *plugin* SCG: `AdaptorsFactory` (criação de adaptadores), `SenderAdaptor` (adaptador *proxy*/*plugin*) e `ReceiverAdaptor` (adaptador *plugin*/invocador).

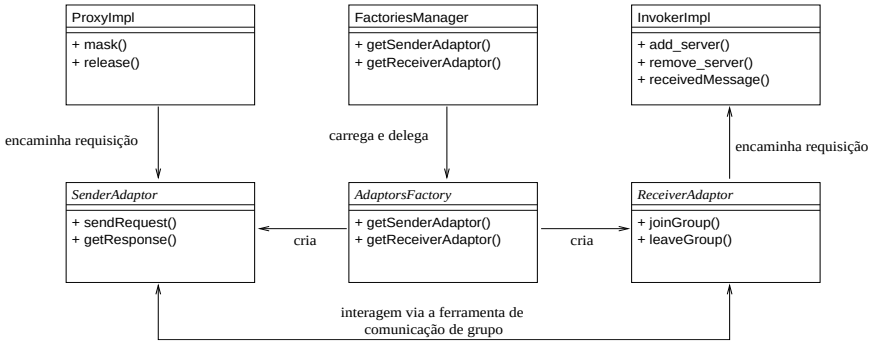

Figura 6: *Framework* SCG.

O grande "truque" empregado no *framework* SCG para conseguir a utilização da ferramenta de comunicação de grupo é a troca das referências dos membros presentes na IOGR por referências a *proxies* para o grupo[5]. A figura 7 apresenta um diagrama de seqüência em UML que apresenta como ocorre a criação de um grupo para replicação ativa.

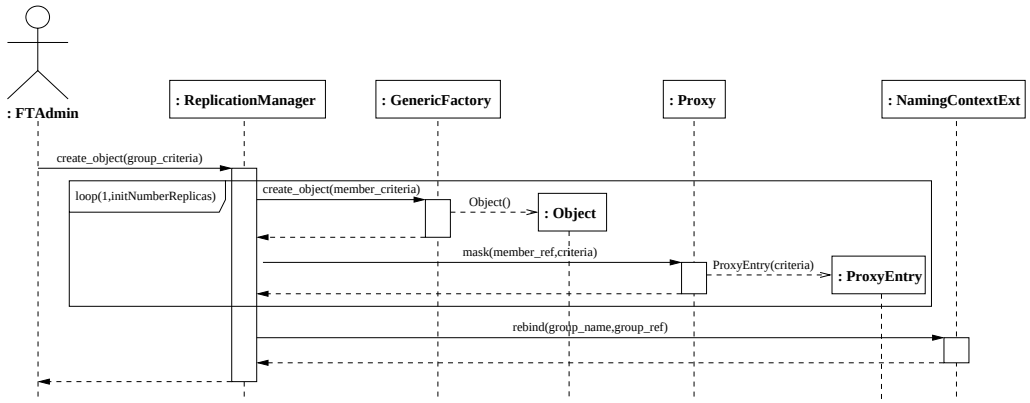

Figura 7: Seqüência de operações para a criação de um grupo.

Nesta figura temos um administrador criando um grupo passando como parâmetro uma série de critérios para o gerenciador de replicação (classe `ReplicationManager`). Este, por sua vez, executa, para cada réplica a ser criada, uma chamada à `GenericFactory` que cria a réplica (observando os critérios de criação) em sua localização

[5] As especificações permitem este tipo de artifício [Object Management Group, 2002].

e uma chamada ao `Proxy` para que este crie uma entrada (`ProxyEntry`) que possa mascarar esta réplica[6]. Dai em diante o gerenciador monta a IOGR de grupo (contendo as referências para os *proxies* criados) e a disponibiliza no serviço de nomes tolerante a faltas. O processamento se encerra com a IOGR sendo devolvida ao administrador.

Uma vez que o cliente possui a IOGR, este pode realizar invocação de operações nas réplicas através dos *proxies*. Conforme já apresentado na figura 5, esta invocação é feita em dois passos de comunicação: cliente-*proxy* via IIOP e *proxy*-grupos de réplicas via o sistema de comunicação de grupo. O diagrama de seqüência da figura 8 apresenta esse processamento.

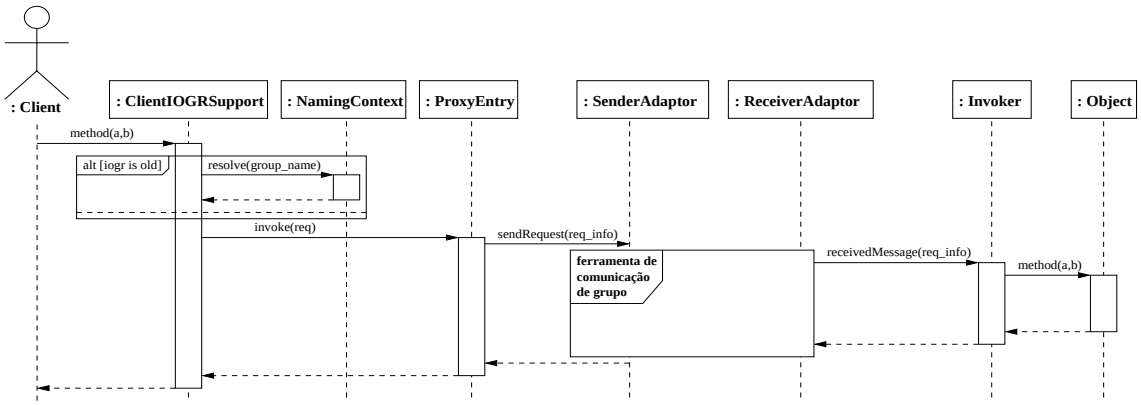

Figura 8: Seqüência de operações para a execução de uma operação.

Na figura 8 um cliente invoca a operação `method(a,b)` em um grupo que implementa a IDL onde a mesma está definida. O primeiro passo a ser executado é a verificação por parte do interceptador portável `ClientIOGRSupport` instalado no cliente. Se a IOGR usada para a invocação não é a versão mais atual para esse grupo o cliente faz esta atualização através do serviço de nomes. A seguir, uma das referências presentes na IOGR é escolhida, e a requisição é enviada ao `ProxyEntry` correspondente (que mascara uma das réplicas). O `ProxyEntry` é um servidor dinâmico que pode atender a qualquer tipo de método (via DSI). Após receber a requisição ele verifica qual ferramenta de comunicação de grupo deve ser usada para esta IOGR e usa o adaptador cliente apropriado (classe que implementa `SenderAdaptor`) para invocar tal ferramenta. A ferramenta difunde a requisição invocada usando um protocolo de difusão atômica implementado pela ferramenta de comunicação de grupo. Esta difusão acaba por ser recebida (na mesma ordem) por todos os adaptadores receptores (implementação da interface `ReceiverAdaptor`). Estes a entregam ao `Invoker` que realiza a invocação do método na réplica. A resposta à requisição (se houver) segue pelo caminho inverso, conforme mostrado na figura.

