Revista de
Informática
Teórica e Aplicada

Recuperação de Processos em Sistemas Distribuídos

Ingrid Jansch-Pôrto Taisy Silva Weber
{ingrid, taisy}@inf.ufrgs.br
Curso de Pós-Graduação em Ciência da Computação
Instituto de Informática
Universidade Federal do Rio Grande do Sul

Resumo

Sistemas computacionais tornam-se mais confiáveis se forem empregadas técnicas adequadas de recuperação pós-falhas. Como estas técnicas baseiam-se em redundância de componentes e dados, e os sistemas distribuídos podem dispor facilmente desta redundância, parece natural incorporar procedimentos de recuperação nesses sistemas. Esse artigo apresenta os conceitos básicos associados à recuperação em sistemas distribuídos mostrando exemplos destes procedimentos incorporados a sistemas operacionais.

Abstract

Computing systems become more dependable when appropriate fault recovery techniques are applied to them. These techniques are based on components or data redundancy. Considering the implicit redundancy of distributed systems, it seems natural to implement recovery facilities in these systems. This paper is a tutorial on the concepts related to recovery in distributed systems and illustrates fault recovery through examples of recovery protocols implemented in operating systems.

Palavras-chave: sistemas distribuídos, tolerância a falhas, modelo de falhas, pontos de recuperação, recuperação por retorno, processos reservas.

1 Introdução

A popularização de microprocessadores de alto desempenho a baixo custo e os avanços na tecnologia de comunicação têm motivado o desenvolvimento acelerado de sistemas distribuídos. Além das vantagens de compartilhamento de recursos, do aumento do desempenho e da possibilidade de expansão modular, um sistema de computação distribuído oferece maior confiabilidade e disponibilidade, pois alguns componentes do sistema podem falhar sem derrubar o restante do sistema. Através da replicação de dados e serviços, os sistemas distribuídos podem ser tornados tolerantes a falhas.

Se por um lado a redundância é o meio-chave de obtenção de dependabilidade [LAP95], por outro lado, esta mesma redundância aumenta a dificuldade gerencial do sistema. Estas dificuldades não devem ser repassadas aos usuários; tampouco devem ser fonte de erros no sistema. Embora haja uma vocação natural dos sistemas distribuídos para suportar tolerância a falhas, esta propriedade deve ser prevista durante o projeto. Sua implementação ocorre, em geral, de forma incorporada ao sistema operacional, a fim de que esteja acessível a todos os usuários e de forma transparente às aplicações.

O projeto do sistema operacional distribuído prevê o uso de mecanismos semelhantes aos dos sistemas convencionais tais como sincronização de processos, escalonamento, sistemas de arquivos, comunicação entre processos, gerenciamento de memórias, recuperação de falhas, entre outros. Entretanto, a complexidade adicional deve-se a características próprias dos sistemas distribuídos que são: ausência de memória compartilhada e de relógio físico global, bem como da imprevisibilidade dos atrasos de comunicação [SIN94]. A questão da recuperação de falhas interessa tanto à área de sistemas operacionais quanto à de tolerância a falhas. Este tema constitui-se no assunto central deste artigo.

O simples uso de um editor de textos ensina que há necessidade de tomar alguns cuidados, tais como salvamentos periódicos dos arquivos que estão sendo modificados, para que o usuário possa executar procedimentos de recuperação, após o aparecimento de falhas. **Recuperar** um sistema de computadores significa restaurar o sistema ao seu estado operacional normal, após a ocorrência de falhas. A recuperação pode parecer uma tarefa simples, pois basta reiniciar o computador após a falha, repetir um processo que não pôde ser completado ou recomeçar as atividades interrompidas. Entretanto, questões econômicas, por um lado, e exigências das aplicações, por outro lado, fazem com que os problemas não sejam resolvidos de forma tão simples.

Em um computador, os recursos tais como memória e arquivos são alocados aos processos em execução. Quando um processo falha, os recursos que estavam alocados a ele devem ser liberados, a fim de que possam ser usados pelos demais processos, e as alterações que foram realizadas pelo processo interrompido devem ser desfeitas. A reexecução desde o início é uma opção viável para implementação, entretanto resulta em desperdício do tempo e dos recursos empregados no processamento já realizado.

Os sistemas distribuídos, por sua vez, oferecem alto desempenho e maior disponibilidade à custa da execução concorrente de diversos processos, os quais cooperam na realização de tarefas. As falhas de um ou mais destes processos repercutem sobre aqueles com os quais interagiram; assim, os efeitos devidos às interações devem ser desfeitos ou o conjunto dos processos envolvidos deve ser reexecutando a partir de um estado adequado.

A maior disponibilidade nestes sistemas é devida principalmente à replicação dos processos, de dados ou dos componentes do hardware. Mas, por ocasião da falha de um nodo, as cópias dos dados nele armazenados perdem as atualizações, ficando inconsistentes com o restante do sistema quando do retorno à operação. Assim, a recuperação envolve a habilidade de utilizar os dados adequados e de realizar os procedimentos que integrem este nodo de forma consistente com o restante do sistema, como parte de suas ações.

São diversas as opções oferecidas para a implementação dos mecanismos de recuperação de sistemas distribuídos, sua definição dependerá das políticas de funcionamento do sistema, das aplicações previstas e do desempenho esperado. Estas questões, além da conceituação básica, do impacto das falhas sobre os sistemas distribuídos e das formas de lidar com conseqüências destas falhas, serão analisadas neste artigo.

O artigo está organizado como segue: a seção 2 é dedicada aos conceitos básicos necessários à abordagem do tema e a seção 3 conceitua pontos de recuperação. A seção 4 apresenta os procedimentos pós-detecção de falhas relativos a recuperação de processos. Através da apresentação de dois algoritmos de recuperação, um síncrono e outro assíncrono, é mostrada a dependência entre as ações de recuperação e as realizadas durante o processamento normal. A seção 5 ilustra com exemplos os conceitos apresentados nas seções anteriores.

2 Tolerância a falhas em sistemas distribuídos

Na área de tolerância a falhas, os termos falha, erro e defeito (*fault*, *error* e *failure*, [LAP85]) têm significados diferentes [NEL90]. Uma **falha** é uma condição física anômala. As causas podem ser erros de projeto ou em sua implementação; problemas de fabricação; fadiga, acidentes ou deterioração; ou ainda distúrbios externos, tais como interferência eletromagnética, entradas imprevistas ou mau uso do sistema. Um **erro** é a manifestação de uma falha no sistema, no qual o estado lógico de um elemento difere do valor previsto. Uma falha existente no sistema não necessariamente resulta em um erro, mas apenas quando ocorre uma condição conveniente para esta manifestação. O **defeito** corresponde à incapacidade de algum componente em realizar a função para o qual foi projetado. Os defeitos são causados pela existência de falhas e conseqüentemente erros neste mesmo sistema.

Um sistema **tolerante a falhas** deve evitar seu próprio mau funcionamento mesmo na presença de falha de algum de seus componentes. A tolerância a falhas dos sistemas é obtida através do uso de redundância de hardware, de software, de informação ou do próprio processamento da computação. Não existe uma técnica geral para "adicionar" tolerância a falhas a um sistema. Entretanto, podem ser identificadas algumas estratégias úteis

ao projeto destes sistemas. As estratégias incluem [NEL90]: mascaramento dos erros; detecção dos erros; prevenção da propagação de erros (confinamento); identificação do módulo em falha responsável pelo erro detectado (diagnóstico); eliminação ou substituição de componente falho ou ativação de um mecanismo para isolá-lo (reparo ou reconfiguração); correção do sistema a um estado aceitável para prosseguir na operação (recuperação).

Nem todas essas estratégias são usadas, simultaneamente, em um mesmo sistema. A escolha das estratégias adequadas tem fortes laços com o tipo de aplicação ao qual o sistema se destina. Por exemplo, estratégias de diagnóstico e reparo não são adequadas para aplicação em sistemas onde é necessária alta confiabilidade sem possibilidade de paradas ao longo da operação. Os principais parâmetros considerados nas aplicações são: graus de confiabilidade e de disponibilidade exigidos, tempo de missão e períodos entre atividades de manutenção.

2.1 Fases do processo de tolerância a falhas

Anderson e Lee [AND81] identificam fases para prover tolerância a falhas: detecção de erros, confinamento e avaliação de danos, recuperação de erros e tratamento de falhas.

A **detecção de erros** é a fase na qual a presença de falhas é percebida, devido a algum desvio no estado do sistema com relação à especificação inicial. Ao detectar o erro, pode-se inferir que há falha, mas não sobre sua localização ou conseqüências. Como existe um intervalo de tempo entre a ocorrência da falha e sua detecção, se os componentes interagem antes da detecção do erro, as alterações devidas à falha podem ter-se propagado pelo sistema. Portanto, o **confinamento** está vinculado a ações que evitem esta propagação; e a **avaliação de danos** consiste em determinar a parte do sistema que foi corrompida. Como os erros se propagam através da comunicação entre os processos, todo o fluxo de informação entre os componentes do sistema deve ser examinado a fim de identificar entre quais processos ocorreu troca de informação e delimitar as conseqüências destas ações.

Uma vez que o erro foi detectado e sua extensão identificada, as alterações indevidas devem ser removidas, caso contrário o estado errôneo pode causar mau funcionamento no sistema futuramente. Na fase de **recuperação de erros**, o sistema deve tornar-se livre dos efeitos causados pelas falhas. Recuperar, pórtanto, é restabelecer um estado livre de erros após uma falha. Existem duas abordagens básicas para recuperação de erros. Se a natureza dos erros causados pelas falhas pode ser completamente avaliada, então é possível remover estes erros do processo e habilitá-lo a prosseguir. Esta técnica é conhecida como **recuperação por avanço**. Se não for possível avaliar a natureza das falhas e remover todos os erros, então o processo deve ser restaurado para um estado prévio (pelo qual o sistema já passou) livre de erros. Esta técnica é conhecida como **recuperação por retorno**.

A especificação e implementação de mecanismos de recuperação por retorno é mais simples do que a recuperação por avanço, pois independe do tipo de falha e dos erros causados pela mesma. Entretanto, o custo para restaurar um processo ou sistema a um estado

anterior pode ser bastante alto, levando à perda de desempenho, inaceitável em algumas aplicações. Podem também ocorrer falhas durante a execução dos procedimentos de retorno. Alguns componentes do sistema podem ainda ser irrecuperáveis.

A recuperação por avanço, por outro lado, possui custo menor durante a execução porque somente aquelas partes do sistema que não atingiram o valor esperado são corrigidas. Contudo, a implementação dos mecanismos depende da capacidade de antecipação das possíveis falhas, sendo as soluções particulares a cada caso.

