Checkpointing Quase-Síncrono no LAM/MPI

Ulisses Furquim Freire da Silva* Islene Calciolari Garcia

Universidade Estadual de Campinas
Caixa Postal 6176
13083-970 Campinas, SP, Brasil
Tel: +55 19 3788 5845 Fax: +55 19 3788 5847

E-mail: {ulisses.silva, islene}@ic.unicamp.br

Resumo

Atualmente, na área de computação de alto desempenho, um número crescente de aplicações distribuídas utiliza alguma biblioteca MPI (Message Passing Interface) para a troca de mensagens. Desse modo, há uma crescente demanda por mecanismos de tolerância a falhas para aplicações que utilizem esse sistema de comunicação. Nesse artigo, é discutida uma infra-estrutura para checkpointing quase-síncrono feita numa implementação livre do padrão MPI como base para a construção de um sistema tolerante a falhas que utilize recuperação por retrocesso de estado.

1. Introdução

Aplicações de várias áreas como programação inteira, genômica, processamento de imagens, aerodinâmica, meteorologia e outras têm se beneficiado da utilização de computação de alto desempenho, por meio de processamento distribuído. Normalmente, essas aplicações executam por longos períodos de tempo e a ocorrência de uma falha obriga o reinício do processamento. Assim, como a execução prolongada combinada com a utilização de recursos computacionais distribuídos aumenta a possibilidade de falhas durante o processamento, há uma necessidade muito grande de prover tolerância a falhas para aplicações distribuídas.

Uma maneira de minimizar os prejuízos de falhas em processamentos distribuídos seria gravar *checkpoints* globais consistentes da aplicação [12] durante a sua execução e reiniciar o processamento a partir do *checkpoint* global consistente gravado mais recente em caso de falha [14]. Nesse sistema tolerante a falhas baseado em recuperação por retrocesso de estado, o mecanismo de obtenção de *checkpoints* (chamado de *checkpointing*) é encarregado de garantir a existência de *checkpoints* globais consistentes.

Aplicações distribuídas são constituídas de vários processos espalhados pelos recursos computacionais existentes, e que se comunicam utilizando algum sistema de troca de mensagens. Nos últimos anos, o MPI [4] tem se estabelecido como um padrão de biblioteca de comunicação utilizado em aplicações distribuídas. O objetivo deste artigo é discutir o projeto e a implementação de uma infra-estrutura para *checkpointing* quase-síncrono [20] na biblioteca MPI conhecida como LAM [3] (*Local Area Multicomputer*). Também será descrita nesse artigo, uma arquitetura de *software* para recuperação de falhas por retrocesso de estado, que irá utilizar a infra-estrutura para *checkpointing* quase-síncrono já implementada.

Apesar de existir uma implementação do algoritmo de Chandy e Lamport [12] no LAM/MPI [23], este não é controlado pelos processos da aplicação, mas sim externamente. Além disso, sendo um protocolo de *checkpointing* síncrono, este também interrompe a execução normal da aplicação. Assim, protocolos de *checkpointing* quase-síncronos tornam-se atrativos, uma vez que não interrompem a execução normal da aplicação e também permitem que a aplicação controle a gravação de alguns *checkpoints* chamados de básicos.

Esta seção introduziu a motivação e o objetivo desse artigo. A Seção 2 introduz *checkpointing* e as diferenças entre as suas abordagens. Na Seção 3 são apresentados alguns trabalhos relacionados. A Seção 4 descreve o ambiente LAM/MPI e a implementação existente do algoritmo de *snapshot* distribuído de Chandy e Lamport. Na Seção 5 é descrita a infra-estrutura para *checkpointing* quase-síncrono que foi implementada no

* Apoio parcial do CNPq e, atualmente, da FAPESP (processo no. 03/01525-8)

LAM/MPI. A Seção 6 descreve uma arquitetura para recuperação de falhas chamada CURUPIRA que irá utilizar a infra-estrutura descrita na Seção 5. Por fim, a Seção 7 traça algumas considerações finais.

2. *Checkpointing*

Existem vários algoritmos na literatura para a construção de *checkpoints* globais consistentes, que podem ser classificados segundo três abordagens: assíncrona, síncrona e quase-síncrona. Estas abordagens diferem na autonomia dada aos componentes da aplicação na seleção dos *checkpoints* e na garantia de existência de um *checkpoint* global consistente diferente daquele representado pelos estados iniciais dos processos da aplicação.

A abordagem assíncrona oferece total autonomia para os processos da aplicação na seleção dos *checkpoints*. Porém, não há garantias quanto a formação de um *checkpoint* global consistente a partir dos *checkpoints* selecionados. Este problema foi detectado por Randell no contexto de recuperação de falhas por retrocesso de estado [22]. Em um cenário denominado *efeito dominó*, uma aplicação pode ser obrigada a retroceder ao seu estado inicial, apesar de ter gravado *checkpoints* ao longo da sua execução.

