# Arquitetura de um Núcleo para Programação de Aplicações Distribuídas Tolerantes a Falhas

Marcelo Migueletto de Andrade

Osvaldo S. F. Carvalho

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Caixa Postal 702, CEP 30161
Belo Horizonte - MG
Tel.: (031) 443-4088 - Fax: (031) 443-4352
Email: miguel@dcc.ufmg.br - vado@dcc.ufmg.br *

## Resumo

Em uma rede de computadores, o ganho de desempenho obtido na execução de aplicações em paralelo pode ser comprometido por uma diminuição da sua confiabilidade diretamente proporcional ao número de estações usadas. Este trabalho apresenta a arquitetura de um núcleo de programação distribuída cuja principal característica é a tolerância a falhas. São empregadas duas técnicias retroativas de recuperação de erros em conjunto, uma otimista e outra pessimista, baseadas, respectivamente, na gravação de pontos de recuperação consistentes e em réplicas de processos.

## Abstract

In a computer network, the performance gain in the parallel execution of applications may be compromised by a reduction in reliability which is directly proportional to the number of used stations. This work presents the architecture of a distributed programming kernel whose main feature is fault tolerance. Two retroactive error recovery techniques are employed, an optimistic and a pessimistic one, based respectively on consistent checkpoint saving and on process replication.

*Este trabalho foi financiado com recursos da FAPEMIG (proc. TEC 1113/90) e do CNPq (proc. 502353/91-0(NV)

# 1 Introdução

Uma rede local de estações de trabalho pode constituir um poderoso recurso computacional para a execução de aplicações paralelas com a utilização de núcleos de programação distribuída [17] [5], que permitem o seu uso como um multiprocessador. Ao se dividir uma tarefa em sub-tarefas menores e distribuí-las em uma rede, porém, corre-se um risco maior de não conseguir terminá-la com sucesso. A confiabilidade e a disponibilidade [16] de um sistema qualquer são funções inversas do seu número de componentes. Quanto mais estações forem usadas e mais sub-tarefas existirem, maior a probabilidade de alguma delas falhar. Assim, é importante que os núcleos de programação distribuída provejam mecanismos que garantam a execução das aplicações na presença de falhas, mesmo que com uma degradação de desempenho.

Este trabalho apresenta a arquitetura de um núcleo de programação distribuída de uso geral para redes de estações de trabalho cuja principal característica é a tolerância a falhas de processos e de máquinas. A descrição da arquitetura é feita de maneira independente de sistema operacional, embora o núcleo esteja sendo projetado visando uma implementação em Unix.

A maioria dos sistemas operacionais distribuídos [22] [8] [4] [18] provê mecanismos transparentes de tolerância a falhas, não exigindo nenhum esforço adicional na programação das aplicações. O núcleo aqui proposto não apresenta esta característica, pois roda sobre o sistema operacional original das estações. Mesmo assim, tenta ser o mais transparente possível, escondendo do programador a maioria dos detalhes do problema de se garantir a completude da execução de computações distribuídas, mesmo que falhas venham a ocorrer.

Algumas premissas com relação ao modelo de aplicações e tipos de falhas toleradas foram feitas. Para garantir uma maior generalidade de uso, assume-se que as aplicações que vão fazer uso do núcleo não são de tempo real e são compostas por processos não determinísticos [7]. Em um sistema não determinístico, pode haver mais de uma transição possível a partir de um estado e assim é impossível prever com exatidão qual o próximo estado atingido. Os processos se comunicam unicamente através da troca de mensagens e podem ser executados em qualquer estação. Nenhuma falha provoca o particionamento da rede, de modo que uma estação só não consegue se comunicar com uma outra se esta tiver falhado.

Serão consideradas apenas falhas *fail-stop* [24] e transientes [16]. Falhas *fail-stop* provocam sempre o fim do funcionamento dos componentes de hardware e da execução dos processos em que ocorrem, em oposição a falhas bizantinas [15] [24], nas quais o afetado continua trabalhando, só que de maneira incorreta. Falhas transientes são aquelas decorrentes de condições temporárias como, por exemplo, defeitos de hardware ou terminações anormais de processos. Não são abordadas, neste estudo, falhas de software [2] [13] [1], que são, geralmente, resultantes de erros de projeto e, portanto, permanentes.

A próxima seção discute a escolha da técnica de recuperação de erros em sistemas distribuídos a ser usada no núcleo. A técnica de recuperação de erros e a arquitetura de um sistema distribuído estão intimamente relacionadas. Nas seções 3 e 4, respectivamente, são mostrados o posicionamento do núcleo perante as aplicações e como se dá a interação entre eles. A arquitetura do sistema é detalhada na seção 5, após a qual vêem a conclusão e as perspectivas futuras do trabalho.

# 2 Seleção de uma Técnica de Recuperação de Erros

Sistemas distribuídos tolerantes a falhas são capazes de, quando da ocorrência de um erro, recuperarem-se, posicionando-se novamente em um estado consistente [7], a partir do qual possam prosseguir sua execução normalmente. De acordo com [20], um conjunto de estados de processos pode ser dito consistente se cada par de processos concorda sobre a comunicação ocorrida e não ocorrida entre eles. A ausência de informações sobre o estado global do sistema é a principal dificuldade na tarefa de recuperação.

Pode-se classificar as técnicas de recuperação de erro em sistemas distribuídos em três categorias básicas: técnicas específicas, transações e técnicas retroativas.

## 2.1 Técnicas Específicas

São consideradas específicas as técnicas desenvolvidas sob medida e programadas diretamente no código dos sistemas tolerantes a falhas. Geralmente, têm desempenho superior aos apresentados por técnicas mais gerais, pois podem fazer uso de um profundo conhecimento da aplicação e do hardware usado. Alterações nestes, porém, mesmo que pequenas, podem acarretar grandes mudanças no esquema de tolerância a falhas. Esta é a maior desvantagem deste tipo de técnica.