5.1. *Plugins* Disponíveis para o SCG

Atualmente estão implementados *plugins* SCG para as ferramentas ENSEMBLE [van Renesse et al., 1998] e JGROUPS [JGroups, 2004] e para o protocolo baseado nos serviços de comunicação do ORB MJACO [Bessani et al., 2002] apresentado em [Bessani et al., 2004]. Esta subseção apresenta uma breve descrição destes *plugins* e seus adaptadores.

Os *plugins* ENSEMBLE e JGROUPS são muito parecidos entre si. Eles implementam apenas uma "casca" bastante simples que transforma requisições CORBA em men-

[6]Por questões de simplicidade, o diagrama apresenta apenas uma `GenericFactory` e um `Proxy` que são chamados uma vez para cada réplica criada. Na verdade, estas chamadas ocorrem em fábricas e *proxies* diferentes que executam em diferentes localizações.

sagens e as envia através da API desses sistemas de comunicação de grupo (adaptador emissor). A recepção destas mensagens é implementada através dos adaptadores receptores e classes auxiliares que implementam interfaces de *callback*[7]. Tanto o ENSEMBLE quanto o JGROUPS trabalham com microprotocolos que são agrupados em pilhas para oferecer a funcionalidade desejada, que, no nosso caso, é a replicação ativa.

Já o *plugin* MJACO utiliza os serviços de comunicação providos por este ORB para a construção de um protocolo de difusão confiável com ordem total. Este protocolo explora o modelo de objetos definido no padrão UMIOP [Object Management Group, 2001], que permite o acesso a vários objetos a partir de uma única referência de grupo via *multicast* IP e o serviço de *membership* do FT-CORBA, dando suporte assim a um serviço de replicação ativa inteiramente baseado em padrões da OMG.

A idéia básica para a construção deste *plugin* foi refletir o grupo FT-CORBA, representado pela IOGR, em um grupo UMIOP, usado no contexto do SCG, através de uma referência de grupo UMIOP (para difusão não confiável). Para tanto foi definido um algoritmo que implementa difusão com ordem total baseado nas premissas do FT-CORBA: comunicação ponto a ponto confiável (pilha IIOP/TCP/IP), comunicação multiponto não confiável (pilha MIOP/UDP/*multicast* IP) e serviço de *membership* com propriedades fortes como as definidas em [Ricciardi and Birman, 1991] (implementado pelo `ReplicationManager`). Este protocolo, construído em camadas, usa um algoritmo de difusão confiável extremamente leve sob um outro de difusão atômica baseado no paradigma do ordenador fixo, onde um processo do grupo é o responsável por definir a ordem total de todas as mensagens [Défago et al., 2000].

Uma análise do desempenho destes três *plugins* (em termos de tempos de resposta) pode ser encontrada em um artigo complementar a este [Bessani et al., 2004].

6. Trabalhos Relacionados

Na literatura, são encontrados diversos trabalhos envolvendo a disponibilidade de serviço de comunicação de grupo e tolerância a faltas na arquitetura CORBA. Estas experiências (classificadas em três abordagens: integração [Maffeis, 1995, ISIS, 1995], serviço [Felber, 1998] e interceptação [da Silva Fraga et al., 1997, Moser et al., 1999, Lung et al., 2000]) se preocupam basicamente em manter características de portabilidade e de desempenho. Considerações sobre interoperabilidade eram limitadas, uma vez que estes trabalhos foram realizados antes da publicação das especificações FT-CORBA e UMIOP pela OMG.

A abordagem de integração consiste na construção ou na modificação de um *middleware* CORBA existente, adicionando mecanismos de comunicação de grupo. Nesta abordagem, o ORB é modificado para que as aplicações não possam distinguir objetos simples de grupos de réplicas, oferecendo, desta forma, um alto grau de transparência. A idéia principal nesta abordagem é que o processamento de grupo seja suportado por uma ferramenta de comunicação de grupo abaixo do núcleo do ORB. Todas as chamadas que envolvam comunicação ou gestão de grupo são repassadas pelo núcleo do ORB a este suporte de mais baixo nível.

A idéia básica na abordagem de serviço é prover a comunicação de grupo como um objeto de serviço sobre o ORB, e não como parte do próprio ORB. Dessa forma, o suporte de grupo deve ser formado por uma coleção de objetos de serviço, que podem estar distribuídos em diferentes estações da rede. Estes objetos são responsáveis pela

[7] Chamadas pelas ferramentas quando da chegada de uma mensagem.

gestão de grupos e pela entrega de mensagens aos objetos membros de grupo, mantendo as propriedades definidas pelo tipo de comunicação de grupo utilizado.

A abordagem de interceptação prevê que as requisições enviadas aos objetos servidores devem ser capturadas e desviadas para um sistema de comunicação de grupo, de maneira transparente para a aplicação. Para tal, podem ser utilizados estruturas das linguagens de programação [Lisbôa, 1997], interfaces do sistema operacional [Moser et al., 1999] ou mesmo mecanismos do próprio ORB [da Silva Fraga et al., 1997, Lung et al., 2000], conhecidos como interceptadores.

A abordagem apresentada neste trabalho pode ser entendida como híbrida, já que combina características das abordagens de interceptação (usando interceptadores do CORBA) e serviço (objeto de serviço SCG). O uso dessa combinação permite uma melhor transparência dos mecanismos de comunicação de grupo: os clientes não precisam acessar o SCG diretamente como ocorre na abordagem de serviço.

7. Conclusão

O presente artigo apresenta um conjunto de estensões ao padrão FT-CORBA que visam melhorá-lo sem no entanto ferir as especificações correntes. Três pontos básicos são estendidos: gerenciamento de grupo, onde um mecanismo de gravação e recuperação do estado de grupos é proposto a partir de uma estrutura de representação de grupos de réplicas em XML; arquitetura do servidor tolerante a faltas, onde os serviços definidos na especificação são agrupados em um pequeno servidor de aplicações que suporta a execução de réplicas; e, finalmente, o suporte a replicação ativa, onde criamos o SCG para a integração de qualquer ferramenta de comunicação de grupo à infra-estrutura de tolerância a faltas. Estas estensões tornam o GROUPPAC um dos suportes de tolerância a faltas mais avançados (senão o mais) em termos de objetos distribuídos.

8. Agradecimentos

Agradecimentos especiais à Ricardo Sangoi Padilha e Luciana de Sá Moreira (pesquisadores do DAS/UFSC) pela implementação original do GROUPPAC e do SCG.

Referências

Bessani, A. N., da Silva Fraga, J., and Lung, L. C. (2002). MJaco: Integração do multicast IP na arquitetura CORBA. In *Anais do XX Simpósio Brasileiro de Redes de Computadores - SBRC'2002*, Buzios, RJ.