A abordagem síncrona garante a obtenção de um *checkpoint* global consistente por meio da propagação de mensagens de controle e da interrupção temporária da execução dos processos [12, 19]. Esta abordagem permite uma certa autonomia na seleção de *checkpoints*, mas exige sincronização de todos os processos, interrompendo a execução normal da aplicação. Um algoritmo síncrono é o de Chandy e Lamport [12], que grava um *snapshot* distribuído da aplicação, tratando também do recebimento de todas as mensagens em trânsito no momento da gravação do *snapshot*.

Por outro lado, a abordagem quase-síncrona permite aliar uma total autonomia na seleção de *checkpoints* com a garantia de que um *checkpoint* global consistente da aplicação poderá ser formado. Os processos selecionam *checkpoints* de maneira autônoma, denominados *checkpoints* básicos, mas eventualmente podem ser induzidos a selecionar *checkpoints* adicionais, denominados *checkpoints* forçados, de acordo com informações de controle propagadas juntamente com as mensagens da aplicação. Esta verificação da necessidade de gravar um *checkpoint* forçado está representada na Figura 1. No diagrama espaço-tempo desta figura, os processos selecionam seus *checkpoints* básicos (quadrados pretos), e em determinado momento, o processo p_1 recebe uma mensagem de p_2, e de acordo com as informações de controle obtidas da mensagem, o processo p_1 pode ser obrigado a gravar um *checkpoint* forçado (quadrado hachurado).

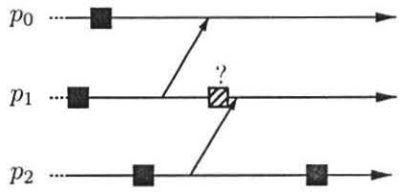

Figura 1. Possível indução de um checkpoint forçado em um protocolo quase-síncrono

Assim, percebe-se que os protocolos de *checkpointing* quase-síncronos deixam a aplicação executar normalmente e sem interrupções, permitindo sincronizar os processos apenas em caso de falhas, de modo que cada processo retroceda para o seu *checkpoint* pertencente ao *checkpoint* global consistente mais recente. Existem vários algoritmos quase-síncronos, dentre os quais pode-se citar o FDAS [28] (*Fixed-Dependency-After-Send*).

3. Trabalhos relacionados

Na literatura, pode-se encontrar implementações de *checkpointing* utilizadas para depuração distribuída, migração de processos, recuperação de falhas por retrocesso de estado. Entre estes trabalhos, o ambiente Condor [2] usa um protocolo de *checkpointing* síncrono para migrar processos e realizar balanceamento de carga nos recursos computacionais utilizados. Além disso, é possível gravar *checkpoints* periódicos da aplicação, de modo que a computação possa ser retrocedida para algum desses estados globais consistentes gravados na ocorrência de uma falha.

As implementações de *checkpointing* feitas para suportar aplicações que utilizam MPI diferem por serem feitas ou em cima da biblioteca de passagem de mensagens ou dentro dela. O CoCheck [26] é um exemplo de implementação feita sobre a biblioteca MPI, que usa um protocolo de *checkpointing* síncrono baseado no algoritmo de Chandy e Lamport [12] para realizar migração de processos em um sistema distribuído. Já o MPICH-V [5, 11] e o MPICH-GF [29] implementam os protocolos de *checkpointing* dentro da biblioteca MPI, com o propósito de oferecer recuperação de falhas por retrocesso de estado às aplicações. O MPICH-V pode utilizar um protocolo assíncrono de *checkpointing* ou o protocolo síncrono de Chandy e Lamport, enquanto o MPICH-GF utiliza um protocolo síncrono similar ao de Chandy e Lamport. O MPICH-GF se inte-

gra ao Globus [6], para prover tolerância a falhas para as aplicações que rodam nesse ambiente.

Além desses trabalhos, convém mencionar o RENEW [21], que é uma ferramenta para implementação e teste de desempenho de protocolos de *checkpointing* e de recuperação por retrocesso de estado. O RENEW exporta para as aplicações boa parte da API do MPI e permite implementar protocolos síncronos, assíncronos e quase-síncronos de *checkpointing*.

Percebe-se que as implementações existentes de protocolos de *checkpointing* ou têm o enfoque principal em migração de processos e podem ser adaptadas para fornecer recuperação de falhas por retorocesso de estado, como o Condor [2] e o CoCheck [26], ou têm o foco em recuperação de falhas por retrocesso de estado, mas implementam protocolos síncronos, como o MPICH-V [5, 11] e o MPICH-GF [29]. Além disso, nesses ambientes, não é fácil a implementação de algum outro protocolo de *checkpointing*. O RENEW [21], por sua vez, oferece um modo fácil de implementar qualquer protocolo de *checkpointing*, mas o seu objetivo é testar novos protocolos, e não ser um ambiente que ofereça tolerância a falhas para aplicações distribuídas usando MPI.