O erro é primeiramente detectado, sua extensão avaliada e então removido, ficando assim o sistema livre de erros. Isto pode ser suficiente se sua causa foi uma falha transitória. Porém, se as falhas forem permanentes, então a causa do erro ainda estará no sistema, mesmo após a recuperação. Se o sistema retomar o caminho de execução anterior, novamente a mesma falha causará o mesmo erro. É essencial, portanto, que o componente falho seja identificado e reparado ou isolado. O **tratamento de falhas** implica na localização da falha e do reparo do sistema. Na localização, é diagnosticado o componente causador da falha. O reparo constituiu-se na substituição do componente. Outra opção é reconfigurar o sistema para não usar mais o componente falho. Uma vez que o sistema foi reparado, o serviço normal deve prosseguir.

2.2 Modelo de um sistema distribuído

Sistemas distribuídos (SDs) são compostos por um conjunto de processadores autônomos ligados através de uma rede de comunicação e dotados de software distribuído. O software permite a coordenação da atividade entre os diversos processadores e o compartilhamento dos recursos do sistema. Os usuários de um SD adequadamente projetado devem percebê-lo como um sistema único e integrado, embora ele possa estar implementado por vários computadores em diversos locais [COU94]. Assim, um sistema distribuído pode ser usado para implementar a abstração de um sistema computacional de uso geral, aparentemente composto por um único processador, com maior confiabilidade e longevidade do que um sistema monoprocessador. A preocupação com a ocorrência de falhas, entretanto, não deve ser imputada ao programador de aplicações.

Para simplificar, os sistemas distribuídos são descritos através de um modelo lógico compreendendo processos e canais entre processos [JAL94]. Processos são nodos no modelo, e os canais são as ligações entre eles. A rede física que os suporta é considerada como totalmente conectada no nível lógico. Os processos comunicam-se através de mensagens. É interessante observar que a autonomia e independência de cada componente do sistema acarreta independência das falhas de hardware, o que reforça a conveniência do uso de SDs, quando um comportamento robusto a falhas é desejado. Quando estas ocorrem nos processadores, que são elementos reais, provocam degradação gradual no modelo lógico. Esta degradação gradual é a propriedade fundamental dos sistemas distribuídos.

Algoritmos fundamentais para o enfoque distribuído são os de ordenação de eventos de Lamport [LAM78] e algoritmos para determinação do estado global (consistente) do sistema, de Lamport e Chandy [CHA85]. Os principais problemas para elaboração de

algoritmos distribuídos são: a ausência de memória comum, os atrasos aleatórios na comunicação e a necessidade de procedimentos de tolerância a falhas não previstos nos processadores e na rede de comunicação. A visão que os processadores têm sobre um SD ocorre exclusivamente através das mensagens que eles enviam e recebem. Basicamente, se estas mensagens estão de acordo com o protocolo especificado, os processadores envolvidos são considerados corretos.

Nos SDs, a redundância de hardware pode ser usada para assegurar a realocação de tarefas essenciais em caso de falha - deste ponto de vista, a tolerância a falhas pode ser tratada de forma muito mais eficiente nos sistemas distribuídos do que nos sistemas centralizados. Entretanto, a recuperação dos defeitos manifestados a nível de hardware e software, sem que ocorra a perda de dados, necessita de um cuidadoso projeto.

2.2.1 Sistemas síncronos e assíncronos

Uma divisão conceitual separando dois modelos extremos: sistemas síncronos e assíncronos [BIR96] auxilia no entendimento de questões relacionadas ao funcionamento prático dos sistemas distribuídos. No modelo assíncrono, não pode ser assumida nenhuma hipótese sobre a velocidade relativa dos sistemas de comunicações, processadores ou processos na rede. Uma mensagem enviada de um processo a outro pode ser entregue em tempo zero, enquanto que a próxima pode ter um atraso de anos. O modelo assíncrono pondera uma hipótese sobre tempo, mas não sobre falhas. Sua simplicidade permite ao projetista focalizar o interesse sobre as propriedades fundamentais do sistema, sem confundir a análise pela inclusão de um grande número de considerações práticas.

No modelo síncrono, há uma noção muito forte de tempo que os processos compartilham no sistema. Pode-se pensar na noção de um tempo marcado periodicamente; ao "compasso" deste tempo, os processos realizam mais uma rodada de atividades, composta por leitura e envio de mensagens, dentre outras atividades. Estas mensagens são entregues às aplicações, ou a outros processos, apenas no início da próxima rodada. Em geral, o modelo síncrono assume limites nos tempos da comunicação entre processos, nos tempos de relógio e de suas variações. Os sistemas reais não são exatamente síncronos, mas este modelo pode ser tomado por base para estabelecer limites mínimos para resolução de problemas.

No modelo lógico [JAL94], um sistema é dito **síncrono** se, enquanto o sistema estiver funcionando corretamente, ele sempre realizar seus objetivos dentro de um limite de tempo finito e conhecido; o sistema é dito **assíncrono** em qualquer outra situação. Um sistema síncrono tem canais de comunicação síncronos, nos quais o atraso máximo das mensagens é conhecido e limitado, e processadores síncronos, nos quais o tempo de execução de uma seqüência de instruções é limitado e conhecido. Em um sistema síncrono, podem ser empregados limites de tempo (*time-outs*) para a detecção de falhas.

Independentemente do modelo, a comunicação entre processos, realizada através da troca de mensagens, pode ser feita de forma síncrona ou assíncrona. Na modalidade assíncrona, a primitiva *receive* não especifica qualquer fonte e consome a primeira men-

sagem da fila de espera. Na comunicação síncrona, o emissor e o receptor sincronizam-se a cada troca de mensagens. *Communicating Sequential Processes* (CSP) [HOA78] e *Remote Procedure Call* (RPC) [NEL81] são exemplos para este tipo de comunicação.

Em um sistema distribuído, vários processos podem ser executados concorrentemente e cada processo realiza eventos. Os nodos no sistema têm relógios com valores diferentes. A sincronização de processos nestes sistemas é complexa devido à indisponibilidade de memória compartilhada. Um sistema operacional distribuído precisa estabelecer um ordenamento relativo entre os processos que estão sendo executados em computadores diferentes. O procedimento é baseado nas mensagens trocadas entre eles. Para a implementação são usados relógios lógicos: estes relógios atribuem tempos aos eventos no sistema. A atribuição é feita através de marcas de tempo (*timestamps* [LAM78]) que mantêm a consistência dos eventos com o ordenamento parcial dos processos definido anteriormente.

2.2.2 Redundância em sistemas distribuídos

Exemplos de técnicas de redundância, através das quais pode-se prover tolerância a falhas de hardware a custo relativamente baixo, são inúmeros. Servidores podem ser replicados. O hardware redundante necessário à tolerância a falhas pode ser alocado à execução de atividades não críticas, enquanto não ocorrem falhas. Já uma base de dados pode ser replicada em vários servidores para assegurar que os dados se mantêm acessíveis em caso de defeito de qualquer um dos servidores. Estes servidores podem ser projetados para detectar falhas em seus pares; assim, quando uma falha é detectada em um deles, os clientes podem ser redirecionados para os servidores remanescentes.

No software, a opção mais freqüente é o projeto de programas que permitam a recuperação ou a reconstrução de dados a partir da detecção de erros. Sem estes procedimentos, em caso de falhas, processamentos incompletos ou dados inconsistentes seriam possíveis. A recuperação é baseada na preservação de informações suficientes e necessárias, de modo que o sistema, após a ocorrência de uma falha, retorne à operação normal em uma posição correta do processo. Os principais problemas relacionados à aplicação da técnica, nos sistemas distribuídos, são: (a) o erro pode ter-se propagado, não havendo condições de retorno; (b) é difícil manter a coerência na interface sendo necessária a repetição de saídas do sistema; (c) é complexo implementar uma base de dados confiável, sem afetar significativamente o desempenho do sistema; (d) é necessário determinar o universo de informações para retorno; ou definir ou antecipar parâmetros para efetuar a correção.

Existem várias técnicas baseadas em redundância que podem ser usadas para prover tolerância a falhas nos sistemas distribuídos; entretanto, este texto ficará restrito ao contexto básico da recuperação.

2.2.3 Modelo de falhas e defeitos

As definições apresentadas por Cristian [CRI91] foram inicialmente usadas para caracterizar a atuação de servidores, mas podem ser generalizadas. Um componente

projetado para fornecer um dado serviço está **correto** enquanto ele se comporta de uma forma consistente com sua especificação, em resposta a um conjunto de entradas. Esta especificação também deve considerar o intervalo de tempo no qual a resposta deve ocorrer. Um **defeito** deste componente acontece quando ele não de comporta da maneira especificada. Este defeito pode ser de:

- **omissão**, quando o componente deixa de responder a uma dada entrada.
- **temporização,** quando a resposta do componente é funcionalmente correta mas ocorre fora do intervalo de tempo especificado. Tanto pode corresponder a uma resposta antecipada como a uma resposta tardia (defeitos de desempenho).
- **resposta**, quando o componente apresenta valor de saída incorreto ou realiza uma transição de estado incorretamente.
- **colapso** (*crash*), quando, após a primeira omissão, o sistema deixa de produzir saídas para as entradas subseqüentes, até que seja reinicializado.

Jalote [JAL94] apresenta uma outra categoria de falhas, que produz uma modalidade diferente de defeito: a falha bizantina. Uma **falha bizantina** faz com que o componente se comporte de maneira totalmente arbitrária. Esta categoria é mais genérica que as demais propostas por Cristian. Segundo Jalote, existe uma hierarquia entre os diferentes tipos de defeitos, sendo que colapso é o mais simples, mais restritivo e mais bem definido. Ele é englobado pelos defeitos de omissão, por sua vez englobado pelos de temporização, todos contidos no conjunto mais universal, de comportamento bizantino (ou malicioso). Os defeitos de resposta são englobados pelo modelo bizantino, mas são ortogonais aos demais. Na prática, nem sempre os efeitos se manifestam através de uma simples falta de resposta. Por esta razão, considera-se neste texto (assim como em boa parte das aplicações) que os sistemas se comportam de tal maneira que os defeitos correspondem a paradas totais ou parciais na operação e não por respostas alternando entre corretas/incorretas. Esta decisão não é meramente hipotética, mas encontra amparo em termos de implementação.

Nos sistemas computacionais distribuídos, os defeitos podem variar em sua manifestação, dependendo do tipo de serviço ou estrutura afetada pela falha. O resultado revela-se através do mau funcionamento dos processos ou através de problemas no armazenamento ou comunicação. Defeitos nos **processos** podem resultar em saídas incorretas, em desvio do estado do sistema de sua especificação ou em que o processo não consiga mais progredir. As causas podem ser erros tais como: *deadlocks*, ocorrência de *time-outs*, violação de proteções, entradas incorretas fornecidas pelo usuário, violações de consistência. As conseqüências podem ser o retorno a um estado anterior, a partir do qual o processo tenta refazer suas ações, ou o aborto do processo.