Enquadra-se nesta categoria a recuperação preditiva de erros [6], na qual, já durante a fase de projeto do sistema, são previstos os erros que poderão ocorrer e são definidos manipuladores de exceção para tratá-los.

## 2.2 Transações

Neste modelo, a computação é dividida em unidades de trabalho chamadas transações [10] [11]. Embora as transações possam ser executadas concorrentemente, um sistema deste tipo deve se comportar como se elas fossem executadas uma após a outra, numa seqüência qualquer. Esta propriedade das transações é chamada de *serializabilidade*. Outra propriedade que possuem é a *atomicidade*: a execução de uma transação deve se dar por completo ou então o sistema deve permanecer como se ela não tivesse sido executada. Assim, falhas nunca deixam estados intermediários de uma transação visíveis a outras transações e suas fronteiras sempre definem estados consistentes do sistema a partir dos quais computações podem se recuperar.

Este modelo é bastante apropriado a gerenciadores de bancos de dados [3], mas nem todas as aplicações podem ser expressas através de transações. Segundo [20], as transações são eficientes para se implementar processos únicos e logicamente seriais, através da execução paralela de suas partes independentes, mas não são uma técnica geral de tolerância a falhas.

## 2.3 Técnicas Retroativas

As chamadas técnicas retroativas realizam a recuperação de erros através da mudança do estado do sistema para estados anteriores à ocorrência dos mesmos. Há dois tipos de técnicas retroativas: as otimistas e as pessimistas.

### 2.3.1 Técnicas Otimistas

Técnicas otimistas recebem este nome porque baseiam-se na suposição de que falhas são pouco comuns. Assim, optam por minimizar o atraso provocado pelo mecanismo de tolerância a falhas na execução de computações livres (ou quase livres) de erro, sujeitando-se a, em caso de erro, gastar um tempo maior na operação de recuperação.

Durante a execução de uma computação distribuída, de tempos em tempos os estados dos processos que a compõem são salvos em armazenamento estável (isto é, que não se perde com falhas de processos ou processadores), constituindo os chamados *pontos de recuperação*. Se um ou mais processos falham, eles, e possivelmente outros processos com os quais tenham se comunicado, se recuperam retornando aos estados gravados nos seus pontos de recuperação mais recentes para que a computação possa prosseguir. Quanto mais freqüente for a gravação de pontos de recuperação, menores serão os tempos de recuperação, mas maiores os atrasos impostos às execuções em que não ocorram erros.

Problemas podem ocorrer durante a recuperação de um erro se os processos gravarem e retornarem a seus estados de maneira completamente independente uns dos outros. Por exemplo, sejam PRA e PRB os pontos de recuperação de dois processos A e B, respectivamente. Se PRB registrar o recebimento de uma mensagem M de A e PRA não registrar o seu envio e ambos os processos retornarem a esses pontos, então o estado da computação estará inconsistente (deve-se notar que a situação inversa, ou seja, PRA registrar o envio de M e PRB não registrar o seu recebimento, não é inconsistente, pois equivale à perda de M pelo canal de comunicação). Inconsistências como esta podem provocar o que se chama de *efeito dominó* [14] [20], no qual a busca de um estado

consistente para a computação provoca uma cascata de retornos de dois ou mais processos a pontos de recuperação anteriores. *Livelocks* [14], que são seqüências infinitas de retornos a pontos de recuperação, também podem acontecer.

Em [20] Strom e Yemini propõem um esquema de recuperação otimista no qual a gravação de pontos de recuperação pode ser feita assincronamente em relação à computação graças ao acompanhamento das dependências de causalidade entre as mensagens trocadas pelos processos. Esse acompanhamento permite que, quando da ocorrência de uma falha, um estado consistente para a computação seja reconstruído a partir dos pontos de recuperação gravados. Assim, além de pontos de recuperação, são gravados também os históricos das mensagens recebidas por cada processo. Assume-se que, a partir do estado inicial de um processo e da seqüência de mensagens que ele recebe, pode-se determinar o seu estado final e a seqüência de mensagens que enviará. Em outras palavras, pressupõe-se que os processos são determinísticos.

Uma técnica que funciona com computações não determinísticas é apresentada por Koo e Toueg em [14]. A operação de gravação de pontos de recuperação exige, porém, uma coordenação entre os processos, de modo que o conjunto de estados gravados seja sempre consistente. Os algoritmos de gravação de pontos de recuperação e de recuperação toleram falhas durante a sua execução e envolvem um número mínimo de processos. Mínimo também é o número máximo de pontos de recuperação que cada processo pode ter gravados em qualquer instante: apenas dois.

A principal desvantagem das técnicas otimistas em geral é a dificuldade de se prever o tempo gasto pela recuperação de erros, haja visto que, além dos processos que falharem, um número arbitrário de processos pode ter que retornar a seus pontos de recuperação para evitar que o estado da computação fique inconsistente. Em [9] é descrito um protocolo chamado *Manetho* para gravação e retorno a pontos de recuperação que consegue restringir a operação de retorno apenas aos processos que falharem. A gravação de pontos de recuperação não exige coordenação de processos, mas cada processo tem que registrar as mensagens que envia e manter um grafo com informações sobre ordenação de eventos. Além disso, o esquema de recuperação de erros é bastante complexo e o suporte a não determinismo é limitado.

### 2.3.2 Técnicas Pessimistas

Nas técnicas pessimistas, os processos gravam pontos de recuperação com muito mais freqüência do que nas técnicas otimistas, por exemplo a cada envio de mensagem. Desta forma, a recuperação de um erro envolve apenas o processo atingido e consiste somente no seu retorno ao estado guardado no seu ponto de recuperação mais recente. O tempo gasto na operação é, portanto, pequeno e previsível. Entretanto, quando as falhas forem infreqüentes ou não ocorrerem, atrasa-se desnecessariamente a computação com a gravação de informações para recuperação.