4. *Snapshot* distribuído no LAM/MPI

A implementação livre LAM do padrão MPI fornece uma biblioteca para troca de mensagens e um ambiente para controlar a execução de aplicações distribuídas. Antes de executar aplicações, o ambiente precisa estar rodando em todas as máquinas do sistema distribuído utilizado. Para realizar essa tarefa, é utilizado o comando *lamboot*, que inicia um *daemon* chamado *lamd* em cada uma das máquinas. Após esse passo, aplicações previamente compiladas com um dos compiladores do LAM podem ser executadas usando-se o comando *mpirun*, que se encarrega de transferir o código para cada uma das máquinas e rodá-lo. Assim, uma aplicação chamada *Teste* com 3 processos no LAM/MPI pode ser representada como na Figura 2. Os *daemons lamds* estão conectados uns aos outros, assim como os processos da aplicação, que podem, então, comunicar-se diretamente.

Ainda na Figura 2, percebe-se alguns módulos próprios da arquitetura do LAM/MPI. Os módulos CRLAM (*Checkpoint/Restart* LAM), CRMPI (*Checkpoint/Restart* MPI) e RPI (*Request Progression Interface*) são responsáveis por partes específicas no ambiente e possuem, normalmente, implementações utilizando diferentes tecnologias. Assim, ao executar o comando *mpirun*, os módulos para as funções de CRLAM, CRMPI e RPI podem ser previamente escolhidos na execução de uma determinada aplicação. O módulo chamado de MPI corresponde à API do MPI, que implementa primitivas como MPI_Send(), MPI_Recv() e outras.

Os módulos CRLAM e CRMPI fazem parte da implementação do algoritmo de Chandy e Lamport presente no LAM/MPI [23]. Tais módulos coordenam o processo de gravação de um *snapshot* distribuído da aplicação, que posteriormente pode ser aproveitado na reexecução da aplicação. Nesta implementação, quem controla a seleção dos *snapshots* é o usuário ou algum programa externo à aplicação, que usando o comando *lamcheckpoint* requisita a gravação de um *snapshot* distribuído de toda a aplicação. O comando *lamrestart* permite que uma aplicação seja reexecutada a partir de algum *snapshot* distribuído armazenado.

A requisição do comando *lamcheckpoint* é enviada ao módulo CRLAM do processo *mpirun*, que por sua vez cria um esquema da aplicação distribuída que poderá ser usado para a sua reexecução, e envia um sinal a todos os processos da aplicação. Este sinal é recebido pelo módulo CRMPI de cada um dos processos, que então inicia a sincronização dos processos para que todas as mensagens em trânsito sejam recebidas e os estados sejam gravados.

O módulo RPI presente nos processos da aplicação é responsável pela comunicação entre eles e está implementado em várias tecnologias de comunicação entre processos como: *sockets* TCP, *sockets* UDP, memória compartilhada, etc. Além da função de comunicação, o módulo RPI também participa do processo de gravação do *snapshot* distribuído, garantindo que as mensagens em trânsito sejam todas recebidas.

Os módulos CRLAM e CRMPI não gravam os estados dos processos, apenas coordenam a criação do *snapshot* distribuído. A gravação propriamente dita dos estados dos processos é feita por algum mecanismo externo ao LAM/MPI e que é controlado pelos módulos CRLAM e CRMPI. Um dos mecanismos utilizados no LAM/MPI para a gravação dos estados dos processos é o pacote chamado BLCR [1, 13] (*Berkley Labs Checkpoint/Restart*). Este pacote de *software* é composto por alguns módulos para o *kernel* Linux [7] e uma biblioteca que padroniza a gravação de estados de processos. O BLCR também fornece transparência na gravação do estado de um processo, pois os módulos inseridos no *kernel* Linux permitem o acesso direto às estruturas de dados e a sua gravação em um arquivo. Assim, existe uma implementação dos módulos CRLAM e CRMPI chamada de `blcr` (para ambos os módulos), que utiliza o pacote BLCR para a gravação dos estados dos processos. Os módulos `blcr` aliados ao módulo RPI chamado de `crtcp`, que usa *sockets* TCP para a