Considera-se um defeito do **sistema** quando o processador interrompe a execução devido a falhas de software ou de hardware. Estas falhas podem corresponder a situações afetando elementos do hardware como: parada do processador, incapacidade de resposta da memória ou falta de alimentação, ou a perdas de informações da base de dados. Os procedimentos subseqüentes dependerão das causas deste defeito. Quando existe parada total do sistema, por exemplo, na retomada da operação, o sistema é reinicializado a partir

de um estado correto, pré-definido em tempo de projeto. Neste tipo de situação, a informação armazenada na memória volátil e em registradores do processador é perdida. A recuperação trata principalmente deste tipo de ocorrência.

Os defeitos no **meio de armazenamento secundário** impedem a obtenção, parcial ou integral, dos dados armazenados. Estes defeitos manifestam-se a partir de um erro de paridade, um problema mecânico ou da existência de partículas de poeira no meio físico. Como conseqüência, os dados armazenados ficam corrompidos e precisam ser reconstruídos a partir de uma versão anterior. Uma forma eficiente para lidar com este tipo de defeitos é o uso de sistemas de "espelhamento" de discos.

Defeitos no **meio de comunicação** caracterizam-se pela incapacidade de comunicação entre nodos operacionais pertencentes à rede. As causas mais comuns estão nos nodos de chaveamento, incluindo defeitos do nodo e meio de armazenamento, ou no sistema de comunicação, incluindo rupturas físicas e ruído nos canais. Conseqüências possíveis são perda ou recebimento de mensagens corrompidas, particionamento da rede ou perda completa das atividades de comunicação. Transmissões envolvendo quantidade excessiva de mensagens, bem como atrasos no processamento destas devido a alguma sobrecarga, revelam-se como defeitos de desempenho de comunicação.

2.3 Conseqüências das falhas em sistemas distribuídos

A recuperação tem o grau de complexidade bastante aumentado se, ao invés de atuar com processos independentes, lidar com processos concorrentes. Neste segundo cenário, é preciso avaliar não só processos isolados mas também suas interrelações com os demais. Em sistemas distribuídos, o estado completo de cada processo pode ser salvo periodicamente através do estabelecimento de pontos de recuperação (PRs). Após falha de um dos processos cooperantes, inicia-se o processo de recuperação, que consiste no restabelecimento do estado salvo e na retomada da execução dos processos a partir deste estado. Os efeitos causados a outros processos após o estabelecimento do estado base para a recuperação, devido à informação trocada com eles, deverão ser desfeitos. Esta seção aborda alguns dos problemas que podem ocorrer como resultado destas interações.

Considere-se os três processos concorrentes, x, y e z, da figura 1. Cada símbolo [representa um PR estabelecido; este é o único tipo de informação armazenada de forma permanente, assegurando que não será perdida na ocorrência de falhas. As setas representam trocas de mensagens entre os processos. Suponha-se que y falhe após enviar a mensagem m e retorne para y_2, em conseqüência. Neste caso, x_3 recebeu m, mas y_2 não tem registro de que a enviou, o que corresponde a um estado inconsistente. Diz-se, então que m é uma **mensagem órfã** e o processo x deverá retornar para x_2 por questões de consistência [SIN94]. Suponha-se agora que, após algum tempo de processamento, z não obtenha sucesso em um teste realizado, indicando a possível ocorrência de uma falha. Este processo não chega a estabelecer novo PR e precisa retornar para z_2. A mensagem que z enviou para y, além de ser órfã, pode estar incorreta, afetada pela mesma falha que impediu z de

prosseguir. O processo y retorna para y_1 e, a fim de eliminar as mensagens órfãs, obriga x e z a retornarem respectivamente para x_1 e z_1. Assim, todos os processos voltam para seus primeiros pontos de recuperação, x_1, y_1, z_1. Esta situação, onde o retorno de um processo causa o retorno encadeado dos demais, é conhecida como **efeito dominó**.

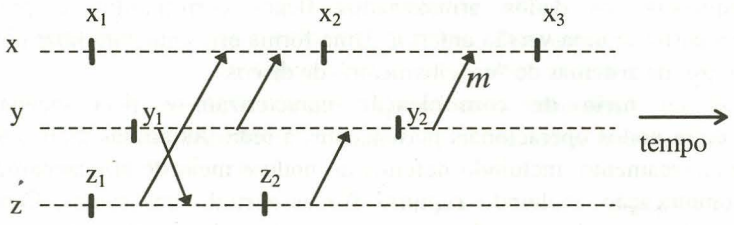

Figura 1: Execução de três processos concorrentes

Em outra hipótese, se ocorrer uma falha e for necessário retornar para y_1 e z_1, y_1 terá o registro de envio da mensagem m mas z_2 não terá registro de recebimento. Diz-se que m é uma **mensagem perdida**. Esta situação também ocorre quando há falha no canal de comunicação entre os processos. Não é possível distinguir quando a mensagem foi perdida por falha no processo ou no meio de comunicação.

O grande problema das mensagens órfãs e das mensagens perdidas é seu potencial de causarem o efeito dominó. Juntas podem causar o retorno de um processo ao seu estado inicial a cada falha. Assim, se ocorrerem falhas freqüentemente, um processo corre o risco de não conseguir chegar ao seu término devido aos sucessivos reinícios. Existem políticas para tomada de pontos de recuperação em momentos adequados que minimizam a quantidade de computação desfeita. Este assunto será abordado nas próximas seções.

Livelock é a situação na qual uma falha pode causar um número infinito de retornos, impedindo o sistema de fazer progresso [KOO87]. Na figura 2, observa-se a atividade de dois processos x e y. A falha de y ocorre antes do recebimento da mensagem m_2 de x. Quando y retorna para y_1, não há registro do envio da mensagem m_1, assim x deve retornar para x_1. Quando y se recupera, ele envia m_1 e recebe m_2. Entretanto, como x retornou, não há registro de envio de m_2 e, assim, y retornará pela segunda vez. Esta situação pode repetir-se indefinidamente, impedindo o progresso da computação.

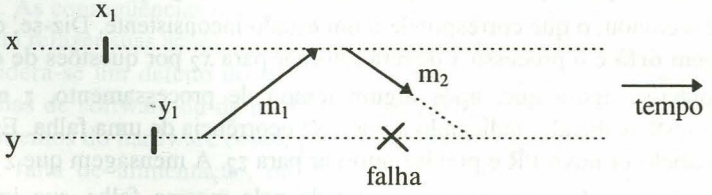

Figura 2: Situação de *Livelock*

3 Procedimentos prévios à ocorrência de falhas

Considerando que a maior parte das falhas corresponde a falhas transitórias ou intermitentes, o procedimento mais simples de recuperação seria uma mera repetição da computação executada. Entretanto, a conseqüência evidente deste procedimento é a queda do desempenho a cada nova tentativa. Se o principal objetivo dos sistemas fosse a busca de alto desempenho, esta constatação já serviria como motivo ao estudo de técnicas mais eficientes. Mas os problemas não se concentram apenas neste aspecto.

Em sistemas distribuídos, diversas atividades são realizadas por processos concorrentes, que não estão apenas disputando recursos mas estão também compartilhando informações. Resultados errados gerados por um processo podem ser espalhados pelo sistema, causar danos em outros processos ou podem, até mesmo, provocar a necessidade de uma reinicialização global, com a perda dos resultados. Para estes casos é interessante que o sistema ofereça alternativas de recuperação cuja penalidade não seja tão severa. As alternativas, conforme caracterizado na seção 2.1, recaem sobre os seguintes enfoques:

- recuperação por avanço: prosseguindo na execução do programa, a partir de onde o erro foi detectado, usando mecanismos alternativos para assegurar a correção;
- recuperação por retorno: voltando, o mínimo possível o programa a um estado anterior correto, para a partir deste ponto retomar a operação.

Sua utilização não é mutuamente exclusiva, mas do ponto de vista conceitual, o tratamento com este ponto de vista simplifica as colocações.

A implementação da recuperação por avanço utiliza-se de métodos particulares à aplicação e às hipóteses de erros que possam ocorrer no sistema; tratam-se de soluções do tipo *ad hoc* e não serão tratadas nesse texto. A implementação da recuperação por retorno pode ser realizada através de mecanismos genéricos, que serão apresentados nas próximas seções. Serão comentados os enfoques baseados no armazenamento de operações ou estados, bem como a questão de armazenamento estável, da ordem empregada no ordenamento dos passos de salvamento e conseqüências sobre a recuperação. O enfoque do armazenamento de estados será usado para caracterizar PRs e as técnicas de tomada destes pontos, além do impacto destas decisões sobre o desempenho dos sistemas.

3.1 Enfoques básicos para a recuperação por retorno

Há dois enfoques principais para a recuperação por retorno: um é baseado em estados e o outro em operações. Podem ser empregados de forma complementar.

O enfoque mais comum é baseado no **estado** do sistema e se dá através do armazenamento de informações referentes a ele em pontos discretos da execução - pontos de recuperação, PRs (*checkpoints*[1] ou *rollback points*). Estas informações devem ser as

[1] *Checkpoint*, em inglês, identifica um posto de controle de fronteira. Esta figura é importante para compreender-se o sentido do termo: há ações de verificação anteriores ao armazenamento e à continuidade da computação.

necessárias para assegurar o retorno a estes pontos em caso de falha no nodo, e recomeçar a computação como se não tivesse ocorrido qualquer falha.

O estado varia de um sistema para outro; em um processador, por exemplo, envolve registradores, o contador de programa, o estado da *cache*, e possivelmente a parte da memória que foi alterada desde o salvamento do último ponto de recuperação. Estes PRs devem ser guardados em meio de armazenamento confiável, isto é, que não perde dados mesmo na presença de defeitos do sistema, denominado **memória estável** [SIN94]. Sua forma mais tradicional de implementação é em disco protegido por códigos de correção de erros (ECC) e por mecanismos de duplicação ("espelhamento", por exemplo). Por razões de desempenho, sugere-se hoje o uso de memória principal ou registradores também protegidos por ECC, duplicação e bateria [PRA96].

O outro enfoque é baseado em **operações**: as modificações efetuadas sobre o estado do processo são gravadas, em memória estável, de tal forma que o estado anterior à falha possa ser restaurado desfazendo as alterações realizadas. O conjunto das informações referentes à atividade do sistema é denominado de *audit trail* ou ***log***. Estas informações são suficientes para operações de restauração de objetos (processos, por exemplo) antigos ou novos (*undo* e *redo*). Quando o processo ou a transação é completada com sucesso, as mudanças tornam-se permanentes. Problemas podem sobrevir em caso de falha após uma operação de atualização mas antes do armazenamento do *log*; entretanto, o uso de protocolos adequados através dos quais a atualização de objetos seja realizada apenas após a gravação do *log* (*write-ahead-log* é um exemplo) resolve o problema [SIN94].