Para evitar que esses atrasos sejam inaceitáveis, as técnicas pessimistas, ao invés de gravarem os pontos de recuperação em armazenamento estável, valem-se da replicação de processos em processadores diferentes para guardá-los. Os processos são replicados formando pares ou grupos.

Um par de processos é composto por um processo primário e um reserva. O estado do processo reserva é mantido atualizado pelo primário de modo que, em caso de falha, ele possa começar a executar ativamente no seu lugar. Sempre que um dos processos do par falhar, um novo processo de reserva é criado. Não são toleradas falhas simultâneas (ou praticamente simultâneas) nos dois processos do par.

Os grupos de processos podem funcionar como uma generalização dos pares de processos, em que um deles é o primário e os demais são secundários, ou em um esquema onde as réplicas têm igual status. Quanto maior o número de réplicas, maior a tolerância a falhas e menor a performance do sistema. Em [12] é apresentado um protocolo de comunicação confiável para grupos de processos de mesmo status e é discutida a sua aplicação em tolerância a falhas.

## 2.4 A Técnica escolhida para o Núcleo

Não se pode dizer que uma técnica é definitivamente superior às demais sem levar-se em conta a situação em questão. Dadas as necessidades e premissas levantadas, a escolha da técnica recai sobre

os algoritmos de Koo e Toueg para gravação e recuperação de pontos de recuperação consistentes. Uma técnica específica não pode ser usada, pois o núcleo deve ser de uso geral, e modelo de transações é mais voltado para gerenciadores de bancos de dados Assim, uma técnica retroativa seria mais razoável. Optou-se por um modelo otimista por se acreditar que as falhas não são muito freqüentes no tipo de ambiente em questão e que as aplicações podem tolerar demoras causadas por eventuais recuperações de erros. Destas técnicas, a escolhida é a mais geral, atendendo à restrição imposta inicialmente de que os processos poderiam ser não determinísticos.

### 2.5 Descrição dos Algoritmos de Koo e Toueg

O algoritmo de gravação de pontos de recuperação é baseado nos protocolos de *commit* de duas fases. Na primeira fase, o processo iniciador da operação grava um ponto de recuperação tentativo e solicita aos demais processos que façam o mesmo. Cada processo, ao receber a solicitação, ou grava o ponto tentativo e envia uma resposta positiva ao iniciador ou não o grava e envia uma resposta negativa. Caso todas as respostas tenham sido positivas, o iniciador decide transformar em permanentes os pontos tentativos; caso contrário, decide descartá-los. Na segunda fase, essa decisão é comunicada e executada pelos demais processos. Uma vez que todos ou nenhum dos processos gravam os seus estados em caráter definitivo, o mais recente conjunto de pontos permanentes é sempre consistente. Para cada processo, há gravados, portanto, no máximo dois pontos de recuperação em qualquer instante. Este número é demonstrado ser mínimo.

Modificações no algoritmo acima permitem que apenas um número mínimo de processos tenha que gravar pontos de recuperação juntamente com o processo iniciador. A idéia é que cada processo só tenha que gravar um ponto tentativo caso o ponto tentativo do processo que lhe enviou a solicitação registre o recebimento de uma mensagem enviada por ele e o envio dessa mensagem não esteja registrado no seu último ponto permanente.

Para que os processos sejam capazes de tomar a decisão sobre ter que gravar ou não um ponto de recuperação tentativo, um esquema de numeração de mensagens, no qual todas as mensagens de aplicação enviadas por um processo são rotuladas com um número monotonicamente crescente, é adotado. Considera-se, no escopo deste trabalho, mensagens de aplicação como sendo quaisquer mensagens que não são enviadas pelos algoritmos de tolerância a falhas; estas são chamadas de mensagens de sistema.

O algoritmo de retorno a ponto de recuperação também é baseado nos protocolos de *commit* de duas fases. Na primeira fase, o iniciador pede a todos os processos que retornem aos seus pontos de recuperação permanentes. Cada processo que receber uma mensagem nesse sentido responde positiva ou negativamente, dependendo da sua possibilidade de retornar sua execução ao seu ponto permanente. O iniciador só decide pelo retorno caso todos os processos consultados o aprovem. Na segunda fase, a decisão é comunicada e executada pelos processos consultados.

Apenas um número mínimo de processos pode ser obrigado a retornar aos seus pontos permanentes se a seguinte modificação for feita nesse algoritmo: um processo que receber uma solicitação de retorno só terá que atendê-la se o estado para o qual o processo que a enviou irá retornar é anterior ao envio de alguma mensagem deste para ele (caso não retornasse, o processo ficaria em um estado no qual estaria registrado o recebimento de uma mensagem que ainda não havia sido enviada). A decisão sobre a necessidade ou não do retorno também é baseada na numeração das mensagens.

Conforme foi colocado, ambos os algoritmos toleram falhas que por ventura venham a ocorrer durante a sua execução.

## 3 O Núcleo e as Aplicações

O núcleo proposto serve de suporte de tolerância a falhas para a execução de várias aplicações concorrentes e independentes. Ele é composto por um gerenciador e por uma biblioteca de funções que implementam os algoritmos de gravação e retorno a pontos de recuperação e que constituem a interface dos processos de aplicação com o gerenciador.

Todo processo de aplicação é conectado através de uma ligação ponto a ponto ao gerenciador para com ele poder se comunicar. A iniciativa dessa comunicação é sempre dos processos, que através de um protocolo de mensagens solicitam informações e serviços. Quando uma mensagem de sistema é enviada, porém, o processo ao qual ela é destinada é avisado assincronamente pelo gerenciador, para que ele então possa solicitar a sua recepção.