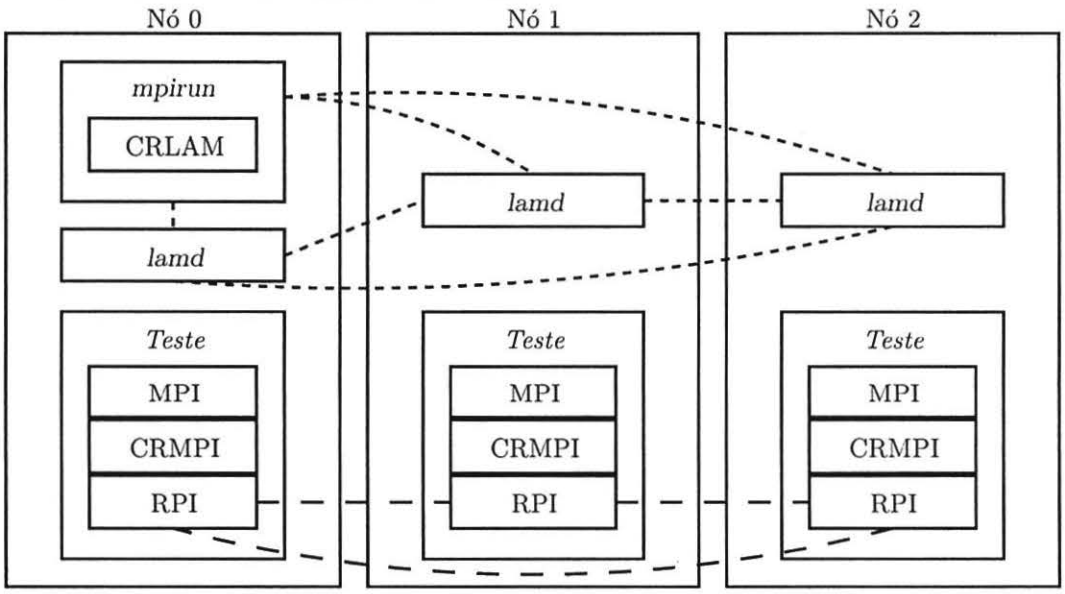

Figura 2. Aplicação Teste com 3 processos executando no LAM/MPI

comunicação, permitem que *snapshots* distribuídos sejam criados de aplicações distribuídas de forma transparente.

5. *Checkpointing* quase-síncrono no LAM/MPI

A implementação da infra-estrutura para *checkpointing* quase-síncrono dentro do LAM/MPI foi feita alterando-se os módulos MPI, CRMPI e RPI para criar um outro módulo chamado de CKPTQS (*ChecKPoinTing* Quase-Síncrono). Como optou-se por continuar a utilizar o BLCR para gravar os estados dos processos, mais precisamente, foram alterados os módulos `blcr` (CRMPI), `crtcp` (RPI) e a API do MPI. Estas alterações estão representadas na Figura 3, que ilustra um processo da aplicação *Teste* no LAM/MPI modificado. Nesta infra-estrutura, o módulo CRLAM não é utilizado, e apesar do comando *mpirun* continuar sendo executado no Nó 0, aquele comando foi retirado da ilustração apenas para simplificar a figura.

Antes de implementar a infra-estrutura, uma modelagem dos protocolos de *checkpointing* quase-síncronos foi feita, de modo a saber quais as necessidades e operações comuns a todos eles. Assim, um conjunto de ações foi definido como sendo suficiente para implementar os algoritmos de *checkpointing* quase-síncronos. Estas ações e o contexto onde elas são utilizadas estão representados na Figura 4.

Percebe-se, na Figura 4, que o módulo CKPTQS separa as ações específicas do protocolo de *checkpointing* quase-síncrono das demais partes do LAM/MPI. Desse modo, para utilizar um protocolo de *checkpointing* quase-síncrono é preciso apenas implementar as suas ações específicas.

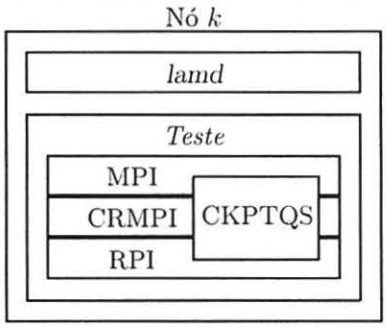

Figura 3. Processo da aplicação Teste executando no LAM/MPI modificado

As alterações feitas na API do MPI estão intimamente relacionadas com as ações: "Inicializa protocolo", "Finaliza protocolo" e "Grava *checkpoint* básico". A primeira precisa ser executada quando o MPI é inicializado em cada processo. Isto ocorre com a chamada da primitiva MPI_Init(), que foi alterada para inicializar o módulo CKPTQS, que por sua vez executa a ação "Inicializa protocolo" do algoritmo de *checkpointing* quase-síncrono utilizado. A segunda ação precisa ser executada quando o MPI é finalizado em cada um dos processos da aplicação. Isto é feito pela pri-