Bessani, A. N., da Silva Fraga, J., Lung, L. C., and Alchieri, E. A. (2004). Replicação ativa no CORBA: Padrões, protocolos e framework de implementação. In *Anais do XXII Simpósio Brasileiro de Redes de Computadores - SBRC'2004*, Gramado, RS.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2004). Extensible markup language (XML) 1.0 (third edition). Recomendation, World Wide Web Consortium, http://www.w3.org/TR/REC-xml.

da Silva Fraga, J., Maziero, C. A., Lung, L. C., and Filho, O. G. L. (1997). Implementing replicated services in open systems. In *Proceedings of the 3rd IEEE International Symposium on Autonomous Descentralized Systems - ISADS'97*, pages 273–280, Lausanne, Suiça.

Défago, X., Schiper, A., and Urban, P. (2000). Totally ordered broadcast and multicast algorithms: a comprehensive survey. Technical Report TR DSC/2000/036, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

Fallside, D. C. (2001). XML Schema part 0: Primer. Recomendation, World Wide Web Consortium, http://www.w3.org/TR/xmlschema-0/.

Felber, P. (1998). *The CORBA Object Group Service - A Service Approach to Object Groups in CORBA*. Tese de doutorado, École Polytechnique Fédérale de Lausanne, Lausanne, Suiça.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading.

Hadzilacos, V. and Toueg, S. (1994). A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell University, New York - USA.

ISIS (1995). Isis distributed systems inc, iona technologies, ltd. - orbix+isis programmer's guide. Document D070-00.

JGroups (2004). Jgroups: A toolkit for reliable multicast communication. Disponível em http://www.jgroups.org.

Lisbôa, M. L. B. (1997). Arquiteturas de meta-nível. In *Anais do Simpósio Brasileiro de Engenharia de Software - SBES'97*, Fortaleza, CE.

Lung, L. C., da Silva Fraga, J., Farines, J. M., and Oliveira, J. R. (2000). Experiências com comunicação de grupo nas especificações fault tolerant corba. In *Anais do 18o. Simpósio Brasileiro de Redes de Computadores*, Belo Horizonte - MG - Brasil.

Lung, L. C., da Silva Fraga, J., Padilha, R., and Souza, L. (2001). Adaptando as especificações ft-corba para redes de larga escala. In *Anais do XIX Simpósio Brasileiro de Redes de Computadores - SBRC'2001*, Florianópolis, SC.

Maffeis, S. (1995). Adding group communication and fault-tolerance to CORBA. In *Proceedings of the USENIX Conference on Object Oriented Technologies*, pages 135–146, Monterey, Canada.

Moser, L. E., Melliar-Smith, P. M., Narasimhan, P., Tewksbury, L. A., and Kalogeraki, V. (1999). The eternal system: an architecture for enterprise applications. In *Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99)*, Mannhein - Alemanha.

Object Management Group (2001). Unreliable multicast inter-orb protocol specification v1.0. OMG Standart ptc/03-01-11.

Object Management Group (2002). The common object request broker architecture: Core specification v3.0. OMG Standart formal/02-12-06.

Ricciardi, A. M. and Birman, K. (1991). Using process groups to implement failure detection in asynchronous environments. In *ACM Symposium on Principles of Distributed Computing*, pages 341–353, Montreal - Quebec - Canada.

Schneider, F. B. (1990). Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

van Renesse, R., Birman, K. P., Mark Hayden, A. V., and Karr, D. (1998). Building adaptative systems using ensemble. *Software - Pratice and Experience*, 28(9):963–979.

Uso de *Broadcast* na Sincronização de *Checkpoints* em Protocolos Minimais

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

Universidade Estadual de Campinas
Caixa Postal 6176
13083-970 Campinas, SP, Brasil
Tel: +55 19 3788 5876 Fax: +55 19 3788 5847

{tiemi, islene, buzato}@ic.unicamp.br

Abstract. *In a synchronous checkpoiting, the application can be easily recovered in case of failures because the processes may rollback to their last checkpoint on stable storage. This article examine the minimal protocols, in which just a minimal number of processes take checkpoints to construct a consistent global checkpoint. Cao and Singhal propose a new approach to develop a minimal protocol. This approach uses a broadcast to block all processes and centralizes to a unique process the task of determine which processes should take checkpoints during the consistent global checkpoint construction. This article contains a prove the protocol proposed by Cao and Singhal is not minimal and we propose a correction to change the protocol and to guarantee the minimality.*

Resumo. *Nos protocolos de checkpointing síncronos, a aplicação pode ser facilmente restabelecida após a ocorrência de uma falha, pois basta que todos os processos retornem ao seu último checkpoint salvo. Neste artigo, exploramos a classe de protocolos síncronos minimais na qual um número minimal de processos salva checkpoints a cada invocação do protocolo para a construção de um checkpoint global consistente. Cao e Singhal propuseram uma nova abordagem para desenvolver um protocolo minimal que utiliza broadcast para bloquear todos os processos e centralizar a um único processo a tarefa de determinar quais processos devem salvar checkpoints durante a a construção do checkpoint global consistente. Neste texto, mostramos a não minimalidade do protocolo de Cao e Sighal e propomos uma correção para tornar o protocolo minimal.*

1. Introdução

Um sistema distribuído é composto por uma coleção de computadores autônomos interligados por uma rede de comunicação e é visto pelos usuários como um único sistema coerente [17]. As aplicações distribuídas possuem melhor desempenho, são escaláveis e permitem compartilhamento transparente de recursos existentes no sistema, porém estão mais sucetíveis a falhas. Para evitar perda de computação na recuperação de um sistema após a ocorrência de uma falha, os protocolos de *checkpointing* armazenam os estados dos

*Tiemi C. Sakata recebe apoio financeiro do CNPq, processo número 141808/01-2.

processos, também chamados de *checkpoints*, periodicamente em memória estável. Para garantir a recuperação de um sistema distribuído com qualquer padrão de comunicação é necessário que o sistema retroceda para o último estado global consistente, ou seja, um estado global que poderia ter sido obtido por um observador onisciente externo [4].

Nos protocolos de *checkpointing* síncronos não ocorre o chamado efeito dominó (retrocesso da aplicação ao estado inicial em caso de falha) [14]. Utiliza-se mensagens de controle para sincronizar os processos sinalizando o momento de início e término da execução do protocolo. O início é determinado pela invocação por algum dos processos de sua implementação local do protocolo. O término é determinado via a verificação de uma condição global que indica que todos os *checkpoints* necessários foram armazenados. A execução do protocolo, desde o início por um dos processos, o iniciador, até seu término é denominada *construção global*. Ao final de uma construção global, o *checkpoint* global consistente representado pelos últimos *checkpoints* locais é o *checkpoint* global consistente mais recente.