A figura 1 mostra duas aplicações distribuídas, A1 e A2, executando no ambiente de tolerância a falhas proporcionado pelo núcleo. Não se preocupou em mostrar a distribuição de processos por estações, mas apenas a posição do núcleo perante as aplicações. A parte destacada dos processos representa as funções da biblioteca mencionada e as estruturas de dados por elas usadas.

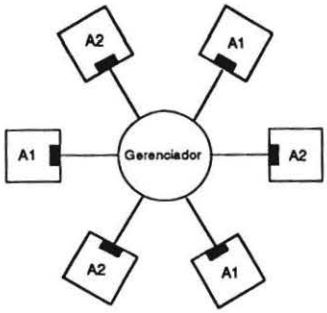


Figura 1: O núcleo e as aplicações

As funções desempenhadas pelo gerenciador do núcleo são: (a) detectar falhas de processos de aplicação; (b) recriar processos de aplicação que tenham falhado; (c) prover armazenamento estável para a gravação de pontos de recuperação e quaisquer outros arquivos usados pelos processos; (d) possibilitar a comunicação entre os processos, garantindo a entrega e a ordenação das mensagens, sejam elas de aplicação ou de sistema. O gerenciador também deve ser tolerante a falhas. A sua confiabilidade é a confiabilidade das aplicações do ambiente.

# 4 Funções da Interface Aplicação-Gerenciador

Serão agora apresentadas em linhas gerais as principais funções da biblioteca do núcleo que constituem a interface deste com as aplicações. Esta seção e a próxima descrevem juntas todo o mecanismo de tolerância a falhas.

## 4.1 Inicialização e Finalização

**join_apl (aplicação,**
**processo,**
**localização do código executável,**
**função de avaliação de estado,**
**função de carregamento de estado)**

Uma chamada a esta função deve ser feita no início de todos os programas de aplicação. Nos três primeiros parâmetros, são passados a identificação do processo, que é composta pelo nome da aplicação à qual pertence e pelo nome que o identificará dentro da mesma, e a localização do seu código executável no sistema de arquivos da estação em que está sendo executado. O quarto e o quinto parâmetros são nomes de funções escritas pelo programador

da aplicação que, respectivamente, sejam capazes de retornar o estado corrente do processo em uma seqüência de bytes e de restaurar o estado do processo para um outro estado que lhe for passado como argumento. Tipicamente, estes estados são constituídos por valores de variáveis do processo.

O primeiro passo dado por **join_apl** é instalar um tratador de eventos assíncronos para receber os avisos de mensagens de sistema do gerenciador. Em seguida, estabelece com o ele uma conexão ponto a ponto e o consulta para saber se o processo de aplicação está sendo iniciado pela primeira vez ou se já falhou antes e foi recriado.

Caso seja a primeira vez, é inicializada a numeração de mensagens de aplicação processo e enviada ao gerenciador, então, a linha de comando usada para disparar o processo. Se a linha de comando não puder ser obtida do sistema operacional, um parâmetro a mais deve ser incluído nesta função para que ela possa ser informada. Esta informação será usada pelo gerenciador caso a máquina em que o processo esteja sendo executado falhe e ele tenha que ser recriado em outro lugar. Feito isto, o gerenciador é avisado da necessidade de se salvar o código executável do processo em armazenamento estável e um ponto de recuperação inicial é gravado. Para tanto, são usadas as funções **stable_storage** e **save_cp**, detalhadas mais abaixo.

Se o processo tiver sido recriado, ele anula a informação no gerenciador que diz que ele estava morto e depois retorna ao seu último ponto de recuperação permanente. Faz isto chamando a função **rollback**, também descrita adiante, até que a operação seja bem-sucedida.

Após o retorno da chamada desta função, o processo estará sob a supervisão do núcleo, que cuidará para que ele seja recriado caso venha a falhar.

**stable_storage (arquivo, freqüência)**

Esta função informa ao gerenciador que um determinado arquivo deve ser gravado em armazenamento estável. Arquivos gravados em armazenamento estável vão estar disponíveis para o processo em qualquer estação em que ele vier a ser executado.

Duas podem ser as freqüências de salvamento do arquivo: sempre que se fizer a gravação de um ponto de recuperação permanente ou apenas uma vez. A primeira deve ser usada para arquivos que serão modificados durante a vida do processo e a segunda para os que não serão.

**leave_apl ()**

Desconecta o processo do gerenciador. Deve ser chamada no fim da execução do processo.

## 4.2 Comunicação Entre Processos

Toda a comunicação entre os processos das aplicações distribuídas deve se dar exclusivamente através de chamadas às funções **send** e **receive**, descritas abaixo. Elas lidam com a numeração das mensagens, mantendo atualizadas informações necessárias à execução dos algoritmos de gravação e retorno a pontos de recuperação.

**send (processo, mensagem)**

Possibilita o envio de uma mensagem de aplicação de um processo a qualquer outro processo de sua aplicação, independentemente da estação em que este esteja. Cada processo rotula suas mensagens com números monotonicamente crescentes e então as envia para o gerenciador, que garante que elas serão entregues, e na ordem em que foram enviadas.

**receive (processo, mensagem, bloquear)**

Esta função permite a um processo receber uma mensagem de outro processo da sua aplicação, cujo nome é indicado pelo primeiro parâmetro. A recepção de mensagens sem restrição de emissor pode ser feita fornecendo-se um nome "coringa" de processo pré-definido. O

argumento *bloquear* pode ser *sim* ou *não* e indica, respectivamente, se a chamada deve bloquear o processo até que a mensagem esperada chegue ou não. Caso o processo especificado tenha falhado e ainda não tenha se recuperado, ou se a chamada não for bloqueante e não haja mensagens a serem recebidas, códigos de erro apropriados são retornados. Como já foi mencionado, o gerenciador evita a entrega de mensagens repetidas aos processos.