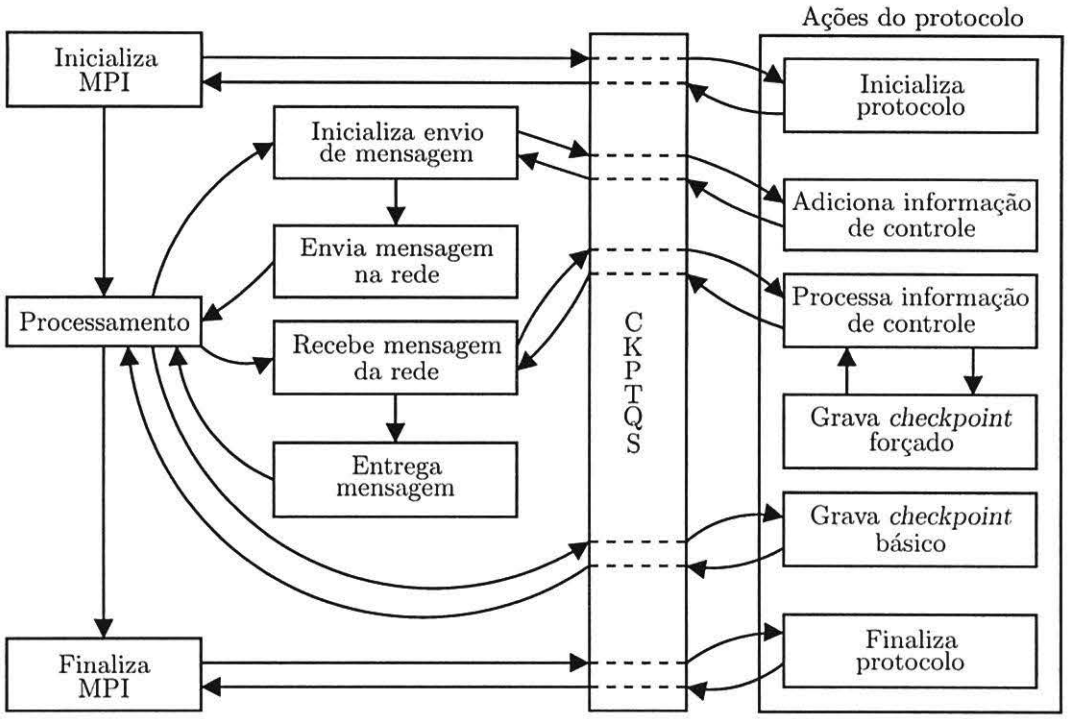

Figura 4. Ações de um protocolo de checkpointing quase-síncrono no LAM/MPI modificado

mitiva MPI_Finalize(), que foi alterada para chamar a função de finalização do CKPTQS, que por sua vez executa a ação "Finaliza protocolo" do algoritmo quase-síncrono. A terceira ação corresponde, na verdade, a uma extensão da API do MPI para incluir a primitiva MPI_Checkpoint(), que ao ser executada, faz com que o módulo CKPTQS execute a ação "Grava *checkpoint* básico" do protocolo de *checkpointing* quase-síncrono.

No módulo `blcr` (CRMPI), alterações foram feitas apenas para inicializar o pacote BLCR, ação que antes era feita em outro módulo do LAM/MPI. A biblioteca do BLCR também precisou ser alterada para que o CKPTQS pudesse ter um controle maior na gravação dos estados dos processos em arquivos.

Por fim, as alterações feitas no módulo `crtcp` (RPI) estão relacionadas com as ações: "Adiciona informação de controle", "Processa informação de controle" e "Grava *checkpoint* forçado". A primeira ação é utilizada quando uma mensagem vai ser enviada para outro processo. Nesse momento, a ação "Adiciona informação de controle" do algoritmo de *checkpointing* é executada para adicionar à mensagem a informação de controle pertinente ao protocolo utilizado. Depois desse passo, a mensagem será enviada pela rede para o processo de destino. A segunda ação será executada pelo módulo CKPTQS quando uma mensagem for recebida de outro processo. Nesse instante, a informação de controle será processada pelo protocolo de *checkpointing*, que poderá executar a ação "Grava *checkpoint* forçado" caso seja necessário. Posteriormente, a mensagem será entregue para a aplicação.

A implementação atual foi feita usando-se a versão 7.0.6 do LAM/MPI, que atualmente é a última versão estável dessa biblioteca, e a versão 0.2.1 do pacote BLCR. Porém, a infra-estrutura implementada apresenta algumas limitações, principalmente, por utilizar o pacote BLCR para a gravação dos estados dos processos. Nessa operação, as informações sobre arquivos abertos ou *sockets* não são levadas em consideração. Deste modo, aplicações que utilizem arquivos ou comuniquem-se com outros processos sem utilizar as primitivas do MPI não terão seu estado completamente gravado. É factível supor que aplicações distribuídas que utilizem o LAM/MPI para a troca de mensagens, não utilizam *sockets* diretamente. Porém, como muitas dessas aplicações utilizam arquivos, uma versão do pacote BLCR com suporte a arquivos deverá ser utilizado em versões futuras da infra-estrutura.