Os protocolos sícronos minimais tentam diminuir o custo total de armazenamento induzindo um número minimal de processos a gravarem seus *checkpoints* [8, 10, 15]. Cao e Singhal provam que todo protocolo minimal é bloqueante [1], ou seja, os processos que salvam *checkpoints* ficam bloqueados para a aplicação durante a construção global.

Houve na literatura um esforço em tentar reduzir o número de mensagens de controle necessários durante uma construção global. Prakash e Singhal propõem um mecanismo de rastreamento de dependências e utilizam um vetor de bits propagado durante a construção global para reduzir o número de mensagens de controle necessário [12]. Notamos porém que este mecanismo não é suficiente para garantir a minimalidade e propusemos um novo mecanismo baseado em precedência entre os *checkpoints* dos processos para desenvolver um protocolo minimal [15].

Uma abordagem alternativa para a construção de protocolos minimais foi proposta por Cao e Singhal [3]. Esta abordagem utiliza o mesmo mecanismo proposto por Prakash e Singhal [12] para rastrear dependências. Ao iniciar uma construção global, o processo iniciador salva um *checkpoint* e faz um *broadcast* pedindo a todos os processos suas informações de dependências. Todos os processos que recebem essa mensagem ficam bloqueados para a aplicação e enviam uma resposta ao iniciador. O iniciador coleta as informações de dependências de todos os processos e seleciona quais processos devem salvar *checkpoints*. O iniciador envia então uma mensagem de requisição aos processos selecionados e uma mensagem de liberação para os não selecionados, desbloqueando-os.

Neste artigo mostramos que o mecanismo de rastreamento de dependências empregado por Prakash e Singhal [12] e Cao e Singhal [3] não garante que a minimalidade do protocolo. Um contra-exemplo demonstra que o protocolo de Cao e Singhal [3] não é minimal e nos permite propor um protocolo minimal que utiliza um mecanismo para capturar as precedências entre *checkpoints* ao invés de dependências entre processos.

O restante do texto está organizado da seguinte maneira. A Seção 2 descreve o modelo computacional que consideramos neste artigo. A Seção 3 compara as duas abordagens existentes para protocolos de *checkpointing* minimais. A Seção 4 descreve uma de nossas contribuições em protocolos minimais. A Seção 5 descreve o protocolo proposto. A Seção 6 encerra o documento descrevendo as conclusões do trabalho.

2. Modelo Computacional

Um sistema distribuído é composto por n processos seqüênciais e autônomos que executam eventos internos, de envio e de recepção—troca—de mensagens. A troca de mensagens é o único mecanismo de comunicação utilizado pelos processos. Há garantia de entrega de mensagens, mas estas podem sofrer atrasos arbitrários e chegar aos seus destinos fora de ordem. A comunicação é feita através de canais unidirecionais que interligam pares de processos. Consideramos que a rede de comunicação é fortemente conexa (nenhum processo é isolado), mas não necessariamente completa (a comunicação entre um par de processos pode se dar via processos intermediários). Não existem mecanismos para compartilhamento de memória, acesso a um relógio global, sincronização de relógios locais ou conhecimento a respeito das diferenças de velocidade entre os processadores. A memória estável é suficiente para gravar todos os *checkpoints* necessários à computação, garantindo a correta recuperação de seu estado em caso de falha.

2.1. Precedência e Consistência

A noção de estado consistente de um sistema distribuído é derivada do conceito de precedência causal entre eventos que foi proposto por Lamport [9]. A definição de precedência causal utiliza e_i^ι para denotar o ι-ésimo evento executado pelo processo p_i.

Definição 1 Precedência causal — *Um evento e_a^α precede um evento e_b^β ($e_a^\alpha \rightarrow e_b^\beta$) se*

(i) $a = b$ *e* $\beta = \alpha + 1$, *ou*

(ii) existe uma mensagem m *que foi enviada em* e_a^α *e recebida em* e_b^β, *ou*

(iii) existe um evento e_c^γ *tal que* $e_a^\alpha \rightarrow e_c^\gamma \wedge e_c^\gamma \rightarrow e_b^\beta$.

Um *checkpoint* é um evento interno que armazena o estado do processo em memória estável e um intervalo entre *checkpoint*s é o conjunto de eventos ocorridos entre dois *checkpoints* consecutivos do mesmo processo. Utilizamos c_i^ι para denotar o ι-ésimo *checkpoint* gravado pelo processo p_i, sendo que $\iota = 1$ representa o *checkpoint* inicial.

Dizemos que c_a^α precede c_b^β ($c_a^\alpha \rightarrow c_b^\beta$) se o evento que originou c_a^α precede o evento que originou c_b^β. Um *checkpoint* c_a^α precede diretamente um *checkpoint* c_b^β se o processo p_a enviou uma mensagem m para p_b após o armazenamento de c_a^α e p_b recebeu m antes do armazenamento de c_b^β. Uma precedência transitiva de c_a^α para c_b^β é formada por uma seqüência de mensagens iniciada por p_a após c_a^α e terminada em p_b antes de c_b^β.

A definição de checkpoint global consistente é baseada no conceito de precedência entre checkpoints, que é capturada formalmente através da relação de z-precedência entre *checkpoints* [6]. Esta relação equivale ao conceito de *zigzagpath* introduzido originalmente por Netzer e Xu [11].

Definição 2 Z-Precedência—*Um checkpoint c_a^α z-precede um checkpoint c_b^β ($c_a^\alpha \rightsquigarrow c_b^\beta$) se*

(i) $c_a^\alpha \rightarrow c_b^\beta$, *ou*

(ii) $\exists c_c^\gamma : (c_a^\alpha \rightsquigarrow c_c^\gamma) \wedge (c_c^{\gamma-1} \rightsquigarrow c_b^\beta)$

Um *checkpoint* global é formado por um conjunto de *checkpoints*, um por processo e é consistente se não existe relação de z-precedência entre os *checkpoints* do conjunto.

Definição 3 ***Checkpoint*** **global consistente** — *Um checkpoint global* $\mathcal{C} = \{c_0^{\iota 0}, \ldots, c_{n-1}^{\iota n-1}\}$ *é consistente se, e somente se,* $\forall i, j : 0 \leq i, j < n : c_i^{\iota i} \not\rightsquigarrow c_j^{\iota j}$.

A Figura 1 ilustra um diagrama espaço-tempo para três processos com quadrados em preto representando *checkpoints*. A linha $\mathcal{C}$ forma um *checkpoint* global inconsistente pois o primeiro *checkpoint* de p_0 z-precede o segundo *checkpoint* de p_1. A linha $\mathcal{C}'$ mostra um *checkpoint* global consistente.