## 4.3 Gravação e Retorno a Pontos de Recuperação

**save_cp ()**

Usada para a gravação de pontos de recuperação, esta função é chamada pelo iniciador da operação e usa a função de avaliação de estado do processo para obter um ponto de recuperação tentativo. Em seguida, prossegue na execução do algoritmo, comunicando-se através de mensagens de sistema com outros processos da aplicação, que, ao receberem a mensagem de solicitação de gravação de ponto de recuperação tentativo, desviam-se da sua execução normal para também seguirem o algoritmo, gerando seus pontos de recuperação da mesma maneira descrita acima.

A decisão final do processo iniciador pode ser descartar os pontos de recuperação tentativos ou transformá-los em permanentes. No primeiro caso, **save_cp** retorna um código de erro ao iniciador; no segundo, todos os processos participantes da operação enviam ao gerenciador os seus pontos de recuperação permanentes. Também nesta hora os arquivos especificados em chamadas a **stable_storage** são, de acordo com a freqüência estipulada, gravados em armazenamento estável.

**rollback ()**

Quando um processo, ao executar **join_apl**, descobre que está recomeçando sua execução após ter falhado, ele assume o papel de iniciador de uma operação de retorno a ponto de recuperação, chamando a função **rollback** para executar o algoritmo.

A comunicação entre os processos se dá por mensagens de sistema. O recebimento de uma mensagem de sistema solicitando o retorno ao último ponto de recuperação faz com que o processo inicie a execução do algoritmo. Caso haja consenso quanto ao retorno, cada processo envolvido na operação solicita ao gerenciador o seu ponto de recuperação e o número com que havia rotulado a última mensagem enviada antes de falhar. Os arquivos gravados em armazenamento estável automaticamente estarão disponíveis. O ponto de recuperação é passado à função de carregamento de estado, que deixará o processo em condições de prosseguir sua execução. Se não houver consenso, a função retorna um código de erro ao iniciador.

Um processo também pode chamar esta função sempre que quiser, por qualquer motivo, retornar ao seu último estado salvo com **save_cp**.

**willing_to_cp (sim ou não)**

Serve para que o processo possa expressar o seu desejo ou possibilidade de gravar um ponto de recuperação tentativo. Caso um *não* seja o argumento, o processo responderá negativamente a qualquer solicitação desse tipo.

**willing_to_rb (sim ou não)**

Análoga à função anterior, serve para que o processo possa expressar o seu desejo ou possibilidade de retornar a sua execução ao último ponto de recuperação permanente. Caso um *não* seja o argumento, o processo responderá negativamente a qualquer solicitação de retorno.

**hold_system_msgs (sim ou não)**

Como já foi dito, os processos são avisados assincronamente pelo gerenciador sobre a existência de mensagens de sistema a eles destinadas. Pode haver, entretanto, trechos do programa sendo executado nos quais interferências deste tipo sejam inconvenientes, como, por exemplo,

os que contenham operações de entrada e saída[1]. Usando **hold_system_msgs**, o programador pode proteger essas seções do código. Se um *sim* lhe for passado como parâmetro, esta função faz com que todos os avisos de mensagem de sistema fiquem pendentes até o instante em que o processo puder novamente recebê-los e a chame com um argumento *não*[2].

## 5 Arquitetura do Gerenciador

A arquitetura do gerenciador do núcleo, mostrada na figura 2 em uma rede com quatro estações, foi projetada de modo a deixá-lo tolerante a falhas. Baseia-se no princípio de grupo de processos de status diferentes, onde um deles é o primário e os demais réplicas secundárias de reserva. Desta maneira, no núcleo são empregadas duas técnicas diferentes de tolerância a falhas em conjunto, de modo a se tirar proveito das vantagens que cada uma oferece.

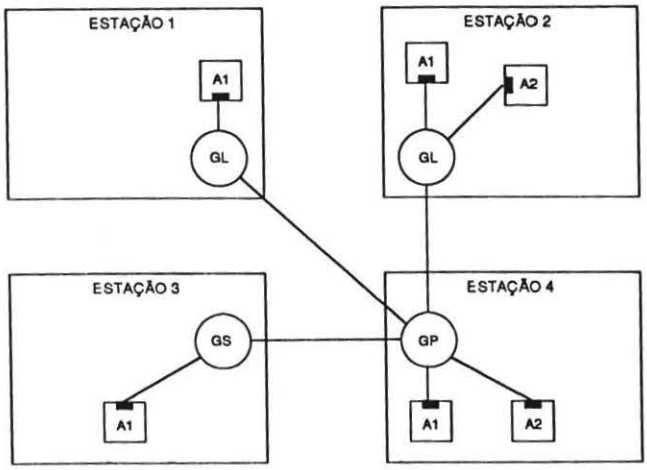


Figura 2: Arquitetura do gerenciador do núcleo

Cada estação da rede possui um processo gerenciador, que pode ser de três tipos: gerenciador global de aplicações primário (GP), gerenciador global de aplicações secundário (GS) e gerenciador local de aplicações (GL). É assumido que estes processos só falham se a estação em que estiverem sendo executadas pára de funcionar.

Os processos de aplicação conectam-se apenas ao processo gerenciador da sua máquina, através de uma ligação ponto a ponto em que a entrega e a ordenação das mensagens sejam garantidas. Caso o sistema operacional não ofereça este recurso, ele deve ser implementado utilizando-se, por exemplo, protocolos de janela deslizante como os descritos em [21]. O gerenciador é organizado numa topologia de estrela, onde o GP é o nó central. As ligações entre ele e os GS's e GL's também são ponto a ponto.

[1] No Unix, por exemplo, este tipo de problema pode ocorrer se o gerenciador usar o envio de sinais como forma de se comunicar assincronamente com os processos. O recebimento de um sinal por um processo provoca normalmente a interrupção de qualquer chamada de sistemas "lenta" (*read*, *write*, *wait*, etc.) que estiver sendo executada [19].

[2] As funções usadas para se bloquear sinais no Unix BSD e System V são, respectivamente, *sigblock* e *sighold* [19].