6. Arquitetura de *software* Curupira

O CURUPIRA é uma arquitetura de *software* que irá utilizar a infra-estrutura para protocolos de *checkpointing* quase-síncronos implementada no LAM/MPI

juntamente com as funções de coleta de lixo e recuperação por retrocesso em um protótipo para fornecer tolerância a falhas para algumas aplicações distribuídas.

Em particular, será utilizado no CURUPIRA um protocolo quase-síncrono que obedece à propriedade RDT [28] (*Rollback-Dependency Trackability*). Protocolos dessa classe facilitam a construção de *checkpoints* globais consistentes a partir de um conjunto de *checkpoints*. Além disso, o retrocesso em caso de falha é menor [8] e foi mostrado recentemente que estes protocolos também permitem que a coleta de lixo seja feita de maneira autônoma pelos processos [24, 25].

Protocolos RDT induzem um número maior de *checkpoints* forçados que outras classes de protocolos quase-síncronos [27]. Em decorrência desse fato, pode-se verificar na literatura da área um esforço direcionado a reduzir este custo. Baldoni, Helary e Raynal propuseram uma condição que seria minimal para garantir a propriedade RDT em tempo de execução [9]. Eles também desenvolveram um protocolo que implementa esta condição com complexidade $O(n^2)$, onde n é o número de processos da aplicação [10].

Entretanto, foi provado que a condição minimal para garantir a propriedade RDT em tempo de execução era na realidade bem mais simples [16] e pode ser implementada com complexidade $O(n)$ [15, 17]. O protocolo que implementa esta condição minimal é chamado RDT–minimal e será utilizado no CURUPIRA.

Além do protocolo de *checkpointing*, limitações de espaço em memória estável tornam a atividade de coleta de lixo importante em arquiteturas que se proponham a tolerar falhas. Assim, para permitir a recuperação por retrocesso, apenas os *checkpoints* que poderão ser utilizados em uma eventual falha da aplicação necessitam ser guardados. Os demais *checkpoints* são considerados obsoletos e podem ser descartados.

Como o CURUPIRA irá utilizar um algoritmo de *checkpointing* quase-síncrono e a característica dos protocolos com essa abordagem é a inexistência de um processo coordenador, é natural adotar uma política de coleta de lixo que possua as mesmas características. Até há pouco tempo, achava-se que a única maneira de se realizar a coleta de lixo seria de forma centralizada. Porém, recentemente, foi provado que a coleta de lixo pode ser feita de forma autônoma quando acoplada a protocolos da classe RDT [24, 25]. O espaço necessário para armazenamento local é de no máximo n *checkpoints* por processo, sendo que o armazenamento global também é proporcional a n^2. O algoritmo proposto é denominado RDT–LGC (RDT–*Local Garbage Collection*) e utiliza a informação propagada por vetores de dependência para determinar quais *checkpoints* ainda podem ser necessários. Este algoritmo também será utilizado na implementação do CURUPIRA.

Para completar a arquitetura, um mecanismo simples de recuperação por retrocesso será implementado. Esta é uma atividade que pode ser feita de forma centralizada por qualquer processo da aplicação. Para tanto, o *checkpoint* global consistente mais recente deverá ser calculado e todos os processos deverão concordar em retroceder para o seu *checkpoint* pertecente ao corte global calculado.

O protocolo de recuperação será similar ao protocolo de validação de duas fases (2PC—*Two Phase Commit*) utilizado para coordenar a finalização de transações atômicas [18, pp. 562-572]. Na primeira fase, o processo coordenador da recuperação pede a todos os outros que retrocedam. O coordenador decide pelo retrocesso se, e somente se, todos os outros concordam em retroceder. Na segunda fase, o coordenador calcula a linha de recuperação (*checkpoint* global consistente mais recente) e a sua decisão é propagada e executada por todos os processos.

A Figura 5 é uma representação em alto nível dos módulos do CURUPIRA dentro de um processo de uma aplicação distribuída. O LAM/MPI não está representado nessa figura apenas para não deixá-la muito complicada, mas o CURUPIRA será implementado dentro dele. Naquela figura, é possível perceber que o módulo CKPTQS fará parte do módulo do CURUPIRA chamado de Coordenação, responsável pela separação dos módulos de *Checkpointing*, Coleta de Lixo e Recuperação do código da própria Aplicação.