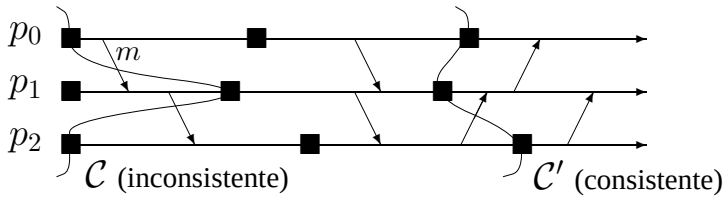

Figura 1: Consistência em *checkpoints* globais

3. Desenvolvimento de Protocolos Minimais

Um protocolo minimal é aquele que induz um conjunto minimal de processos a salvarem *checkpoints* durante uma construção global. Para que isso ocorra ele deve então induzir a retirada de *checkpoints* somente nos processos que durante o seu último intervalo de *checkpoint* forem z-precedentes do *checkpoint* armazenado pelo processo que iniciou a construção. A busca da minimalidade gerou duas abordagens para a organização dos processos durante a construção global: (i) árvore e (ii) *broadcast*.

3.1. Protocolos Minimais com Abordagem Árvore

Os protocolos propostos na literatura que utilizam a abordagem em que os processos se organizam em árvore são executados em três fases [8, 10, 15]:

1. fase de requisições – um processo iniciador invoca o protocolo síncrono armazenando um *checkpoint*, fica bloqueado e envia mensagens de requisição para os processos que dependem dele no seu último intervalo de *checkpoint*.
2. fase de respostas – cada processo que recebe uma mensagem de requisição pode salvar um *checkpoint*, ficar bloqueado e enviar uma mensagem de resposta para o processo emissor da requisição ou diretamente ao iniciador.
3. fase de liberações – após receber mensagens de resposta dos processos envolvidos, o iniciador se desbloqueia e envia uma mensagem de liberação aos processos participantes para que esses também voltem a sua computação normal.

A Figura 2 ilustra as fases de um protocolo minimal. O processo p_3 é o iniciador e envia uma mensagem de requisição para p_2. O processo p_2, após salvar um *checkpoint*, envia uma mensagem de requisição para p_1 e p_0. Note que nem todos os processos necessitam armazenar *checkpoints* em uma construção global. Todo processo que recebe uma mensagem de requisição, envia uma mensagem de resposta ao iniciador que encerra a construção global através do envio das mensagens de liberação.

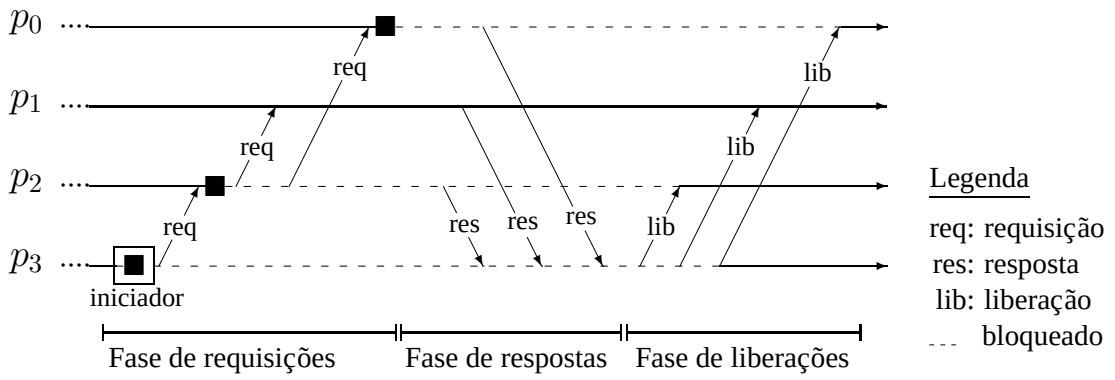

Figura 2: Protocolo Minimal com Abordagem Árvore

3.2. Protocolos Minimais com Abordagem *Broadcast*

Cao e Singhal propuseram uma nova abordagem, a qual chamamos de abordagem *broadcast*, que permite a um único processo (o iniciador) decidir quais os processos que devem armazenar *checkpoints* durante uma construção global [3]. Note que todos os processos devem ficar bloqueados para a aplicação enquanto o iniciador toma essa decisão.

O mecanismo utilizado por Cao e Singhal [3] para rastrear as dependências formadas pelas mensagens da aplicação é semelhante ao mecanismo utilizado por Prakash e Singhal [12]. Neste mecanismo, todos os processos anotam em um vetor de bits as mensagens recebidas no seu último intervalo de *checkpoints*. Este protocolo necessita de duas fases adicionais iniciais:

1. fase de *broadcast* – o iniciador faz um *broadcast* da mensagem de bloqueio. Todo processo que recebe esta mensagem envia seu vetor com as informações de dependência entre processos ao iniciador e fica bloqueado.
2. fase de respostas ao bloqueio – o iniciador constrói uma matriz $n \times n$ a partir dos vetores recebido pelos processos. Através desta matriz, o iniciador pode verificar quais os processos que dependem transitivamente do iniciador, ou seja, os processos que devem armazenar *checkpoints* (participantes) durante a construção global. O iniciador envia uma mensagem de requisição a cada participante e uma mensagem de liberação aos outros processos.

Os processos participantes da construção global passam pelas mesmas três fases da abordagem árvore, ou seja, os processos que recebem uma mensagem de requisição, armazenam um *checkpoint* e enviam uma mensagem de resposta ao iniciador. Quando o iniciador recebe uma mensagem de resposta de cada um dos participantes, fica desbloqueado e envia uma mensagem de liberação a todos os participantes da construção global para que voltem à sua computação normal. Um exemplo de execução da abordagem *broadcast* é ilustrado pela Figura 3.