Em sistemas independentes, há diversas opções para implementar recuperação por retorno através de pontos de recuperação [PRA96]. A maior parte dos métodos [HUN87, WU90, BOW92] pressupõe de que o sistema opera com duas classes de dados diferenciados: os dados do PR e dados ativos. Os dados do PR correspondem ao estado gravado mais recentemente; devem estar armazenados de forma estável. Os dados ativos são os que foram utilizados e eventualmente modificados após a tomada do PR. Estes dados só serão incorporados à informação permanente quando ocorrer o próximo PR.

As opções de implementação variam quanto aos mecanismos de armazenamento e identificação das informações estáveis / sob modificação, na definição do estabelecimento dos pontos de recuperação e na própria arquitetura do sistema, envolvendo recursos disponíveis. Entretanto, a política essencial de funcionamento não se modifica.

Estas variações dos mecanismos terão impacto no funcionamento do sistema; não se pode esquecer que freqüência de tomada de PRs depende das taxas de falhas dos componentes do sistema e da habilidade deste para se recuperar.

3.2 Recuperação em sistemas concorrentes

Checkpointing define a ação de estabelecer pontos de recuperação; em sistemas distribuídos envolve o estabelecimento de pontos de recuperação por todos os processos ou pelo menos por um conjunto de processos que interajam entre si. Tipicamente, todos os nodos salvam seus estados locais, que são conhecidos como *PRs locais*. O conjunto dos

PRs locais, um de cada nodo, forma coletivamente um *PR global.* Ao longo deste texto são usados como sinônimos os termos: tomar, criar ou estabelecer pontos de recuperação como traduções de *checkpointing*.

Para evitar o efeito dominó, pode ser estabelecido um conjunto de PRs locais de tal forma que não haja fluxo de informação entre qualquer par de processos no conjunto, durante o intervalo entre um PR e outro. Tal conjunto de PRs é conhecido como uma linha de recuperação ou conjunto fortemente consistente de pontos de recuperação. Na figura 3, o conjunto $\{x_1, y_1, z_1\}$ forma um conjunto fortemente consistente de PRs, pois não existe troca de mensagens entre os processos durante a tomada dos PRs. Este conjunto corresponde a um estado global consistente. O conjunto de PRs $\{x_2, y_2, z_2\}$ não é um conjunto consistente, pois existe troca de mensagens entre os processos durante a tomada dos PRs. Os sistemas que não estabelecem um conjunto fortemente consistente de PRs têm que lidar com mensagens perdidas durante o retorno.

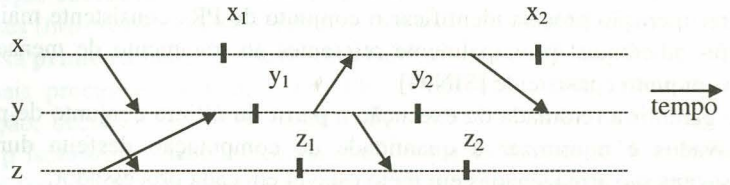

Figura 3: Conjunto consistente de PRs

A forma pela qual são estabelecidos os PRs em processos concorrentes de um sistema, determina uma classificação alternativa das técnicas de recuperação, caracterizando-as como síncronas ou assíncronas.

No **enfoque síncrono** de estabelecimento de PRs, todos os nodos são coordenados para estabelecer os respectivos PRs a partir de um dado momento, não havendo troca de dados durante o intervalo de tomada dos pontos. A única informação autorizada envolve mensagens referentes ao controle do algoritmo. Assim, quando um processo recebe a requisição de tomada de PR de um processo *Pi*, ele susta o envio de mensagens de dados a outros processos até a conclusão (com sucesso ou não) da operação de estabelecimento conjunto do PR. Esta medida garante que todos os PRs tomados no método síncrono são consistentes.

Se cada processo estabelece um PR após enviar cada mensagem, o conjunto de PRs mais recente é sempre consistente. Entretanto, ele não é fortemente consistente. O conjunto dos últimos PRs é consistente porque o último PR de cada processo corresponde ao estado onde todas as mensagens dadas como recebidas foram mesmo recebidas. Assim o retorno de um processo para seu último PR não resultaria em mensagens órfãs e não causaria um estado inconsistente no sistema. Como tomar um PR após cada mensagem enviada é caro, para reduzir o custo deste método, estabelece-se um PR a cada k (k>1) mensagens enviadas. O preço desta decisão é a possibilidade de sofrer o "efeito dominó", com o retorno global ao último PR fortemente consistente.

Pode-se dizer que, enquanto a atividade de estabelecimento dos PRs síncronos simplifica a recuperação (pois possui sempre um conjunto consistente de PRs), possui as seguintes desvantagens: (a) mensagens de controle são trocadas pelo algoritmo para o estabelecimento dos PRs, (b) é necessário esperar por sincronização durante as operações. Logo, se a taxa de falhas for baixa, o método síncrono é uma opção pessimista, pois ocasiona sobrecarga constante no sistema.

No **enfoque assíncrono**, cada processo estabelece os seus próprios pontos de forma independente com relação aos demais processos, no momento em que for mais conveniente para si. Assim, o estabelecimento dos pontos de recuperação locais não é coordenado. Esta opção é, em geral, mais eficiente durante a operação normal - mas é mais complexa em caso de falhas. Como não há garantia de que os PRs locais formarão um conjunto consistente, o sistema precisa lidar com inconsistências possivelmente resultantes. Portanto, o algoritmo de recuperação precisa identificar o conjunto de PRs consistente mais recente ou realizar tarefas adicionais, principalmente referentes ao tratamento de mensagens, de forma a tornar o conjunto consistente [SIN94].

A fim de garantir a retomada da execução a partir do último conjunto de pontos de recuperação gravados e minimizar a quantidade de computação desfeita durante um retorno, as mensagens são armazenadas em meio estável em cada processador:

- a partir de um paradigma pessimista, uma mensagem é armazenada antes de ser processada. Isto torna mais lenta a computação durante a operação normal.
- a partir de um paradigma otimista, os processadores continuam a sua computação e as mensagens recebidas são guardadas na memória volátil e transferidas em intervalos para a memória estável. Durante o processamento normal, este método é mais rápido.

O risco de adotar a hipótese otimista é perder algumas informações, caso ocorra uma falha antes da informação ser transferida para a memória estável, o que não acontece na abordagem pessimista. Contudo, a abordagem pessimista não é adequada para sistemas em que falhas não sejam freqüentes, uma vez que o armazenamento de cada mensagem antes do processamento degrada demasiadamente o desempenho do sistema.

4 Algoritmos de recuperação por retorno

As ações de recuperação por retorno dependem essencialmente das informações armazenadas anteriormente (como pontos de recuperação) e da forma pela qual este procedimento foi realizado. São detalhadas as ações necessárias para implementar os procedimentos de recuperação a partir de dois algoritmos clássicos, bastante representativos de suas respectivas classes, que operam a partir de PRs síncronos, em um caso, e PRs assíncronos, no outro. O algoritmo de Koo e Toueg [KOO87] ilustra o método síncrono, e o método de Strom e Yemini [STR85, STR88] é representativo da recuperação assíncrona.

4.1 Algoritmo de Koo

O algoritmo de Koo e Toueg [KOO87] adota tomada de pontos de recuperação distribuída e coordenada. O algoritmo suporta a ocorrência de falhas durante a execução do protocolo, mas assume que o sistema apresenta as seguintes características: processos comunicam-se por troca de mensagens através de canais de comunicação; os canais operam na modalidade de FIFO (*first in first out*); e os defeitos de comunicação não particionam a rede.

4.1.1 Estabelecimento de pontos de recuperação

O algoritmo de Koo armazena duas classes de PRs: permanentes e temporários (*tentative checkpoints*). Um PR permanente não pode ser desfeito; um PR temporário pode ser desfeito ou modificado para se tornar permanente. Para estabelecer pontos de recuperação coordenados em todos os processos, o algoritmo implementa um protocolo de duas fases (*two-phase commit*) conforme mostrado na figura 4.

Na **primeira fase**, o processo iniciador **q** cria um PR temporário e solicita que todos os demais processos também criem PRs temporários. Um processo **p**, que recebeu a solicitação, decide estabelecer ou não um PR e comunica sua decisão ao iniciador. A seguir, **p** isola-se de qualquer comunicação com os demais processos até a conclusão da segunda fase. Se **q** é informado que todos os processos criaram PRs temporários, então **q** define a conversão de todos PRs temporários em permanentes; em caso contrário, **q** decide que os PRs temporários criados devem ser descartados. Na **segunda fase** a decisão de **q** (tornar permanente ou descartar) é propagada para todos os processos.

Essa abordagem simples assegura que o conjunto de pontos de recuperação mais recente é sempre consistente, uma vez que todos processos estabeleceram PRs permanentes, ou nenhum deles o fez (permanecem com PRs anteriores). Não há mensagens órfãs no último estado consistente armazenado. O algoritmo não assegura a inexistência de mensagens perdidas, mas estas ocorrências devem ser tratadas pelo protocolo de comunicação.

Figura 4: Protocolo de duas fases para criação de PRs

Entretanto, para obter um conjunto consistente de PRs, não é necessário que todos os processos armazenem seus estados a cada solicitação. Novos pontos de recuperação de alguns processos junto a antigos PRs de outros processos podem também originar um estado consistente. Assim, para assegurar que apenas um *número mínimo de processos* estabeleça PRs, a abordagem simples é levemente modificada. Um processo ***p*** cria um PR temporário (atendendo a uma solicitação de ***q***) se ***q*** recebeu alguma mensagem enviada de ***p*** depois do seu último PR. Ou seja, ***p*** só vai criar um PR se existe o risco de mensagens enviadas por ***p*** tornarem-se órfãs. Desta forma, apenas processos que enviaram mensagens a ***q*** após o último PR necessitam estabelecer outro novo quando solicitados por ***q***.

Todo processo executando o algoritmo de Koo possui as estruturas de dados internas mostradas na tabela 1.

Tabela 1 Estruturas de dados para tomada de PRs

willing-to-ckpt	desejo de criar um ponto de recuperação
ckpt-cohortq	processos dos quais ***q*** recebeu alguma mensagem desde o último PR
m.seq	número de seqüência crescente atribuído a toda mensagem enviada
last-recdq(p)	número de seqüência da última mensagem que ***q*** recebeu de ***p*** após ***q*** criar um PR permanente ou temporário
first-sentq(p)	número de seqüência da primeira mensagem que ***q*** enviou a ***p*** após ***q*** ter realizado seu último PR

A seguir são descritas as ações definidas pelo algoritmo para o processo iniciador ***q***: o processo ***q*** começa o algoritmo de tomada de PR através do estabelecimento de um PR temporário e envia uma mensagem de solicitação de PR para os processos que se comunicaram com ele ($ckpt\text{-}cohort_q$). Esta mensagem (figura 5) é composta por uma ordem de criação de PR temporário (TAKE A TENTATIVE CHECKPOINT) seguida do número de seqüência $last\text{-}recd_q(p)$. O iniciador coleta então a resposta de todos os processos para os quais enviou a mensagem de estabelecimento do PR temporário. Caso todas as respostas dos processos em $ckpt\text{-}cohort_q$ sejam positivas, o iniciador ***q*** ordena aos processos que transformem os seus PRs temporários em permanentes. Caso contrário, os PR temporários devem ser desfeitos.