Os módulos de *Checkpointing* e Coleta de Lixo são responsáveis pela implementação dos algoritmos RDT–minimal e RDT–LGC, respectivamente. Na Figura 5, aqueles dois módulos não estão completamente separados, pois a implementação do algoritmo RDT–LGC também pode ser feita juntamente com o protocolo de *checkpointing* quase-síncrono da classe RDT. Por fim, o módulo de Recuperação ficará encarregado do algoritmo de recuperação por retrocesso de estado, que será executado na ocorrência de uma falha.

Após a implementação de todos os módulos do CURUPIRA, pretende-se realizar testes comparativos entre a implementação de *checkpointing* síncrono existente no LAM/MPI e a solução de *checkpointing* quase-síncrono provida pelo CURUPIRA. Uma aplicação de teste deverá ser projetada e implementada, de modo que as vantagens e desvantagens de cada uma das soluções possam ser melhor analisadas.

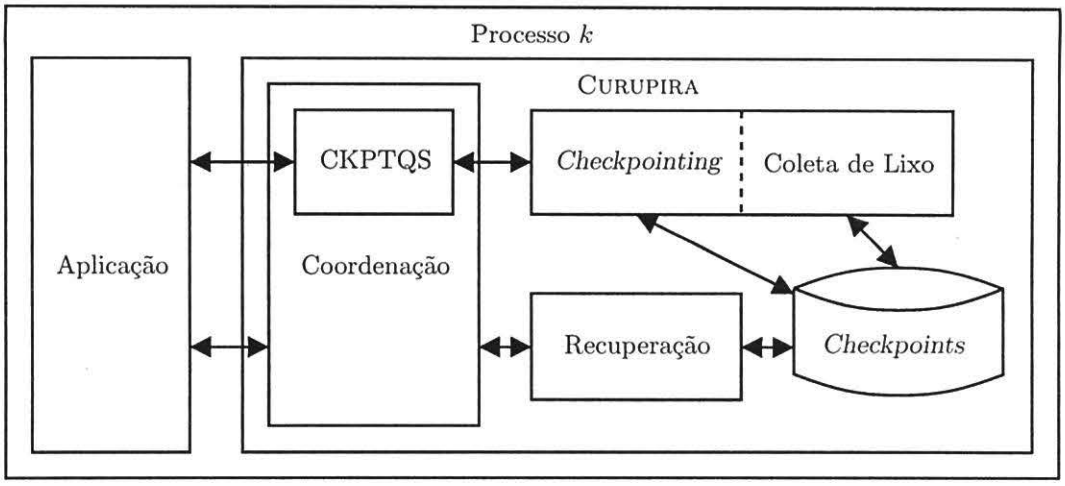

Figura 5. Processo de uma aplicação distribuída com o Curupira

7. Considerações finais

Nos últimos anos, os sistemas computacionais de alto desempenho têm evoluído muito, principalmente com o aparecimento de grandes sistemas distribuídos, como *clusters* e *grids*. As aplicações que rodam nesses sistemas também vêm evoluindo e apresentam além de uma alta complexidade, códigos cada vez mais extensos. Todos esses fatores aumentam a possibilidade de um falha interromper a execução de algum processamento distribuído. Tendo isso em mente, estudos em sistemas distribuídos tolerantes a falhas têm se intensificado, e esse artigo apresentou uma infra-estrutura para *checkpointing* quase-síncrono que foi implementada dentro da biblioteca LAM/MPI [3], que é muito utilizada para a construção de aplicações distribuídas.

Embora exista uma solução baseada num algoritmo de *checkpointing* síncrono implementada no LAM/MPI, esta não oferece muita autonomia para a aplicação na seleção dos *checkpoints*. Além disso, não há suporte para a implementação de algum outro algoritmo de *checkpointing* de maneira fácil. Assim, a infra-estrutura para *checkpointing* quase-síncrono apresentada oferece maior flexibilidade na implementação de novos protocolos e também maior autonomia para a aplicação na gravação dos seus estados. Além disso, embora o propósito dessa infra-estrutura seja a sua utilização numa arquitetura para recuperação de falhas por retrocesso de estado, esta infra-estrutura também pode ser utilizada por aplicações de outras áreas, como depuração distribuída.

Além da infra-estrutura para *checkpointing* quase-síncrono, este artigo também apresentou o modelo de uma arquitetura de *software* para recuperação de falhas por retrocesso de estado. Essa arquitetura, chamada CURUPIRA, será implementada utilizando-se a infra-estrutura para *checkpointing* quase-síncrono apresentada nesse artigo. Além disso, ela pode ser considerada completamente original, pois a utilização do algoritmo de coleta de lixo RDT-LGC [24, 25] permite que durante uma execução sem falhas, nenhuma mensagem de controle precise ser trocada pelos processos da aplicação. Desse modo, o funcionamento da arquitetura influencia o mínimo possível na execução da aplicação e permite oferecer alguma tolerância para as falhas ocorridas durante o processamento.

