Monitorização e Recuperação por Retrocesso utilizando Visões Progressivas de Computações Distribuídas

Islene Calciolari Garcia
Luiz Eduardo Buzato

Relatório Técnico IC–99-10

Março de 1999

Monitorização e Recuperação por Retrocesso utilizando Visões Progressivas de Computações Distribuídas

Islene Calciolari Garcia
Luiz Eduardo Buzato

Resumo

Sistemas distribuídos tolerantes a falhas baseiam-se parcialmente na existência de mecanismos de *checkpointing* e recuperação por retrocesso de estado. Um outro mecanismo importante para esses sistemas é o que permite a *monitorização* do seu estado e a pronta reação a mudanças que afetam o seu funcionamento previsto. Apesar desses dois mecanismos estarem fortemente associados, a grande maioria dos sistemas tolerantes a falhas contruídos até hoje privilegia a implementação de mecanismos de *checkpointing*, em detrimento de mecanismos para monitorização. *Visões progressivas* são seqüências de *checkpoints* globais consistentes que poderiam ter ocorrido nesta ordem durante a execução das computações. Foram denominadas progressivas porque o algoritmo para a sua determinação minimiza o retrocesso necessário durante a fase de recuperação do sistema, em caso de falha parcial de hardware. Neste artigo, comentamos a utilidade de visões progressivas para o desenvolvimento de protocolos mais eficientes para *checkpointing* e para a integração de mecanismos de *checkpointing*, recuperação de estado e monitorização.

Abstract

Fault-tolerant distributed systems are partially based on the implementation of checkpointing and rollback-recovery mechanisms. Another important mechanism associated to tolerance of faults is the one that allows a system to monitor its state and to react to exceptional behaviour. Checkpointing and rollback-recovery are already part of most of the fault-tolerant systems implemented to date. This situation does not apply to monitoring mechanisms, especially to systems that integrate monitoring, checkpointing and rollback-recovery. Progressive views are formed by a sequence of consistent global checkpoints that may have occurred in this order during the execution of the system. Progressive views are called progressive because they have been designed to minimize the rollback of the system when a partial failure of hardware occurs. This article dicusses the application of progressive views towards the deployment of more efficient checkpointing mechanisms and its application to the integration of checkpointing, rollback-recovery and monitoring for fault-tolerant distributed systems.

1 Introdução

Vários mecanismos que permitem a adição de tolerância a falhas a um sistema distribuído estão baseados na seleção de estados (*checkpoints*) dos componentes deste sistema. O conjunto dos *checkpoints* selecionados forma uma abstração da computação executada pelo sistema, que é útil para tarefas como avaliação de predicados globais [8, 15] e recuperação de falhas por retrocesso de estado [10, 19, 28].

A avaliação de predicados sobre estados globais consistentes [7] pode ser utilizada na construção de algoritmos distribuídos para detecção de *deadlocks*, reconstrução de ficha perdida, coleta de lixo, além de monitorização e reconfiguração de componentes da aplicação [9]. A recuperação de falhas por retrocesso de estado permite que a aplicação possa restaurar um estado global consistente após a ocorrência de uma falha, de maneira a diminuir a quantidade de trabalho perdido [28]. Estes mecanimos podem se beneficiar da obtenção de visões progressivas de computações distribuídas, ou seja, seqüências de *checkpoints* globais consistentes [7] que poderiam ter ocorrido nesta ordem durante a execução das computações [11, 13, 14].

Recentemente, obtivemos três resultados nesta área: (i) o estabelecimento de uma relação entre *checkpoints* que determina uma ordem para a observação destes *checkpoints* [11, 14], (ii) a proposta de algoritmos originais para a construção de visões progressivas a partir de *checkpoints* [11, 14] e (iii) o mapeamento destes resultados para o modelo de objetos e ações atômicas [11, 12]. Neste artigo, vamos comentar a utilidade do conceito de visões progressivas para o desenvolvimento de protocolos mais eficientes para *checkpointing* e para a implementação de mecanismos para tolerância a falhas, incluindo a integração .de mecanismos de monitorização e de recuperação de falhas por retrocesso de estado.