O GP centraliza todas as informações sobre os processos das aplicações sendo executadas. Além disso, serve de roteador para a entrega de mensagens (de aplicação ou de sistema), checa periodicamente se todas as estações estão funcionando, acompanha suas cargas e interage com os processos de aplicação da sua estação.

Pode acontecer, entretanto, que a estação em que o GP esteja rodando pare de funcionar. Para contornar este problema, todas as informações relativas às aplicações guardadas pelo nó central são mantidas replicadas nos GS's, de modo que em caso de falha do GP, qualquer GS possa assumir o seu lugar. O armazenamento estável é constituído pelo armazenamento secundário (para arquivos) e primário (para pontos de recuperação) das estações que executam gerenciadores globais.

O custo da manutenção da consistência das informações dos gerenciadores globais é diretamente proporcional ao número de GS's. Portanto, o ideal é que se tenha o mínimo deles, desde que a confiabilidade obtida seja aceitável. Isto pode ser feito alocando-se para eles as estações mais confiáveis. Nas demais, são executados os GL's, que cuidam apenas do gerenciamento dos processos de aplicação da sua estação, contribuindo somente para a performance do ambiente de execução distribuída mas não para a sua confiabilidade. Os GL's devem ser usados também nas estações sem disco, já que elas não podem executar o GP nem um GS porque não têm condições de participar da montagem do armazenamento estável e dependem funcionalmente das estações que armazenam seus sistemas de arquivos.

Para a instalação do núcleo, primeiramente o GP é colocado em execução. Os GS's e GL's são disparados depois e se conectam ao GP, informando-lhe as identificações das suas estações e se são gerenciadores locais ou globais. Todos os GS's são avisados pelo GP sobre a conexão de novas estações. A partir daí, os processos de aplicação podem ser executados, ficando sob a supervisão do núcleo após executarem **join_apl**, como visto na seção anterior.

Todos os dados passados por um processo de aplicação ao gerenciador da sua estação, quando da sua conexão a ele, são retransmitidos para o GP, pois ele centraliza informações sobre os processos de todas as aplicações. O GP adiciona a estes dados a identificação da estação em que o processo está e os passa aos GS's para que eles atualizem seus estados.

Visto em linhas gerais os papéis de cada tipo de processo na arquitetura do gerenciador, descrevemos agora como são implementadas cada uma das suas quatro funções - detectar falhas de processos de aplicação, recriar processos de aplicação que tenham falhado, prover armazenamento estável para a gravação de pontos de recuperação e outros arquivos quaisquer e possibilitar a comunicação confiável entre processos - e sua própria tolerância a falhas.

## 5.1 Detecção de Falhas de Processos de Aplicação

Em cada estação, o processo gerenciador, seja ele de qualquer tipo, de tempos em tempos verifica se todos os processos locais estão vivos. Se não for possível saber se um processo está vivo ou não pelo sistema operacional, esta checagem deve ser feita através do envio periódico de mensagens "estou vivo" dos processos de aplicação para o gerenciador, que associa o fim inesperado do recebimento dessas mensagens à falha do processo.

Feita a checagem, o processo gerenciador envia ao GP (a não ser que ele próprio seja o GP, é claro) uma mensagem dizendo que está vivo e os nomes dos processos locais que falharam desde a última checagem. O GP conta por quantos períodos cada estação está atrasada no envio dessas mensagens; se o atraso ultrapassar um certo limite, ele assume que a estação em que o GL ou GS estava sendo executado parou de funcionar e que, portanto, todos os processos de aplicação que lá estavam falharam. A informação sobre quais processos falharam e ainda não se recuperaram é mantida no GP e replicada nos GS's. Esses processos são marcados como não estando em nenhuma estação. Aos GL's só interessa saber quais GS's estão vivos, conforme será visto mais à frente.

Se uma estação que tiver falhado for reparada, ela pode ser reintegrada ao núcleo. O seu processo gerenciador se conecta ao GP da mesma maneira com que havia feito anteriormente. A reintegração de módulos reparados a um sistema é de grande relevância para a sua confiabilidade [11].

## 5.2 Criação de Processos de Aplicação

Se o GP descobre que um processo de aplicação morreu, ele escolhe uma nova estação para recebê-lo de acordo com a carga de cada uma delas. Quando os GL's e GS's enviam mensagens dizendo que estão vivos, informam também qual a sua carga de processamento corrente. Conhecendo a carga de todas as estações, o GP pode fazer a sua escolha de modo a tentar um melhor balanceamento de carga na rede [23].

Definida a estação onde um processo morto será recriado, o GP registra essa informação e envia ao gerenciador da sua estação a linha de comando para dispará-lo. Se o gerenciador for um GL, são enviados ainda o código executável do processo e os demais arquivos que tenha em armazenamento estável, seu ponto de recuperação permanente e o último número dado por ele a uma mensagem de aplicação. Este número é anotado automaticamente pelo GP a cada envio de mensagem, como será mostrado adiante. Os GS's não precisam receber código do processo porque ele já se encontra no seu armazenamento secundário, que faz parte do armazenamento estável. Os outros dados também não são necessários porque foram neles replicados pelo GP.

Durante toda a operação descrita acima, os GS's são atualizados a cada mudança de estado do GP.

## 5.3 Armazenamento Estável

Para que os gerenciadores globais possam implementar o armazenamento estável, é necessário que se consiga transferir arquivos entre eles e qualquer outra estação da rede. Se o sistema operacional das estações não dispuser de nenhum recurso para a transferência de arquivos, um servidor para tal fim deve ser implementado e colocado em execução em cada estação. Não é interessante que os próprios processos do núcleo cuidem desta tarefa porque se assim fosse os serviços prestados às estações poderiam ser prejudicados.

Para cada processo de aplicação, o GP e os GS's possuem uma lista com os nomes de arquivos em armazenamento estável e a freqüência com que eles devem ser salvos.