Referências

[1] Berkeley Lab Checkpoint/Restart (BLCR). Homepage oficial: `http://ftg.lbl.gov/twiki/bin/view/FTG/CheckpointRestart`. (consultado em 06/08/2004).

[2] Condor Checkpointing. Homepage oficial: `http://www.cs.wisc.edu/condor/checkpointing.html`. (consultado em 06/08/2004).

[3] LAM/MPI Parallel Computing. Homepage oficial: `http://www.lam-mpi.org/`. (consultado em 06/08/2004).

[4] Message Passing Interface Forum. Homepage oficial: `http://www.mpi-forum.org/`. (consultado em 06/08/2004).

[5] MPICH-V. Homepage oficial: `http://www.lri.fr/~gk/MPICH-V/`. (consultado em 06/08/2004).

[6] The Globus Alliance. Homepage oficial: `http://www.globus.org/`. (consultado em 06/08/2004).

[7] The Linux Kernel Archives. Homepage oficial: `http://www.kernel.org/`. (consultado em 06/08/2004).

[8] A. Agbaria, H. Attiya, R. Friedman, and R. Vitenberg. Quantifying rollback propagation in distributed checkpointing. In *Proceedings of the 20th Symposium on Re-*

liable Distributed Systems, pages 36–45, New Orlenas, 2001.

[9] R. Baldoni, J. M. Helary, and M. Raynal. Rollback-dependency trackability: Visible characterizations. In *18th ACM Symposium on the Principles of Distributed Computing*, Atlanta, Estados Unidos, May 1999.

[10] R. Baldoni, J. M. Helary, and M. Raynal. Rollback-dependency trackability: A minimal characterization and its protocol. *Information and Computation*, 165(2):144–173, Mar. 2001.

[11] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *SuperComputing 2002*, Baltimore, Nov. 2002.

[12] M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computing Systems*, 3(1):63–75, Feb. 1985.

[13] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Publicação eletrônica disponível em: `http://ftg.lbl.gov/twiki/pub/Whiteboard/CheckpointPapers/blcr.pdf`, 2003. (consultado em 06/08/2004).

[14] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 3(34):375–408, September 2002.

[15] I. C. Garcia. *Visões Progressivas de Computações Distribuídas*. PhD thesis, Instituto de Computação—Unicamp, Dec. 2001.

[16] I. C. Garcia and L. E. Buzato. On the minimal characterization of rollback-dependency trackability property. In *Proceedings of the 21th IEEE Int. Conf. on Distributed Computing Systems*, Phoenix, Arizona, EUA, Apr. 2001.

[17] I. C. Garcia and L. E. Buzato. An Efficient Checkpointing Protocol for the Minimal Characterization of Operational Rollback-Dependency Trackability. In *23rd Symposium on Reliable Distributed Systems*, Oct. 2004.

[18] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[19] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. *IEEE Trans. on Software Engineering*, 13:23–31, Jan. 1987.

[20] D. Manivannan and M. Singhal. Quasi-Synchronous Checkpointing: Models, Characterization, and Classification. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):703–713, 1999.

[21] N. Neves and W. K. Fuchs. RENEW: A Tool for Fast and Efficient Implementation of Checkpoint Protocols. In *Symposium on Fault-Tolerant Computing*, pages 58–67, 1998.

[22] B. Randell. System Structure for Software Fault Tolerance. *IEEE Trans. on Software Engineering*, 1(2):220–232, June 1975.

[23] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In *LACSI Symposium*, Oct. 2003.

[24] R. Schmidt, I. Garcia, F. Pedone, and L. Buzato. Optimal asynchronous garbage collection for checkpointing protocols with rollback-dependency trackability. In *23rd ACM Symposium on the Principles of Distributed Computing*, July 2004. (Brief Announcement).

[25] R. M. Schmidt. Coleta de Lixo para Protocolos de *Checkpointing*. Master's thesis, Instituto de Computação—Universidade Estadual de Campinas, 2003.

[26] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, Honolulu, Hawaii, 1996.

[27] G. M. D. Vieira. Estudo comparativo de algoritmos para *Checkpointing*. Master's thesis, Instituto de Computação—Universidade Estadual de Campinas, Dec. 2001.

[28] Y. M. Wang. Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints. *IEEE Trans. on Computers*, 46(4):456–468, Apr. 1997.

[29] N. Woo, H. Y. Yeom, and T. Park. MPICH-GF: Transparent Checkpointing and Rollback-Recovery for GRID-enabled MPI Processes. In *The 2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing(SHPSEC03)*, New Orleans, Louisiana, Sept. 2003.