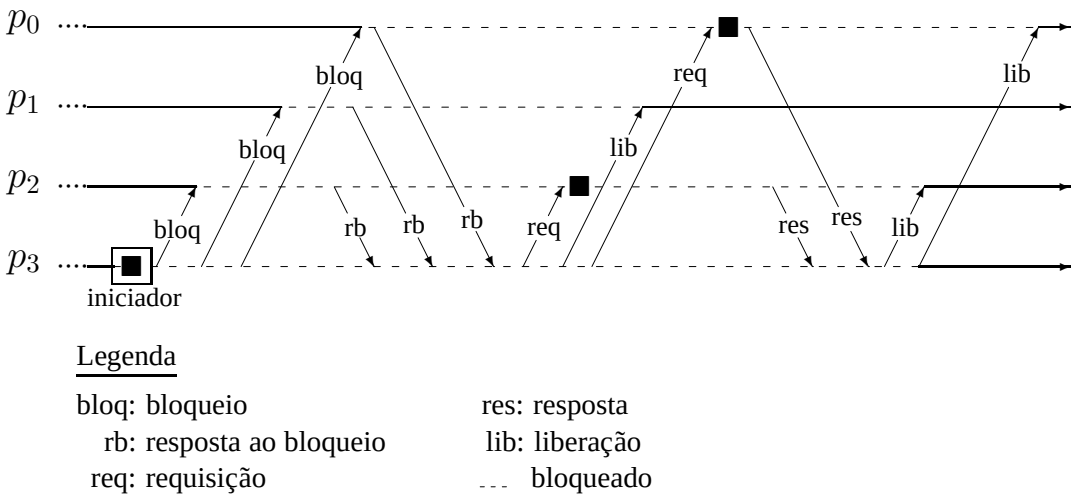

Figura 3: Protocolo Minimal com Abordagem *Broadcast*

4. Uma Contribuição para o Desenvolvimento de Protocolos Minimais

A Seção anterior descreveu os protocolos síncronos minimais em função das fases e estratégias de troca de mensagens utilizadas para a obtenção de *checkpoints* globais consistentes. Nesta Seção, descrevemos esses protocolos sob o ponto de vista da busca pela minimalidade e observamos que dois deles não atingem esse objetivo. Essa é uma das contribuições deste trabalho, a outra é a descrição de um algoritmo que corrige essa deficiência.

Um protocolo minimal pode deixar de ser correto durante a construção global se:

- ele inclui no conjunto de participantes um processo cujo último *checkpoint* não z-precede o *checkpoint* armazenado pelo iniciador e/ou
- ele deixa de incluir entre os participantes um processo cujo último *checkpoint* z-precede o *checkpoint* armazenado pelo iniciador.

Um protocolo que exibe o primeiro defeito deixa de ser minimal porém constrói um *checkpoint* global consistente, porque incluiu no conjunto de participantes um processo desnecessário. Um protocolo que exibe o segundo defeito, obterá ao final da construção global um *checkpoint* global *inconsistente*.

Os protocolos propostos por Prakash e Singhal [12] e por Cao e Singhal [3] baseiam-se em um mecanismo de rastreamento de dependências que faz com que eles exibam o primeiro defeito, deixando portanto, de serem minimais. O restante desta Seção demonstra este fato através da análise de um contra-exemplo aplicado ao protocolo de Cao e Singhal [3] (Figura 4); a não minimalidade do protocolo proposto por Prakash e Singhal é descrita em [15].

O mecanismo de execução do protocolo de Cao e Singhal está descrito na Seção 3.2 e um exemplo do funcionamento do protocolo é ilustrado pela Figura 4. Na figura, p_1 envia uma mensagem para p_0 e inicia uma construção global trivial porque p_1 não recebeu mensagens durante o seu último intervalo de *checkpoint*. Assim, essa construção encerra-se sem que p_0 tenha que salvar um *checkpoint* e o *checkpoint* global consistente minimal obtido contém os *checkpoints* $\mathcal{C} = \{a, b\}$. Em um outro momento,

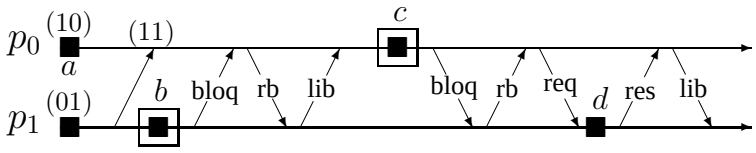

Figura 4: Contra exemplo para o protocolo proposto por Cao e Singhal [3]

p_0 assume o papel de iniciador. Observe que p_0 recebeu uma mensagem de p_1 durante o seu último intervalo de *checkpoint* e é por isso que p_0 pede a p_1 que salve o *checkpoint* d. Neste ponto, o protocolo de Cao e Singhal exibe o primeiro defeito porque induziu p_1 a salvar um *checkpoint* desnecessário já que o $\mathcal{C}' = \{c, b\}$ é consistente.

5. Protocolo Broad-min

Nesta seção, propomos um protocolo síncrono minimal, chamado Broad-min, baseado no protocolo proposto por Cao e Singhal [3].

O protocolo Broad-min utiliza vetores de relógios para rastrear as precedências existentes entre os últimos *checkpoints* de cada processo [5, 16]. Cada processo possui um vetor de relógios de n entradas iniciadas com valor 0. Todo processo p_i incrementa o valor da posição i de seu vetor imediatamente antes da gravação de um *checkpoint*. Na recepção de cada mensagem msg da aplicação o processo p_j atualiza o seu vetor de relógios da seguinte maneira:

se $\texttt{VR}_j\texttt{[i]} < \texttt{msg.VR[i]}$, então $\texttt{VR}_j\texttt{[i]} \leftarrow \texttt{msg.VR[i]}$, para todo $i \neq j$.

5.1. Descrição do Protocolo

Nesta seção, mostramos o mecanismo de execução do protocolo Broad-min. Ao final da seção, descrevemos o algoritmo em pseudo-código deste protocolo e ilustramos sua execução através de um exemplo.

Início da construção global

Um processo p_i, ao iniciar uma construção global, armazena um *checkpoint*. Se p_i não recebeu nenhuma mensagem no último intervalo de *checkpoint*, a construção global é encerrada. Caso contrário, p_i fica bloqueado para a aplicação e faz um *broadcast* da mensagem de bloqueio.

Recepção da mensagem de bloqueio

Todo processo p_j, ao receber uma mensagem de bloqueio, fica bloqueado para a aplicação e envia o seu vetor de relógios `VR` e um vetor de bits, chamado vetor de participantes `VP`, ao iniciador através da mensagem de resposta ao bloqueio. O vetor de participantes indica quais os processos que dependem direta ou transitivamente de p_j no último intervalo e é calculado comparando o vetor de relógios `VR` atual com o vetor de relógios salvo juntamente com último *checkpoint* `ultimo_VR`. Ou seja, existe uma precedência causal entre um *checkpoint* de p_i com o último *checkpoint* de p_j se p_j recebeu direta ou indiretamente uma informação nova de p_i:

$$\texttt{VP}_j\texttt{[i]} = 1 \;\Rightarrow\; \texttt{VR}_j\texttt{[i]} - \texttt{ultimo\_VR}_j\texttt{[i]} > 0$$

Recepção da mensagem de resposta ao bloqueio

O iniciador p_i ao receber uma mensagem de resposta ao bloqueio de p_j, insere `VR` de p_j na linha j da matriz `MVR` e da mesma forma, insere `VP` em sua matriz `MVP`. Ao receber a mensagem de resposta ao bloqueio de todos os processos, p_i possui duas matrizes nxn: `MVR` com os vetores de relógio e `MVP` com as informações de dependências de cada processo. O iniciador calcula então todos os prováveis participantes de sua construção global através da multiplicação do seu vetor de participantes e `MVP`. Para garantir a minimalidade do protocolo, um processo p_p faz parte do conjunto de processos participantes da construção global somente se pelo menos outro participante do conjunto conhece o último *checkpoint* de p_p. O iniciador p_i envia uma mensagem de requisição para cada processo participante e uma mensagem de liberação para os outros processos.