Os pontos de recuperação são transferidos pelos gerenciadores das estações ao GP, que os guarda na memória principal e os replica nos GS's.

## 5.4 Comunicação

No GP, há para cada processo de aplicação uma caixa de correio para mensagens de aplicação e outra para as de sistema. Quando um processo de aplicação envia uma mensagem usando **send**, ela é transmitida para gerenciador da sua estação e daí para o GP. O GP vê em qual estação está o destinatário da mensagem e a coloca na sua caixa de correio (de aplicação ou de sistema), juntamente com o nome do emissor. O número da mensagem é registrado como sendo o maior usado até então pelo processo. Como visto, este número é necessário aos processos que estão se recuperando de erros, e é copiado em todos os GS's. Se a mensagem for de sistema, o gerenciador da estação do processo é alertado para que o avise assincronamente.

As caixas guardam enfileiradas (agrupadas por emissor) as mensagens, esperando que que elas sejam requisitadas pelos processos através de chamadas a **receive**. Esta função envia ao gerenciador da estação do processo um pedido de mensagem, que é então retransmitido ao gerenciador primário. Se a mensagem esperada não estiver disponível (isto só acontece com mensagens de aplicação, pois as de sistema são buscadas somente depois do aviso da sua chegada) e a chamada for bloqueante, o GP anota o nome do processo do qual se espera a mensagem para que, quando ela chegar, possa ser entregue de imediato. Neste caso ou no que o emissor especificado na chamada da função estiver (temporariamente) morto, a resposta ao pedido de mensagem é um código de erro. A mensagem solicitada ou o código de erro percorrem o caminho inverso do pedido até chegarem ao processo de aplicação.

O retorno a um ponto de recuperação pode fazer com que um processo perca as mensagens de aplicação recebidas desde a sua gravação [14]. Portanto, elas só podem ser removidas definitiva-

mente da caixa de correio quando um novo ponto de recuperação permanente for salvo, para que possam ser relidas pelo processo, se necessário.

Outro problema que pode acontecer com um processo que retorna a um estado anterior da sua execução é o envio de mensagens repetidas. Este problema poderia acontecer se um processo enviasse uma mensagem e, antes que ela fosse recebida, falhasse e tornasse a enviá-la. O GP identifica todas as mensagens repetidas que são enviadas e as descarta. Para isso, usa duas outras informações, que também estão nos GS's, sobre cada processo: o valor do contador de encarnações [20] [9]e o número da última mensagem enviada antes da gravação do seu ponto de recuperação mais recente.

O contador de encarnações é colocado em 1 quando o processo é iniciado pela primeira vez e daí para frente vai sendo incrementado a cada retorno seu a um ponto de recuperação. Em cada mensagem enviada por um processo, o GP anota o seu número de encarnação. Uma mensagem é identificada como repetida se o número de encarnação nela anotado for menor que a encarnação atual do processo que a enviou e se ela foi enviada após o processo ter gravado o seu último ponto de recuperação permanente, ou seja, se o número de seqüência da mensagem é maior que o número da última mensagem enviada antes do salvamento do ponto de recuperação.

## 5.5 Tolerância a Falhas do Gerenciador

O GP devolve aos gerenciadores de estações todas as mensagens "estou vivo" que deles recebe. Se a máquina em ele está pára de funcionar, os GL's e GS's vão perceber a falha do GP quando as mensagens deixarem de ser ecoadas por um período maior do que um determinado atraso aceitável.

Os gerenciadores locais e globais secundários executam, então, um mesmo algoritmo para determinar qual GS deve ser eleito o novo GP. Este algoritmo deve se basear nas identificações das estações dos GS's, pois essa é a única informação que os GL's possuem com relação aos GS's. Pode ser eleito, por exemplo, o GS cuja estação tenha a menor identificação numa ordem numérica ou lexicográfica, dependendo do tipo das identificações. Escolhido o novo gerenciador primário, todos os demais a ele se conectam. O insucesso na tentativa de conexão significa que o novo GP também falhou e que um novo deve ser escolhido, repetindo-se a operação acima descrita. Caso não haja mais nenhum GS, o núcleo falha e todos os processos de aplicação são mortos pelos GL'S sobreviventes, que em seguida abortam.

Um outro aspecto que deve ser observado é o da consistência das informações guardadas nos processos gerenciadores. As informações são propagadas entre eles de modo a garantir que o GP seja sempre o repositório da imagem mais atualizada de todas as aplicações e que esta imagem esteja replicada em todos os GS's. Isto é conseguido da seguinte maneira. Quando um GL ou GS deseja modificar o seu estado e esta mudança influi no estado global do núcleo, ele primeiro solicita que a modificação seja feita no GP lhe enviando uma mensagem. O GP retransmite a mensagem para todos os GS's e então envia ao seu emissor uma confirmação, que somente após recebê-la atualiza o seu estado. Este esquema é ilustrado na figura 3.

Ainda há, porém, um problema que pode ocorrer. Se o GP falhar após ter atualizado o estado de alguns GS's mas não de todos, os estados deles, e portanto o do núcleo, estarão inconsistentes. A solução neste caso é baseada nos protocolos *stop and wait* [21]. Os processos gerenciadores incrementam um contador módulo 2 a cada vez que recebem uma confirmação de recebimento de mensagem do GP. Caso a confirmação de uma mensagem não chegue dentro de um certo tempo, ela é retransmitida. O valor do contador é enviado juntamente com cada mensagem transmitida ao GP. Os GS's também mantêm um contador deste tipo para cada GL e GS do núcleo, registrando o número módulo 2 da última mensagem deles recebida e aceita. Se o número de uma mensagem que chega a um GS for diferente do da última mensagem recebida do mesmo gerenciador, ela é aceita e o respectivo contador é incrementado; caso contrário, a mensagem é descartada.