Este documento está estruturado da seguinte forma. A Seção 2 faz um resumo do estado da arte em protocolos para *checkpointing*, comentando problemas em aberto na área. A Seção 3 introduz visões progressivas de computações distribuídas, ressaltando suas aplicações para *checkpointing* e avaliação de predicados instáveis. A Seção 4 apresenta duas arquiteturas de *software* para tolerância a falhas que se beneficiam da construção de visões progressivas. Finalmente, a Seção 5 encerra o trabalho.

2 Protocolos para *Checkpointing*

A escolha adequada de *checkpoints* pelos componentes de uma aplicação distribuída não é uma tarefa simples quando se necessita de estados globais consistentes desta aplicação [7, 26]. Nas próximas subseções, vamos comentar alguns resultados obtidos e problemas em aberto na área de *checkpointing*. Na Seção 3, argumentaremos que a utilização de visões progressivas de computações distribuídas pode contribuir para o melhor entendimento e o desenvolvimento de soluções mais eficientes para alguns dos problemas atuais nesta área.

2.1 Estados Globais Consistentes

Um estado global de um aplicação distribuída é formado pela união dos estados locais dos componentes desta aplicação. Informalmente, um estado global é *consistente* se corresponde

a um estado global que poderia ter sido obtido por um observador onisciente externo [7]. Existem vários algoritmos na literatura para a construção de estados globais consistentes [2, 4, 7, 10, 19, 22, 23, 32, 34, 36], que podem ser classificados segundo três abordagens [23]: (i) assíncrona, (ii) síncrona e (iii) quase-síncrona.

A primeira abordagem oferece autonomia máxima aos componentes de uma aplicação para a seleção de *checkpoints*. Em contrapartida, não há garantias quanto a formação de um estado global consistente a partir dos *checkpoints* selecionados. Este problema foi detectado por Randell no contexto de recuperação de falhas por retrocesso de estado [28]. Em um cenário denominado *efeito dominó*, uma aplicação pode ter de retroceder ao seu estado inicial após a ocorrência de uma falha, apesar de ter gravado *checkpoints* ao longo de sua execução. Outro inconveniente desta abordagem é o espaço total requerido para o armazenamento de *checkpoints*: Wang *et al.* determinaram que o limite máximo necessário é proporcional ao quadrado do número de componentes da aplicação [33].

A abordagem síncrona garante a obtenção de um estado global consistente e requer o armazenamento de apenas dois *checkpoints* por componente da aplicação [19]. No entanto, esta abordagem restringe a liberdade dos componentes para a escolha de *checkpoints*, apresenta um custo extra de comunicação e pode acarretar a suspensão das atividades dos componentes durante a sincronização [7, 19].

A abordagem quase-síncrona está baseada em um protocolo obedecido pelos componentes da aplicação para a seleção de *checkpoints*. Prioritariamente, os componentes selecionam *checkpoints* livremente, mas eventualmente podem ser induzidos a selecionar *checkpoints* adicionais de acordo com informações de controle propagadas através das mensagens da aplicação [10, 23]. Esta abordagem apresenta um compromisso entre a autonomia dos componentes para a escolha dos *checkpoints* e as garantias oferecidas para a formação de estados globais consistentes a partir dos *checkpoints* selecionados. Devido a esta característica, acreditamos que esta abordagem é a mais flexível para a implementação de mecanismos para tolerância a falhas. Na próxima Seção, faremos um resumo do estado da arte em protocolos quase-síncronos.

2.2 Protocolos Quase-Síncronos para *Checkpointing*

O conjunto de *checkpoints* selecionados em uma computação distribuída forma um *padrão* de *checkpoints* para esta computação. As características deste padrão são dependentes do protocolo quase-síncrono adotado [23]. Nesta Seção, vamos citar alguns protocolos quase-síncronos desenvolvidos para o modelo de processos e mensagens, e comentar alguns aspectos teóricos que influenciaram a avaliação dos padrões de *checkpoints* gerados por estes protocolos.