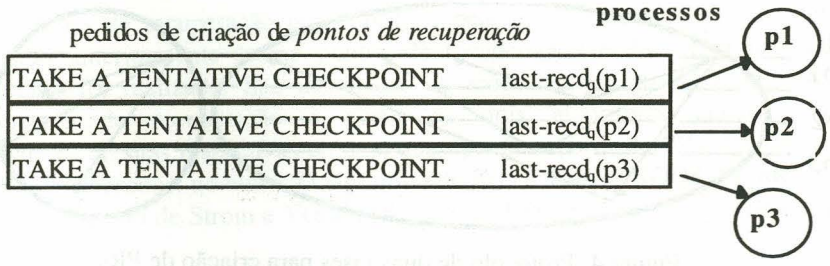

Figura 5: Mensagens de pedido de PRs temporários

As ações definidas para os demais processos envolvidos (processo ***p*** genérico) são: um processo ***p***, recebendo a solicitação de criação de PR temporário, envia essa solicitação a todos os processos contidos em seu $ckpt\text{-}cohort_p$. Dessa forma, todos os processos que potencialmente podem afetar o estado de ***q*** recebem a solicitação de tomar PRs. Cada processo ***p*** verifica se precisa estabelecer um PR temporário testando se a sua variável *willing-to-ckpt* é positiva e se $last\text{-}recd_q(p)$ (o número de seqüência que chegou junto com o pedido de criação de PR temporário indicando a última mensagem recebida por ***q*** de ***p***) é maior ou igual a $first\text{-}sent_p(q)$ (o número de seqüência da primeira mensagem que ***p*** enviou para ***q*** após ***p*** ter realizado seu último PR). O processo também espera resposta de todos os processos pertencentes ao seu $ckpt\text{-}cohort_p$. Se ***p*** não deseja criar um PR ou se algum dos processos com os quais se comunicou não deseja criar um PR, então *willing-to-ckpt* torna-se negativa, caso contrário torna-se positiva. Finalmente ***p*** envia *willing-to-ckpt* para ***q***.

4.1.2 Procedimentos para recuperação de processos

Um processo ***q*** falhou temporariamente; ao retornar, solicita a execução do algoritmo de recuperação. A abordagem mais simples é retornar, um a um, todos os processos para o seu último PR permanente, garantindo dessa forma o retorno do sistema a um estado consistente. Entretanto, pode não ter ocorrido comunicação entre o processo que falhou e os demais depois do seu último PR; assim nenhum outro necessitaria retornar. A abordagem simples, nesse caso, é por demais onerosa, forçando processos a retornarem sem necessidade. Visando reduzir o número de retornos necessários, o algoritmo proposto por Koo envolve apenas o número mínimo de processos. Ou seja, apenas aqueles que se comunicaram com ***q*** desde o último PR (figura 6) são solicitados a retornar ao último ponto de recuperação permanente.

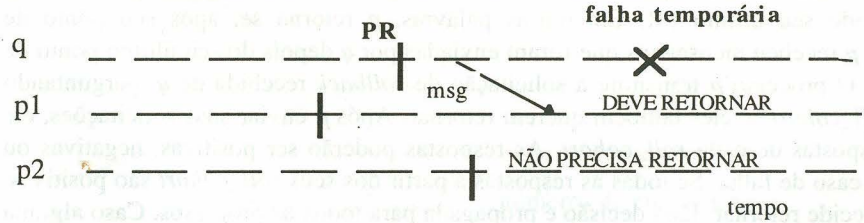

Figura 6: Exemplo de retorno com número mínimo de processos

Na figura 6, o processo ***q*** enviou mensagem para ***p1*** após o seu último ponto de recuperação. Com o retorno de ***q***, a mensagem enviada a ***p1*** torna-se órfã e ***p1*** deve portanto retornar ao seu último PR. Entretanto ***p2*** não precisa retornar, pois não recebeu nenhuma mensagem de ***q*** e portanto não processou nenhuma mensagem órfã.

Todo processo executando o algoritmo de retorno de Koo possui adicionalmente as estruturas de dados mostradas na tabela 2. A recuperação opera também em duas fases. Na primeira fase, o iniciador ***q*** solicita que todos os processos retornem e recomecem o

processamento a partir de seus últimos pontos de recuperação. O processo ***q*** decide reiniciar todos os processos se, e somente se, eles respondem positivamente à requisição. Na segunda fase, a decisão de ***q*** é propagada e executada pelos demais processos envolvidos na recuperação.

Tabela 2 Estruturas de dados para recuperação

willing-to-roll	desejo de retornar (*rollback*)
$roll\text{-}cohort_q$	processos que receberam mensagens de ***q*** desde o último PR
$last\text{-}sent_q(p)$	número de seqüência da última mensagem que ***q*** enviou a ***p*** antes de ***q*** ter realizado seu último PR

O retorno (*rollback*) de um processo ***q*** força outro processo a voltar para trás somente se a volta de ***q*** desfaz a emissão de uma mensagem para ***p***. O processo ***p*** determina se deve retornar após receber uma solicitação de ***q***

O processo iniciador ***q*** dá andamento ao algoritmo de recuperação operando conforme segue: o iniciador envia uma solicitação de recuperação para os processos que receberam mensagens de ***q*** ($roll\text{-}cohort_q$). Esta solicitação é composta por uma ordem de recuperação (PREPARE TO ROLLBACK) seguida do número de seqüência $last\text{-}sent_q(p)$ da última mensagem que ***q*** enviou para ***p*** antes de ***q*** ter realizado seu último PR.

As ações em um processo ***p*** ocorrem conforme segue: o processo ***p*** é um *roll_cohort* de ***q***, se ***q*** emitiu mensagens para ele. Todos os processos ***p*** possuem uma variável *willing-to-roll* para denotar seu desejo de recuperação. O processo ***p*** retorna ao último ponto de recuperação permanente apenas se $last\text{-}recd_q(p) > last\text{-}sent_q(p)$. Ou seja, ***p*** retorna apenas se o número de seqüência da última mensagem recebida de ***q*** depois de seu último PR é maior do que o número de seqüência da última mensagem enviada de ***q*** para ***p*** antes de ***q*** ter estabelecido seu último PR. Em outras palavras, ***p*** retorna se, após seu ponto de recuperação, ***p*** recebeu mensagens que foram enviadas por ***q*** depois do seu último ponto de recuperação. O processo ***p*** transmite a solicitação de *rollback* recebida de ***q***, perguntando para seus *roll_cohort* se eles também querem retornar. Após ***p*** enviar suas solicitações, ele espera as respostas de cada *roll_cohort*. As respostas poderão ser positivas, negativas ou omitidas em caso de falha. Se todas as respostas a partir dos seus *roll_cohort* são positivas, o iniciador decide retornar. Esta decisão é propagada para todos os processos. Caso alguma falha tenha impedido a chegada de alguma decisão ao processo ***p***, ele deve ficar bloqueado até descobrir a decisão do iniciador.

4.2 Método de Strom e Yemini

Um dos métodos mais tradicionais para recuperação assíncrona foi proposta por Strom e Yemini [STR85, STR88]. O método [CAL97] usa armazenamento otimista de mensagens, com cada processo mantendo um rastro de dependência dos estados dos processos com os quais se comunica. Como resultado, é possível para um dado processo ***p1*** detectar que foram feitas computações que dependem de um processo ***p2*** que falhou.

Um estado é restaurado a partir do PR mais recente gravado na memória estável. O processo é então reexecutado a partir da seqüência de mensagens salvas também na memória estável.

4.2.1 Estrutura do sistema

O sistema é visto externamente como uma única máquina que se comunica através de canais (conceito de **máquina lógica**). Internamente, esta máquina contém múltiplas unidades de recuperação (RU), as quais comunicam-se através de troca de mensagens. Processos são vinculados a RUs particulares e são elas que falham e se recuperam.

Diz-se que um processo inicia um **intervalo de estado** no momento em que recebe uma mensagem. Todas as computações que forem feitas até a chegada da próxima mensagem pertencem ao intervalo de estado corrente. Durante um intervalo de estado, uma RU pode gerar várias mensagens de saída destinadas a outras RUs. Quando a RU iniciar o processamento da próxima mensagem de entrada, iniciará o próximo intervalo de estado.

Uma RU pode retornar a um estado anterior para recuperar-se de sua própria falha ou para responder à falha de outra RU. Como cada mensagem recebe um número seqüencial, os números das mensagens poderão ser reutilizados ao se repetir o processamento. Para continuar a ter uma única forma de identificar os intervalos, cada mensagem terá um **índice de estado** representado por [i,u] onde u é o número da mensagem e i é o **número de encarnação**, que é incrementado a cada vez que acontece um defeito e a RU reinicia sua execução como exemplificado através de uma situação hipotética considerada na figura 7. Nesta situação hipotética, o processo foi obrigado a reenviar a mensagem 3, pois a falha aconteceu antes desta mensagem ser recebida. Observa-se que o número da mensagem enviada não muda, sendo alterado apenas o número de encarnação que é incrementado a cada falha.

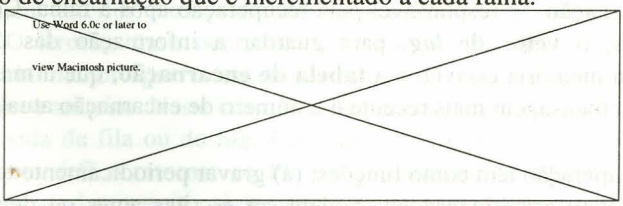

Figura 7: Mensagens e números de encarnação

Cada RU periodicamente grava seus PRs e as mensagens de entrada na memória estável. Esta escrita não é sincronizada com a comunicação. Uma RU possui vários componentes, ilustrados na figura 8 e descritos a seguir.

- ***Half-sessions* de entrada e saída** (HS) - são responsáveis pelo envio e recebimento das mensagens. No remetente, criam os números de seqüência de mensagem e guardam as mensagens até receberem a notificação de recebimento. No destinatário, mantêm os números de seqüência e encarnação esperados e comparam-nos com o número da mensagem recebida. Em operação normal (com número coincidente com o

esperado), a mensagem é passada para o componente de junção e o número de seqüência é incrementado.

- **Componente de junção** (MERGE) - estabelece uma ordem de processamento para as mensagens de acordo com os números de seqüência e encarnação.