Recepção da mensagem de requisição

Todo processo que recebe uma mensagem de requisição, salva um *checkpoint*, e envia uma mensagem de resposta ao iniciador.

Recepção da mensagem de resposta

O iniciador, ao receber uma mensagem de resposta de cada um dos participantes, conclui que a construção global pode ser encerrada e envia uma mensagem de liberação para todos os participantes e volta a processar a aplicação.

Recepção da mensagem de liberação

Todo processo que recebe uma mensagem de liberação, fica desbloqueado voltando à sua computação normal.

O protocolo Broad-min é descrito pelo Algoritmo 1. Consideramos que todo procedimento deste algoritmo é executado de forma atômica.

5.2. Exemplo

A execução deste protocolo pode ser exemplificada através do cenário ilustrado pela Figura 5. Neste cenário, a construção global iniciada por p_2 é encerrada após p_2 salvar um *checkpoint* pois p_2 não recebeu mensagens no último intervalo de *checkpoint*. Já quando p_0 inicia uma construção global, após salvar um *checkpoint*, faz um *broadcast* da mensagem de bloqueio. Ao receber a mensagem resposta ao bloqueio de todos os processos, p_0 possui as matrizes de Vetores de Relógios e de Vetores de Precedências (Figura 6).

Para calcular os possíveis participantes, p_2 multiplica seu vetor de participantes com `MVP` até que não haja mudança no vetor de participantes. Pela Figura 5, este cálculo é feito pelo iniciador p_0 e é descrito pela Figura 7.

Ao final da multiplicação das matrizes, temos como resultado um vetor indicando quais processos da aplicação são prováveis participantes da construção global. O nosso objetivo porém é de induzir um número minimal de processos a armazenarem *checkpoints*. Através da matriz `MVR` podemos notar que nenhum processo conhece o último

Algoritmo 1: Broad-min

Variáveis do processo:
VR $\equiv$ vetor$[0...n-1]$ de inteiros
ultimo_VR $\equiv$ vetor$[0...n-1]$ de inteiros
participantes $\equiv$ vetor$[0...n-1]$ de bits
aux $\equiv$ vetor$[0...n-1]$ de bits
MVR $\equiv$ matriz$[0...n-1][0...n-1]$ de inteiros
MVP $\equiv$ matriz$[0...n-1][0...n-1]$ de bits
pid $\equiv$ inteiro
bloqueado $\equiv$ booleano

Tipos de mensagem:
m – aplicação:
VR $\equiv$ vetor$[0...n-1]$ de inteiros
bloq – bloqueio
rb – resposta ao bloqueio:
VR $\equiv$ vetor$[0...n-1]$ de inteiros
VP $\equiv$ vetor$[0...n-1]$ de bits
req – requisição
res – resposta
lib – liberação

Início:
$\forall i$: VR[i] $\leftarrow$ 0, ultimo_VR[i] $\leftarrow$ 0
$\forall i$: participantes[i] $\leftarrow$ 0, aux[i] $\leftarrow$ 0
$\forall i$: $\forall j$: MVR[i][j] $\leftarrow$ 0, MVP[i][j] $\leftarrow$ 0
bloqueado $\leftarrow$ *falso*
VR[pid] $\leftarrow$ VR[pid] + 1
armazena o *checkpoint*

Envio da mensagem da aplicação (m) para p_k:
se (*não* bloqueado)
m.VR $\leftarrow$ VR
transmite a mensagem da aplicação (m)

Recepção da mensagem da aplicação (m) de p_k:
se (*não* bloqueado)
$\forall i$: se (m.VR[i] $>$ VR[i])
VR[i] $\leftarrow$ m.VR[i]
processa a mensagem da aplicação (m)

Início da construção global:
VR[pid] $\leftarrow$ VR[pid] + 1
armazena o *checkpoint*
MVR[pid] $\leftarrow$ VR
$\forall i$: se ((VR[i] $-$ ultimo_VR[i]) $>$ 0)
MVP[pid][i] $\leftarrow$ 1
se ($\exists i \neq$ pid: MVP[pid][i] = 1)
bloqueado $\leftarrow$ *verdadeiro*
faz um broadcast do bloqueio (bloq)

Recepção do bloqueio (bloq) de p_k:
bloqueado $\leftarrow$ *verdadeiro*
rb.VR $\leftarrow$ VR
$\forall i$: se ((VR[i] $-$ ultimo_VR[i]) $>$ 0)
rb.VP[i] $\leftarrow$ 1
transmite a resposta (rb) do bloqueio para p_k

Recepção da resposta ao bloqueio (rb) de p_k:
MVR[k] $\leftarrow$ rb.VR
MVP[k] $\leftarrow$ rb.VP
participantes[k] $\leftarrow$ 1
se ($\forall i$: participantes[i] = 1)
participantes $\leftarrow$ MVP[pid]
aux $\leftarrow$ participantes
participantes $\leftarrow$ participantes $*$ MVP
enquanto (participantes $\neq$ aux)
aux $\leftarrow$ participantes
participantes $\leftarrow$ participantes $*$ MVP
$\forall i \neq$ pid: se (participantes[i] = 1)
se ($\forall j \neq i$: participantes[j] = 1 *e*
MVR[j][i] $<$ MVR[i][i])
participantes[i] $\leftarrow$ 0
enquanto (participantes $\neq$ aux)
aux $\leftarrow$ participantes
$\forall i \neq$ pid: se (participantes[i] = 1)
se ($\forall j \neq i$: participantes[j] = 1 *e*
MVR[j][i] $<$ MVR[i][i])
participantes[i] $\leftarrow$ 0
se ($\forall i \neq$ pid: participantes[i] = 0)
bloqueado $\leftarrow$ *falso*
senão
$\forall i \neq$ pid: se (participantes[i] = 1)
transmite a requisição (req) para p_i
senão
transmite a liberação (lib) para p_i

Recepção da requisição (req) de p_k:
VR[pid] $\leftarrow$ VR[pid] + 1
armazena o *checkpoint*
ultimo_VR $\leftarrow$ VR
transmite (res) para p_k

Recepção da resposta (res) de p_k:
respostas[k] $\leftarrow$ 1
se (participantes = respostas)
$\forall i \neq$ pid: se (participantes[i] $= 1$)
transmite a liberação (lib) para p_i
$\forall i$: participantes[i] $\leftarrow$ 0, respostas[i] $\leftarrow$ 0
$\forall i$: aux[i] $\leftarrow$ 0
$\forall i$: $\forall j$: MVR[i][j] $\leftarrow$ 0, MVP[i][j] $\leftarrow$ 0
ultimo_VR $\leftarrow$ VR
bloqueado $\leftarrow$ *falso*

