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

Clairton Buligon , Sérgio Cechin , Ingrid Jansch-Pôrto

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

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

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

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

1. Introdução

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

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

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

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

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

Implementar um protocolo de recuperação por retorno em sistemas distribuídos

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

2. Modelo de sistema distribuído

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

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

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

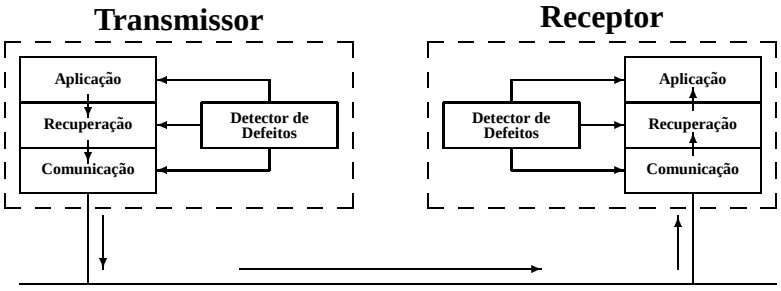

Figura 1: Modelo de computação distribuída

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

3. Algoritmo utilizado

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

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

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

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

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

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

4. Ambiente de implementação

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

especificar o ambiente de desenvolvimento do algoritmo.

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

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

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

4.1. Sistema operacional

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

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

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

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

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

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

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

4.2. Nível de *kernel* ou nível de usuário

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

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

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

4.3. Ferramenta `crak`

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

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

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

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

4.4. Implementação restante

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

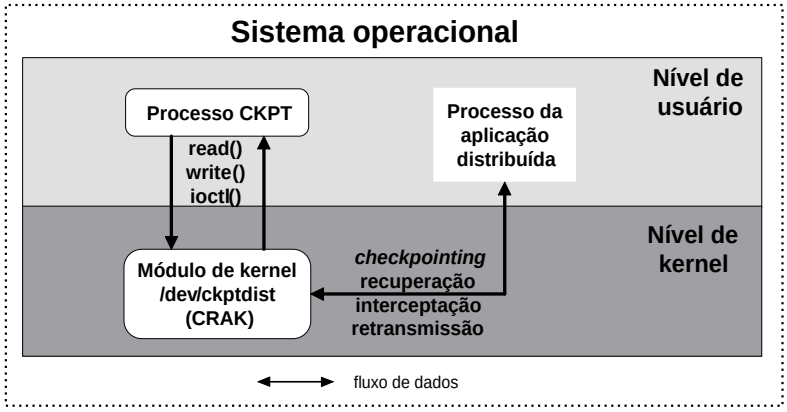

Figura 2: Estrutura da implementação sobre sistema operacional Linux

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

5. Implementação

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

5.1. Estrutura geral

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

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

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

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

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

5.2. Interceptação de mensagens

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

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

5.3. Salvamento de estados e recuperação em caso de falhas

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

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

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

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

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

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

Figura 3: Código-fonte para interceptação de mensagens

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

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

5.4. Demais mecanismos

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

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

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

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

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

6. Conclusão e trabalhos futuros

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

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

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

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

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

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

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

Referências

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

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

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

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

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

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

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

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

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

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

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

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

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