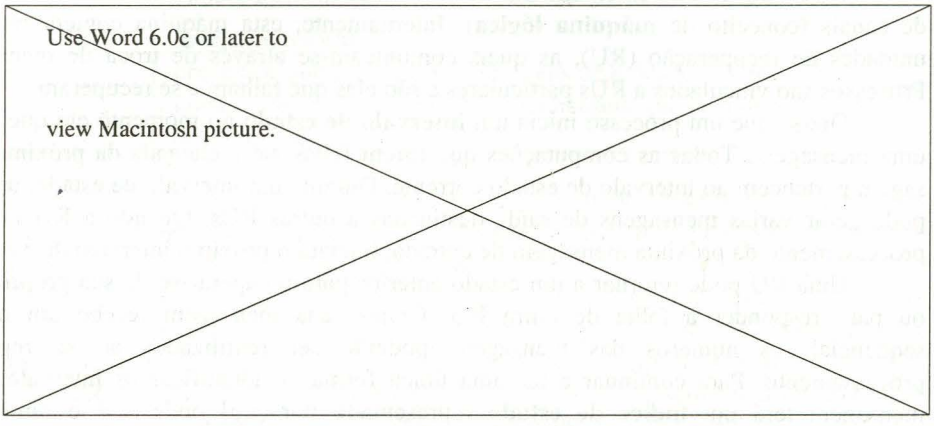

Figura 8: Componentes de uma RU

- **Componente de processamento** - parte da RU onde o processo é executado. Contém o **vetor de dependências**, que guarda a identificação da RU da qual depende (isto é, de quem recebeu mensagens), bem como o intervalo de estado e número de encarnação no qual a mensagem foi recebida.
- **Gerenciador de recuperação** - é responsável pela recuperação após a falha. Contém, como sub-componentes, o **vetor de *log***, para guardar a informação das últimas mensagens gravadas em memória estável; e a **tabela de encarnação**, que armazena o número de seqüência de mensagem mais recente e o número de encarnação atual.

O gerenciador de recuperação tem como funções: (a) gravar periodicamente todas as mensagens na memória estável; estas mensagens podem ser escritas antes ou depois de serem processadas pelo componente processador, pois não há sincronização; (b) manter o vetor de *log* para cada RU; este vetor contém a informação da validade ou não das operações da RU; (c) recuperar após a falha; (d) descartar os PRs que não são mais necessários; (e) incrementar as tabelas de encarnação, após a recuperação de uma falha.

4.2.2 Funcionamento do sistema em operação normal

Cada processo guarda uma trilha de dependências dos outros processos (mais precisamente, seus intervalos de estado) [JAL94]. A cada mensagem enviada, durante o processamento normal, também é enviado o vetor de dependências, como forma de garantir que os processos sempre terão conhecimento exato dos intervalos de estado dos quais dependem.

Periodicamente, o estado completo do processo e as mensagens são gravados na memória estável, durante a operação normal, mas pode haver descarte de um PR ***c***, se outro PR ***c'*** foi estabelecido após ***c***. Um processo nunca precisará retornar para um ponto entre ***c*** e ***c'***, uma vez que, se ***c'*** foi estabelecido, todas as mensagens enviadas entre ***c*** e ***c'*** já estão gravadas em memória estável por seus destinatários.

4.2.3 Operação de recuperação

Em caso de defeito de um processo, ele se recupera restaurando o seu PR mais recente. Sendo defeito de sistema em determinado nodo, o processo reiniciará de um nodo diferente. Tratando a falha, o processo refaz o *log* e reexecuta a computação perdida. Ao término do *log*, o processo incrementa seu número de encarnação e continua o processamento normal.

A recuperação do processo que falhou, entretanto, não é tão simples. Ele precisa lidar com três situações: a perda de mensagens recebidas mas ainda não salvas em memória estável, a repetição do envio de mensagens e a possibilidade de outros processos terem recebido mensagens que agora são órfãs.

O primeiro caso, perda de mensagens, é detectado pelo recebimento uma mensagem com número de seqüência maior que o esperado, por exemplo, com um número de encarnação maior do que o da mensagem anterior. Como a mensagem ainda não está na memória estável, o processo verifica quem a enviou em seu vetor de dependências e pede retransmissão.

Mensagens duplicadas podem ser detectadas pelos números de seqüência de mensagem e números de encarnação que serão inferiores aos esperados. Uma mensagem é duplicada se já foi recebida alguma mensagem com o par de números igual ao de uma mensagem anterior. Neste caso, é só descartá-la.

O terceiro caso ocorre se um processo recebe uma mensagem com número de seqüência menor que o esperado, mas com número de encarnação maior que o da mensagem anterior. Neste caso, se a mensagem ainda não foi processada, ela simplesmente é removida da fila ou do *log*. Caso contrário, é preciso forçar o processo que originou a mensagem órfã a retornar ao PR prévio, anterior ao processamento da mensagem. Para descobrir qual processo originou esta mensagem, basta observar o vetor de dependências que a acompanha.

Na figura 9, é apresentada a suposta execução de dois processos interdependentes ***Pi*** e ***Pj***. Por hipótese, ***Pi*** falha no tempo *t1*, quando a ação *(6)* e as mensagens processadas até *m5* ainda não foram gravadas. Na recuperação, ***Pi*** retorna para ***Ci*** e repete as mensagens até *(5)*. Então incrementará seu número de encarnação para 1 e reiniciará a execução. Quando executar *send(Pj,M)*, ***M*** será enviada a ***Pj*** com o mesmo número de seqüência de antes, mas com um número de encarnação maior. Assim, ***Pj*** retornará a ***Cj*** para desfazer o efeito da mensagem órfã.

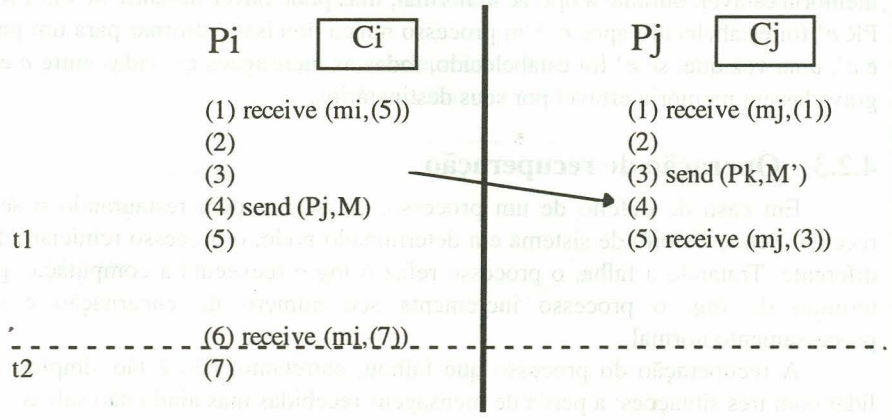

Figura 9: Exemplo do método de Strom e Yemini

Entretanto, se ***Pi*** falhar em *t2*, a recuperação será mais fácil. Em *t2* já foram gravadas as mensagens até *(7)*. Assim, quando repetir a execução, ter-se-á *send(Pj, M)* com o mesmo número de seqüência e mesmo número de encarnação. ***M*** será descartada por ***Pj*** e não será necessário retornar.

4.2.4 Vantagens e desvantagens do método

Este método é dos mais tradicionais em recuperação assíncrona, pois foi uma das primeiras propostas formuladas para o tema. Suas principais vantagens são a sua independência de aplicação ou sistema e a facilidade em detectar mensagens duplicadas ou órfãs, de acordo com os números de encarnação e índices de estado. Esta facilidade, entretanto, tem um preço: a grande quantidade de informação enviada juntamente com a mensagem, ou seja, a necessidade de enviar o vetor de dependências com cada mensagem. Este vetor de dependências só será descartado quando for tomado o próximo PR.

5 Recuperação em sistemas operacionais distribuídos

A seguir, sistemas operacionais comerciais com características de distribuição e paralelismo ilustram as técnicas de tomada de pontos de recuperação. São enfatizados os mecanismos incluídos nesses sistemas para facilitar o processo de recuperação, pressupondo-se conhecimentos prévios das caracteríticas básicas dos sistemas.

Targon/32 e Mach são sistemas operacionais para arquiteturas distribuídas. GUARDIAN90 foi desenvolvido para arquiteturas paralelas fracamente acopladas. Todas estas arquiteturas são baseadas em troca de mensagens. Os sistemas citados realizam recuperação por retorno baseada no armazenamento de pontos de recuperação e no conceito de pares de processos (tarefas no caso do Mach), sendo cada par formado por um processo primário e seu **reserva**. O mecanismo de processos reserva serve de instrumento para a manutenção da integridade dos processos. O primário envia mensagens com

informações do ponto de recuperação ao reserva, de tal forma que o reserva possa assumir as funções do primário, em caso de falha. Entretanto os mecanismos de tomada de pontos de recuperação e o processo de recuperação são diferentes em cada caso.

Targon/32 propõe compatibilidade com UNIX. GUARDIAN90 e Mach são sistemas operacionais proprietários, não compatíveis com UNIX.

5.1 Targon/32

O sistema Targon/32 [BOR89] implementa tolerância a falhas através da técnica de processos reserva. É uma versão tolerante a falhas do UNIX desenvolvida pela Nixdorf para sistemas distribuídos. Sua arquitetura compreende uma rede local com duas a dezesseis máquinas conectadas por barramentos duplos. Cada máquina é formada por três processadores com memória compartilhada. Cada processador executa o *kernel* do sistema operacional. O *kernel* é responsável pela criação e escalonamento dos processos e pelo gerenciamento da comunicação entre processos. Um dos três processadores de cada máquina executa as tarefas de manipulação de mensagens, criação e recuperação de processos reserva. Os outros dois processadores são disponíveis para executar processos UNIX.

Cada sistema possui um único servidor de processos (que mantém a configuração do sistema) e um número configurável de servidores de páginas. O servidor de páginas mantém uma réplica do espaço de memória virtual de um subconjunto de processos primários do sistema. Mantém também os pontos de recuperação necessários para os processos reserva.

5.1.1 Funcionamento básico

A unidade de recuperação no Targon/32 é o processo. Cada processo residente em uma máquina pode possuir um reserva em outra máquina, diferente da primeira. Um processo reserva contém uma quantidade suficiente de informações para reiniciar a execução quando a máquina do processo principal falhar.

O servidor de processos é responsável por determinar em quais máquinas o processo primário e seu reserva residirão. É também responsável por decidir quando e onde um novo processo reserva será localizado após uma falha de colapso. O servidor de processos também possui reserva.

O estado do processo primário é periodicamente salvo no reserva com o auxílio de uma operação de sincronização denominada *sync*. Entretanto, mensagens recebidas pelo primário depois do último *sync* também devem ser tornadas disponíveis para o reserva. Assim, quando uma mensagem (figura 10) é enviada pelo processo A, três processos recebem essa mensagem: o reserva do processo transmissor (A'), o processo receptor (B) e o reserva do processo receptor (B'). Uma difusão (*broadcast*) atômica de três vias assegura que: ou os três destinatários recebem a mensagem (B, B', A'), ou nenhum a recebe.