Recepção da liberação (lib) de p_k:
bloqueado $\leftarrow$ *falso*

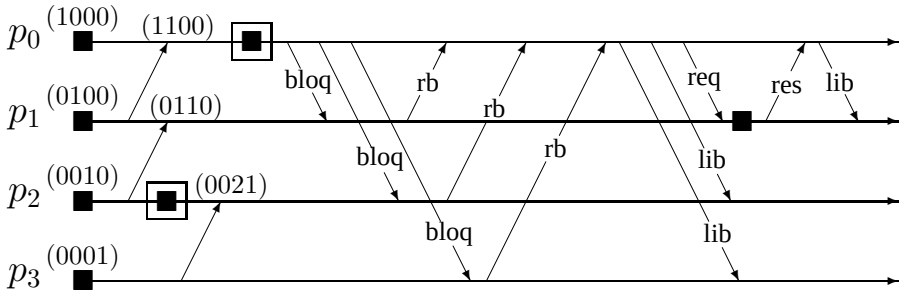

Figura 5: Cenário que ilustra a execução do Broad-min

$$MVR = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad MVP = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(a) Matriz de Vetores de Relógios **(b) Matriz de Vetores de Participantes**

Figura 6: Matrizes construídas pelo iniciador

checkpoint de p_2 (basta verificar a coluna 2 da matriz `MVR`). Portanto p_2 deixa de ser participante da construção de do iniciador p_0. Como o único que conhecia o último *checkpoint* de p_3 era p_2 e p_2 não é participante, então p_3 também deixa de ser participante da construção global. Portanto, p_0 envia uma mensagem de requisição para o seu único participante e envia uma mensagem de liberação para p_2 e p_3. O processo p_1, ao receber a mensagem de requisição, armazena um *checkpoint* e envia uma mensagem de resposta para o iniciador. Após receber a mensagem de resposta de p_1, o iniciador envia uma mensagem de liberação para p_1 e a construção global é encerrada.

$$\begin{pmatrix} 1 & 1 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} =$$

$$= \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \end{pmatrix}$$

Figura 7: Cálculo dos prováveis participantes

6. Conclusão

Os protocolos de *checkpointing* síncronos bloqueantes simplificam a recuperação de uma falha e evitam o efeito dominó mantendo *checkpoints* globais consistentes em memória estável. Para reduzir o custo de cada construção global, os protocolos minimais induzem apenas um número minimal de processos a armazenarem *checkpoints*. Sabemos que todo protocolo minimal é bloqueante, ou seja, os processos participantes construção global suspendem suas atividades para a aplicação durante a execução de *checkpointing*. Observamos que durante o desenvolvimento desses protocolos duas estratégias foram utilizadas: árvore e *broadcast*. A abordagem em que os processos se organizam em árvore bloqueia apenas os processos participantes da construção global e eles determinam se salvam ou não um checkpoint. Em contraste, a abordagem baseada em *broadcast* bloqueia todos os processos e transfere exclusivamente para o iniciador a tarefa de determinar quais processos armazenarão *checkpoints* durante a construção global.

Neste artigo, provamos que o protocolo proposto por Cao e Singhal [3], apesar de construir *checkpoints* globais consistentes, não é minimal. Notamos que o mecanismo de rastreamento de dependências entre processos utilizado por Cao e Singhal [3] não é suficiente para garantir a minimalidade do protocolo. A demonstração desse fato é simples e apóia-se em um contra-exemplo que nos motivou a procurar um novo mecanismo de rastreamento, baseado no uso de vetores de relógios lógicos. Finalmente, utilizamos o novo mecanismo para desenvolver o protocolo Broad-min, que é baseado em *broadcast* e minimal.

Cao e Singhal fizeram uma análise teórica e concluíram que a abordagem *broadcast* minimiza o tempo de bloqueio durante uma construção global. No entanto, acreditamos que a melhor abordagem depende da aplicação e também da configuração do sistema distribuído. Como trabalho futuro, seria interessante desenvolver várias aplicações e executá-las em diferentes configurações de sistemas distribuídos a fim de comparar as duas abordagens.

Outro trabalho futuro seria analisar os protocolos de *checkpointing* síncronos não bloqueantes [1, 2, 3, 7, 12, 13]. Nos protocolos não bloqueantes, os processos não suspendem a execução da aplicação durante uma construção global. Os protocolos síncronos não bloqueantes propostos na literatura são muito complexos e suas provas são baseados nos diferentes cenários gerados por eles. O problema deste tipo de prova é que a falta de um cenário pode gerar conclusões erradas. Cao e Singhal [1] detectam dois problemas existentes no protocolos proposto por Prakash e Singhal [13] e ao analisarmos as provas de correção deste útlimo protocolo, notamos que estão incompletas.

Referências

[1] G. Cao and M. Singhal. On Coordinated Checkpointing in Distributed Systems. *IEEE Trans. on Parallel and Distributed Systems*, 9(12):1213–1225, Dec. 1998.

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

[3] G. Cao and M. Singhal. Checkpointing with Mutable Checkpoints. *Theoretical Computer Science*, 290(2):1127–1148, jan 2003.

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

[5] Ö. Babaoğlu and K. Marzullo. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

[6] I. C. Garcia and L. E. Buzato. Progressive Construction of Consistent Global Checkpoints. In *19th IEEE International Conference on Distributed Computing Systems*, Austin, Texas, EUA, June 1999.

[7] E. Gendelman, L. Bic, and M. B. Dillencourt. An Efficient Checkpointing Algorithm for Distributed Systems Implementing Reliable Communication Channels. In *Symposium on Reliable Distributed Systems*, pages 290–291, 1999.

[8] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transaction on Software Engineering*, 13:23–31, Jan. 1987.

[9] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[10] P. J. Leu and B. Bhargava. Concurrent Robust Checkpointing and Recovery in Distributed Systems. In *4th IEEE Int. Conference on Data Engineering*, pages 154–163, 1988.

[11] R. H. B. Netzer and J. Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Transaction on Parallel and Distributed Systems*, 6(2):165–169, 1995.

[12] R. Prakash and M. Singhal. Minimal Global Snapshot and Failure Recovery using Infection. Technical Report OSU-CISRC-12/93-TR42, Department of Computer Science, The Ohio State University, 1993.

[13] R. Prakash and M. Singhal. Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems. *IEEE Transaction on Parallel and Distributed Systems*, 7(10):1035–1048, Oct. 1996.

[14] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transaction on Software Engineering*, 1(2):220–232, June 1975.

[15] T. C. Sakata, I. C. Garcia, and L. E. Buzato. Checkpointing Síncrono Bloqueante Minimal com Iniciadores Concorrentes. In *Simpósio Brasileiro de Redes de Computadores*, pages 681–696, Natal, Rio Grande do Norte, May 2003.

[16] R. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, Mar. 1994.

[17] A. S. Tanenbaum and M. Steen. *Distributed Systems Principles and Paradigms*. Alan Apt, 2002.