## 5.6 Informações Utilizadas pelos Processos Gerenciadores

É feito agora um resumo de todas as informações que devem ser mantidas pelos GL's, GP e GS's para que possam desempenhar suas funções.

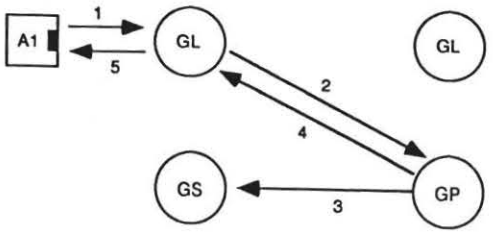


Figura 3: Propagação de informações de estado entre os processos gerenciadores

- **GL**

  **Tabela de Processos Locais**
  <aplicação, processo,
  acesso a conexão ponto a ponto com o processo>

  **Tabela de GS's**
  <estação>

  **Número módulo 2 da última mensagem enviada**

  **Contador de Atraso do GP**

  **Acesso a conexão ponto a ponto com o GP**

- **GP**

  **Tabela de Processos Locais**
  vide acima.

  **Tabela Global de Processos**
  <aplicação, processo, linha de comando, estação,
  morto ou não, encarnação, ponto de recuperação permanente,
  último número de mensagem antes do ponto de recuperação,
  último número de mensagem de aplicação,
  caixa de correio de mensagens de aplicação,
  caixa de correio de mensagens de sistema,
  bloqueado por mensagem de qual processo,
  lista de arquivos em armazenamento estável>

  **Tabela de Estações**
  <estação, carga, contador de atraso,
  tipo de gerenciador (GL ou GS),
  acesso a conexão ponto a ponto ao gerenciador da estação>

- **GS**

  **Tabela de Processos Locais**
  vide acima.

  **Tabela Global de Processos**
  vide acima.

**Tabela de Estações**

<estação, tipo de gerenciador (GL ou GS),
número módulo 2 da última mensagem aceita>

**Contador de Atraso do GP**

**Acesso a conexão ponto a ponto com o GP**

## 6 Conclusões e Perspectivas Futuras

Neste trabalho foi apresentada a arquitetura de um núcleo de programação distribuída cuja principal característica é a tolerância a falhas. São empregadas duas técnicas retroativas de recuperação de erros em conjunto. Uma, otimista, baseia-se na gravação de conjuntos consistentes de pontos de recuperação; na outra, pessimista, um processo guarda réplicas do seu estado em outros processos.

Defeitos de hardware e mortes inesperadas de processos foram as falhas previstas. Um outro tipo de falha, porém, também pode ser identificado: a sobrecarga de estações. Se uma estação não estiver conseguindo dar o andamento desejado à execução de um processo de aplicação nela sendo executado, é interessante considerar que esse processo falhou e migrá-lo para outra máquina que esteja mais ociosa, melhorando assim o balanceamento de carga na rede. Estudos serão feitos para a incorporação da tolerância a este tipo de falha à arquitetura do núcleo, pois, como está proposta, ela só prevê a migração de processos que morrem.

O núcleo será implementado para um ambiente composto por redes de estações Unix rodando protocolos de comunicação da família TCP/IP. Testes já realizados apontam para o uso de soquetes ao invés da TLI [19] como interface com a camada de transportes na implementação das ligações entre os gerenciadores, por causa de limitações desta com relação ao número máximo de conexões ponto a ponto que podem ser abertas simultaneamente, fator importante para a escalabilidade do sistema.

## Referências

[1] Massimo Ancona et al. A System Architecture for Fault Tolerance in Concurrent Software. *Computer*, 23(10):23–32, October 1990.

[2] Algirdas Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery on Database Systems*. Addison-Wesley, Reading, Mass., 1987.

[4] Anita Borg et al. Fault Tolerance under Unix. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[5] Wilton S. Caldas et al. SPD: Um Núcleo de Programação Distribuída em Redes de Computadores. In *Anais do IV Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho*, pages 445–457, Sao Paulo, 1992.

[6] R. H. Campbell and B. Randell. Error Recovery in Assynchronous Systems. *IEEE Transactions on Software Engineering*, SE-12(8):811–826, August 1986.

[7] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[8] Roger S. Chin and Samuel T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[9] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[10] J. Gray et al. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223–242, June 1981.

[11] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, USA, 1993.

[12] Frans M. Kaashoek and Andrew S. Tanenbaum. Fault Tolerance Using Group Communication. *ACM Operating Systems Review*, 25(2):71–74, April 1991.

[13] K. H. Kim. Programmer-Transparent Coordination of Recovering Concurrent Processes: Philosophy and Rules for Efficient Implementation. *IEEE Transactions on Software Engineering*, 14(6):810–821, June 1988.

[14] Richard Koo and Sam Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[15] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[16] Victor P. Nelson. Fault-Tolerant Computing: Fundamental Concepts. *Computer*, 23(7):19–25, July 1990.

[17] Dorgival O. G. Neto and Osvaldo S. F. Carvalho. Um Núcleo Linda para o Desenvolvimento de Aplicações Distribuídas em uma Rede Unix. In *Anais do X Simpósio Brasileiro de Redes de Computadores*, pages 573–585, Recife, 1992.

[18] Özalp Babaoglu. Fault Tolerant Computing Based on Mach. *ACM Operating Systems Review*, 24(1):27–39, January 1990.

[19] W. Richard Stevens. *Unix Network Programming*. Software Series. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1990.

[20] Robert E. Strom and Shaula Yemini. Optmistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[21] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, Englewood Cliffs, New Jersey 07632, second edition, 1989.

[22] Andrew S. Tanenbaum and Robert van Renesse. Distributed Operating Systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[23] Marvin Theimer and Keith A. Lantz. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engineering*, 15(11):1444–1458, November 1989.

[24] John Turek and Dennis Shasha. The Many Faces of Consensus in Distributed Systems. *Computer*, 25(6):8–16, June 1992.