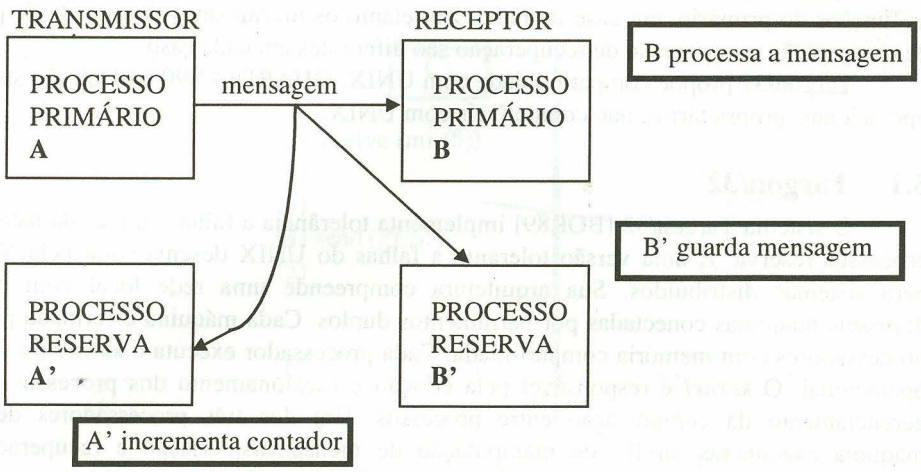

Figura 10: Comunicação entre processos no Targon/32

No processo reserva do receptor (B'), as mensagens recebidas são guardadas até a próxima operação de *sync*. Já no processo reserva do transmissor (A'), a cada mensagem recebida, é incrementado um contador (*write-since-sync*) para posterior controle de mensagens, quando o reserva for executado por falha do principal.

Sempre que a máquina que executa o processo primário falha, seu reserva é ativado. O reserva inicia a operação a partir da última operação de *sync*. O reserva lê as mesmas mensagens (que estão guardadas) que foram recebidas pelo principal. O reserva evita a retransmissão das mensagens já enviadas pelo principal usando o contador (*write-since-sync*). Quando o reserva alcança o mesmo estado em que estava o principal, as mensagens recebidas guardadas são descartadas e o contador de mensagens enviadas é zerado.

5.1.2 Pontos de recuperação no Targon

O estabelecimento de um ponto de recuperação no Targon/32 sincroniza um processo reserva com seu primário. O estado de um reserva é tornado idêntico ao seu processo primário através de uma operação de sincronização (*sync*). A operação *sync* é automaticamente iniciada pelo *kernel* sempre que o número de mensagens recebidas pelo primário excede um dado valor ou o processo primário está executando há um tempo muito longo desde a última operação de *sync*. Normalmente, o momento em que ocorre a sincronização é imediatamente antes de retornar de uma chamada ao sistema (*system call*) ou de uma interrupção de gerenciamento de memória virtual (*page fault*). Pode ocorrer também no início de uma nova "fatia de tempo".

A operação *sync* é executada em duas fases pela máquina do processo principal. Na **primeira fase** de sincronização o mecanismo convencional de paginação da gerência de memória virtual é empregado para enviar todas as páginas de memória alteradas e não

salvas (*dirty*) para o servidor de páginas e também para o reserva do servidor de páginas. O mesmo mecanismo de paginação é usado também para salvar a pilha do processo primário, no caso desta ter sido alterada desde a última operação de sincronização. Nesse sistema, a pilha do processo é mantida nas páginas de memória do processo e não na área de memória endereçada pelo *kernel*. Após o recebimento das páginas do processo primário, o servidor de páginas incorpora-as ao conjunto de páginas do processo primário.

A **segunda fase** de sincronização constrói a mensagem de *sync*, que contém: (a) todas as informações de estado independentes de máquina do processo primário, como valor dos registradores e endereço virtual da próxima instrução; (b) informação sobre todos os canais de comunicação abertos e as mensagens lidas de cada canal desde a última operação de *sync*; (c) informação para construir a pilha do *kernel* para que, na recuperação, o processo apareça como se estivesse justamente entrando ou retornando de uma chamada do sistema.

Logo após a construção de uma mensagem *sync*, ela é enviada para o processo reserva do processo primário, para o servidor de páginas e para o reserva do servidor de páginas usando o mecanismo de difusão atômica de três vias do sistema.

O processo primário volta a execução de suas tarefas tão logo a mensagem de *sync* seja enviada. O servidor de páginas, quando recebe uma mensagem de *sync*, torna o espaço de endereçamento de páginas do reserva idêntico ao primário e libera as antigas páginas do reserva que não sejam mais necessárias. A máquina que está executando o processo reserva, quando recebe uma mensagem de *sync*, atualiza o estado do reserva e descarta todas as mensagens antigas recebidas pelo reserva desde a última operação de *sync* (a variável *write-since-sync* é zerada).

5.1.3 Detecção de falhas

Falhas são detectadas usando o mecanismo de anel virtual e mensagens de *I'am alive*. Uma máquina é considerada falha quando pára de se comunicar com as demais. Falhas podem ser detectadas por hardware ou software. Após a identificação da falha, a máquina é desativada. Uma máquina pode também ser desativada por outras máquinas.

As máquinas estão organizadas em um anel virtual. Cada máquina periodicamente se comunica com a máquina da direita reportando atividade (*I'am alive*) e cada máquina espera o recebimento de comunicação do seu vizinho da esquerda. Através do anel virtual, uma máquina desativada é facilmente reconhecida pelos seus vizinhos e seu estado é notificado para o resto do sistema.

Se uma máquina A pára de se comunicar, seu vizinho da direita, B, tenta se comunicar com ela. Se B não tem sucesso, primeiro B tenta se comunicar com outras máquinas no sistema para estabelecer se a falha é sua (B) ou do seu vizinho (A). Se B ainda não tem sucesso, assume que está falho e pára. Se B tem sucesso na identificação de A como uma máquina desativada, B envia uma mensagem para A ordenando que pare (caso A não tenha percebido ainda que falhou), difunde uma mensagem chamada *machine dead*

para todos os nodos do sistema indicando a desativação de A e localiza um novo vizinho para a sua esquerda.

5.1.4 Recuperação no Targon

O algoritmo para recuperação do processo reserva é simples. Quando uma máquina é desativada, uma mensagem de *machine dead* chega na fila de mensagens de todos processos reserva que deverão ser ativados.

Para cada processo da máquina desativada, o *kernel* toma as seguintes medidas: (a) aloca e inicializa estruturas de dados necessárias para o estado do *kernel* local e mapeamento de memória; (b) solicita a lista de páginas mantidas pelo servidor de páginas de modo que o mapeamento de memória possa ser corretamente inicializado; (c) atualiza a pilha do *kernel* a partir da última informação de sincronização; (d) coloca o processo na fila de escalonamento.

O período da execução no qual um processo reexecuta o código que já foi executado pelo principal é chamado ***roll forward.*** Quando um processo tenta enviar uma mensagem, ele decrementa seu contador *write-since-sync* e a mensagem é descartada.

A partir do momento que uma máquina anteriormente desativada por falha retorna ao sistema, os antigos processos reserva dessa máquina, que até o momento estão executando sem reserva em outras máquinas do sistema, devem criar reservas na máquina reintegrada. A máquina reintegrada notifica o servidor de processos que está novamente ativa. O servidor de processos notifica então todas as demais máquinas do sistema.

5.2 Mach

Mach [BAB90] visa: fornecer uma base para a construção de sistemas operacionais, suportar espaços de endereçamento grandes e esparsos, permitir acesso transparente aos recursos da rede, explorar paralelismo e possibilitar portabilidade para uma grande diversidade de máquinas.

O *kernel* do Mach fornece os mecanismos essenciais para o funcionamento do sistema, como gerenciamento de processos, gerenciamento de memória, comunicação e serviços de I/O, mas deixa as demais operações para o nível de processos de usuário. Arquivos, diretórios, e outras funções tradicionais de sistemas operacionais são manuseadas no espaço do usuário. O *kernel* gerencia cinco principais abstrações: *threads*, tarefas, objetos de memória, portas e mensagens. Para recuperação, tarefas, *threads* e mensagens são os elementos de interesse.

O sistema consiste de múltiplos nodos autônomos, que podem estar fortemente acoplados através de um barramento paralelo ou fracamente acoplados através de uma rede. Processadores são *fail-stop*, isto é, diante da ocorrência de falhas, páram de funcionar. Falhas de um nodo não afetam a operação nos outros nodos. Não há componentes críticos que possam deixar grande parte do sistema inativo. Entretanto, o subsistema de comunicação pode perder mensagens.

O Mach é orientado à comunicação e é apropriado para arquiteturas multiprocessadas e distribuídas. Aplicações são denominadas tarefas no Mach. Uma tarefa (*task*) no Mach, consiste de um espaço de endereçamento e uma coleção de *threads* que executam naquele espaço de endereçamento. Uma *thread* é a menor unidade independente de execução escalonamento no Mach. Uma tarefa é a unidade básica de alocação de recursos. Tarefas são passivas. A execução está associada com as *threads*.

A comunicação entre tarefas é baseada na troca de mensagens. Para receber mensagens, a tarefa solicita para o *kernel* criar uma espécie de *mailbox* protegido, chamado porta. Uma porta é alocada dentro do *kernel* e tem a capacidade de ordenar uma fila de mensagens.

5.2.1 Recuperação de tarefas no Mach

A recuperação é semelhante à aplicada no sistema operacional Targon/32 [BOR89]. A unidade de replicação é a tarefa (*task*).

Cada nodo do sistema executa um *kernel* do Mach independente, que gerencia os recursos locais daquele nodo. As unidades de replicação são as tarefas. *Threads* estão implicitamente replicadas dentro de uma tarefa reserva. Tarefas reservas são replicadas em nodos diferentes. A tarefa reserva mantém-se inativa, mas o estado da tarefa principal é guardado através de um ponto de recuperação. Quando a tarefa principal falha, a tarefa reserva torna-se ativa e executa ***roll forward***, processando as mensagens que estão enfileiradas, e continuando o processamento da tarefa principal.

Um mecanismo de difusão de mensagens por três vias, semelhante ao Targon/32 [BOR89], é suprido pelo *kernel* para manter a tarefa reserva atualizada com a principal.

Targon/32 conta com suporte de hardware para a difusão de mensagens. No Mach, o *Trans Protocol* é uma solução de software para este tipo de comunicação. No *Trans Protocol*, o reconhecimento (*acknowledgement*) de uma mensagem difundida é enviada por um único nodo na próxima difusão de mensagens, evitando o envio de reconhecimentos separados e diminuindo o tráfego de mensagens na rede.

5.2.2 Pontos de recuperação e procedimentos pós-falhas no Mach

Periodicamente, o estado da tarefa reserva é igualado com o estado da tarefa principal. Esta tomada de ponto de recuperação é iniciada automaticamente pelo *kernel* local da tarefa principal. O *kernel* do nodo, onde está localizada a tarefa principal, precisa guardar as páginas que foram modificadas desde a última operação de tomada de ponto de recuperação. Quando solicitado para fazer o PR, o *kernel* da tarefa principal envia as páginas modificadas para o servidor de páginas do reserva

O *kernel* da tarefa principal é responsável pela atualização correta da tarefa reserva. A tarefa principal envia uma mensagem para o *kernel* da tarefa reserva contendo o estado de todas as *threads* contidas dentro da tarefa e o número de mensagens lidas pelas *threads* na tarefa principal. O *kernel* da tarefa reserva instala o estado criando portas que não

existem no reserva, retirando aquelas que foram liberadas e inicializando o contador de mensagens lidas para zero.

Uma falha pode afetar uma única tarefa ou muitas tarefas em um nodo. No primeiro caso, o *kernel* do Mach no nodo onde ocorreu a falha do principal envia uma solicitação para o nodo reserva para ativar a tarefa reserva. Quando o *kernel* reserva recebe esta solicitação, ele examina um parâmetro que identifica se o reserva será criado em um outro nodo da rede ou no nodo em que houve a falha. Em qualquer caso, o *kernel* completa a operação de recuperação colocando as *threads* de uma tarefa em execução.

No caso da falha atingir o nodo inteiro, envolvendo múltiplas tarefas, o nodo inteiro é identificado como inativo e cada *kernel* de cada um dos nodos do sistema se examina para determinar se contém alguma tarefa que seja reserva das tarefas principais do nodo que falhou. Para cada tarefa, o procedimento de recuperação é executado.

5.3 Tandem

Tandem Computers Inc. vem desenvolvendo computadores desde 1970, sendo uma pioneira e líder no segmento de computadores de grande porte tolerantes a falhas. Entre os mais recentes lançamentos da empresa podem ser citados os sistemas NonStop Cyclone, que executam o sistema operacional proprietário GUARDIAN90.

O sistema Tandem GUARDIAN90 é um multiprocessador fracamente acoplado, com interação entre processos suportada por troca de mensagens, especialmente construído para o processamento de transações *on-line*. A base do sistema de programação é formada por pares de processos: um processo primário e o seu reserva. Uma função crítica é replicada em dois processadores, o processo primário executa o serviço necessário, o processo reserva só entra em operação em caso de falha no primário.

GUARDIAN90 complementa a redundância em hardware típica dos computadores NonStop provendo detecção de erros e recuperação a partir de erros de software e hardware usando testes de consistência, protocolos especiais e processos reservas.

O sistema operacional GUARDIAN90 é formado por um *kernel* e um grande número de processos, em especial processos de supervisão para cada um dos processadores. Tanto para processos do sistema como para processos do usuário, GUARDIAN90 permite a criação de pares. Um par é formado por um processo primário ativo e um processo reserva passivo, cada processo residindo em um processador diferente. O processo primário envia pontos de recuperação ao processo substituto.

5.3.1 Detecção de erros no sistema GUARDIAN90

O sistema operacional foi projetado para parar imediatamente o processador suspeito de se encontrar em um estado errôneo, de forma que os demais processadores possam perceber sua inatividade. Detecção de erros se processa da seguinte forma: erros em um processador são detectados pelo sistema operacional do processador através de testes de consistência. A cada segundo, o processo supervisor de cada processador envia sinal de vida, através de um protocolo do tipo *I'm alive,* a todos os outros módulos no

sistema. A cada dois segundos, o processo supervisor verifica se recebeu sinal de vida de cada um dos outros módulos. Se faltar um sinal, o processo entende que o módulo correspondente falhou.

Além desse controle mútuo, para cada operação de entrada e saída é realizado controle de cumprimento de prazo (*time-out*). Em caso de falha, o processo de entrada e saída substituto entra em operação.

5.3.2 Recuperação no GUARDIAN90

Um vez diagnosticada a falha em um processador, todos os processos reservas relacionados aos processos primários que estavam sendo executados no processador com falha, são rolados para o último ponto de recuperação e ativados, tornando-se então processos primários. O sistema é reconfigurado em função dos novos processos primários. Tão logo o processador faltoso seja reparado, os novos processos primários criam seus processos reserva nesse processador. Em caso de falha de um canal de entrada e saída, o processo substituto correspondente é rolado e ativado, enquanto o processo primário é desativado passando a ser substituto do primeiro.

Consistência entre processos primários e seus reservas é mantida através de mensagens periódicas de ponto de recuperação. Antes da execução de cada função crítica, o processo primário envia um ponto de recuperação ao seu reserva. A mensagem de ponto de recuperação contém os dados e a informação de estado necessárias para completar a operação crítica se o processo primário falhar.

6 Conclusões

As implicações e soluções para a recuperação de sistemas sujeitos a falhas são extensas e variadas, exigindo um estudo muito extenso para seu aprofundamento. Assim, decidiu-se optar pelo estudo das questões que são resolvidas no âmbito dos sistemas operacionais. Estas soluções são preferidas devido às suas qualidades de transparência, eficiência e confiabilidade. Mas a resolução dos aspectos de detecção e recuperação apenas no sistema operacional, e eventualmente no âmbito de banco de dados, é questionada por alguns autores [SAL84, HUA93]. Estes autores demonstram que a tolerância a falhas não pode ser completa sem o conhecimento e o auxílio do software de aplicação, argumentando que os métodos para a detecção e recuperação de falhas implementados em níveis inferiores são ineficientes nos níveis superiores. Eles justificam seus argumentos através dos seguintes exemplos:

- a replicação de dados através de "espelhamento" em disco, usando uma facilidade disponível no sistema operacional, será mais ineficiente do que replicar apenas os arquivos da aplicação identificados como "críticos", por ação da camada de aplicação, já que o sistema operacional não dispõe de conhecimento suficiente para identificá-los;
- esquemas generalizados de tomada de pontos de recuperação embutidos no sistema operacional armazenam integralmente os dados da aplicação existentes na memória,

enquanto que métodos controlados pela aplicação irão armazenas apenas os dados críticos.

Estes aspectos podem ser ponderados se o objetivo for definir o projeto de sistemas de uso restrito, onde sejam conhecidas, de antemão, as características da aplicação. Em sistemas de aplicações gerais, deixar a implementação destas soluções para o programador de aplicações é destiná-las ao esquecimento.

Um outro aspecto a considerar é a possível integração de mecanismos de recuperação. Neste texto, foram apresentadas as visões conceituais puras dos mecanismos de recuperação por retorno. Entretanto, hoje é defendida a integração, usando os benefícios do sistema de avanço nos casos possíveis de prever antecipadamente as falhas e retorno nos demais. Não se pode esquecer, entretanto, que esta técnicas de integração devem lidar permanentemente, em funcionamento normal, com toda sobrecarga devida ao armazenamento dos pontos de recuperação.

Referências

[AND81] ANDERSON, T.; LEE; P. Fault Tolerance - Principles and Practice. Prentice-Hall, 1981.

[BAB90] BABAOGLU, O. Fault Tolerant Computing Based on Mach. *ACM Operating Systems Review*, New York, v.24(1):27-39, Jan. 1990.

[BIR96] BIRMAN, K. *Building Secure and Reliable Network Applications*. Manning, 1996.

[BOR89] BORG, A.; BLAU, W.; GRAETSCH, W.; HERRMANN F.; OBERLE, W. Fault Tolerance under UNIX. *ACM Trans on Computer Systems*. v.7(1):1-24; Feb. 1989.

[BOW92] BOWEN, N.; PRADHAN, D. Virtual checkpoints: architecture and performance. *IEEE Trans. on Computers*, vol.41(5):516:25, May 1992.

[CAL97] CALLEGARO, A. P. Estudo de algoritmos para recuperação assíncrona de processos em sistemas distribuídos. CPGCC da UFRGS. Jan. 1997 (TI n.602)

[CHA85] CHANDY, K.; LAMPORT, L. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*. v.3(1):63-75; Feb. 1985.

[COU94] COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Distributed Systems - concepts and design*. Addison Wesley, 1994.

[CRI91] CRISTIAN, F. Understanding Fault-Tolerant Distributed Systems. *Comm. of the ACM*. v. 34(2):57-78; Feb. 1991.

[HOA78] HOARE, C. Communicating Sequential Processes. *Communications of ACM*. vol. 21, pp.667-77.

[HUA93] HUANG, Y.; KINTALA, C. Software Implemented Fault Tolerance: Technologies and Experience. In: *IEEE Intl. Symposium on Fault-Tolerant Computing, 23*. Jun. 1993. p. 2-9.

[HUN87] HUNT, D.; MARINOS, P. A general purpose cache-aided rollback error recovery. In: *IEEE Intl. Symp. on Fault-Tolerant Comp.* Jun. 1987. p. 170-5.

[JAL94] JALOTE, P. *Fault Tolerance in Distributed Systems.* New Jersey: Prentice-Hall, 1994.

[KOO87] KOO, R.; TOUEG, S. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Trans.on Software Engineering*, v.SE-13(1):23-31, Jan. 1987.

[LAM78] LAMPORT, L. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM.* v.21(7): 95-114. July 1978.

[LAP85] LAPRIE, J-C. Dependable Computing and Fault-Tolerance: Concepts and Terminology. In: *IEEE Intl. Symp. on Fault-Tolerant Computing, 15.* Jun. 1985. p. 2-11.

[NEL81] NELSON, B. *Remote Procedure Call.* PhD. Dissertation. CMU-CS-81-119. Carnegie Mellon University, Pittsburgh, PA 1981.

[NEL90] NELSON, V. Fault-tolerant computing - fundamental concepts. *IEEE Computer.* v.23 (7): 19-25, July 1990.

[PRA96] PRADHAN, D. *Fault-Tolerant System Design.* Prentice Hall, New Jersey, 1996.

[SAL84] SALTZER, D.; REED, D.; CLARK, D. End-to-End Arguments in System Design. *ACM Trans. on Computer Systems.* v.2(4):277-288; Nov. 1984.

[SIN94] SINGHAL, M.; SHIVARATRI, N. *Advanced Concepts in Operating Systems.* McGraw-Hill, 1994.

[STR85] STROM, R.; YEMINI, S. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, v.3(3):204-226, Aug. 1985.

[STR88] STROM, R.; BACON, D.; YEMINI, S. Volatile Logging in n-Fault-Tolerant Distributed Systems. In: *IEEE International Symposium on Fault-Tolerant Computing, 18.* Jun. 1988. p. 44-49.

[WU90] WU, K-L.; FUCKS, W.; PATEL, J. Error recovery in shared memory multi-processors using private caches. *IEEE Trans. on Parallel and Distributed Systems.* v.1(2): 231-240. Apr. 1990.