1. Protocolos baseados no Modelo MRS (*Mark-Receive-Send*) [29]. Estes protocolos induzem *checkpoints* para garantir um padrão de envio e recepção de mensagens, no qual todos os eventos de recepção de mensagens em um intervalo de *checkpoints* precedem os eventos de envio neste mesmo intervalo. Wang identificou quatro protocolos neste modelo: *No-Receive-After-Send*, *Checkpoint-After-Send*, *Checkpoint-Before-Receive*, *Checkpoint-After-Send-Before-Receive* [32].

2. Protocolo proposto por Venkatesh, Radhakrishnan e Li [18]. Este protocolo propaga informações de precedência causal entre *checkpoints* utilizando vetores de relógios [25]. O padrão de *checkpoints* resultante garante que as dependências causais são fixas em um intervalo de *checkpoints*.

3. Protocolo proposto por Wang [32]. Esta é uma otimização do protocolo anterior, na qual o padrão de *checkpoints* garante que as dependências causais são fixas apenas após o primeiro evento de envio de mensagem dentro de um intervalo de *checkpoints*.

4. Protocolo proposto por Baldoni et al. [1, 2]. Carrega informações adicionais sobre a causalidade entre *checkpoints* na forma de uma matriz de booleanos, de maneira a diminuir o número de *checkpoints* induzidos em relação ao protocolo anterior.

5. Protocolo proposto por Briatico, Ciuffoletti e Simoncini [5]. Atribui índices, semelhantes aos relógios lógicos propostos por Lamport [20], a *checkpoints* de forma que *checkpoints* com o mesmo índice podem fazer parte de um mesmo estado global consistente. De maneira análoga ao protocolo proposto por Venkatesh, Radhakrishnan e Li [18], o padrão de *checkpoints* resultante garante que os índices são fixos por intervalo e a otimização proposta por Wang [32] também se aplica a este caso [4].

6. Protocolo proposto por Baldoni, Quaglia e Fornara [4]. Esta é uma otimização do protocolo anterior. Procura adiar o incremento do índice baseando-se em informações adicionais de precedências contidas em um vetor de relógios relativo ao estado global consistente determinado pelo índice corrente.

7. Protocolo proposto por Baldoni, Quaglia e Ciciani [3, 27]. Garante que todos os *checkpoints* são úteis, ou seja, participam de pelo menos um *checkpoint* global consistente. Altamente custoso, controla a indução de *checkpoints* através da propagação de matrizes de relógios.

8. Protocolo proposto por Xu e Netzer [36]. Não garante a total ausência de *checkpoints* inúteis, mas faz um esforço para diminuir a sua ocorrência. Pode-se obter uma versão mais eficiente deste protocolo através do trabalho de Baldoni et al. [1, 2].

9. Protocolo proposto por Wang e Fuchs [34]. Variação do protocolo proposto por Briatico, Ciuffoletti e Simoncini [5] que induz *checkpoints* apenas quando os índices são múltiplos de um determinado valor. Apenas estes *checkpoints* são garantidamente úteis.

Um dos resultados teóricos mais importantes para a avaliação de protocolos quase-síncronos para *checkpointing* foi a determinação, por Netzer e Xu, das condições necessárias e suficientes para participação de um *checkpoint* em um *checkpoint* global consistente [26]. Este resultado está baseado em *zigzag paths*, que são seqüências, não necessariamente causais, de mensagens transmitidas entre intervalos de *checkpoints*. Considerando *zigzag paths*, Manivannan e Singhal obtiveram uma classificação para protocolos quase-síncronos, que relaciona a classe do protocolo ao número de *checkpoints* induzidos e à dificuldade de se obter um estado global consistente [23].

2.3 Problemas em Aberto

Dadas as características dos algoritmos para a obtenção de estados globais consistentes, enumeradas na Seção 2.1, a abordagem quase-síncrona parece ser bastante apropriada, tanto para a monitorização de aplicações distribuídas quanto para a recuperação de falhas por retrocesso de estado. No entanto, determinar qual protocolo quase-síncrono deve ser adotado não é uma tarefa simples, pois requer uma análise de custo/benefício que envolve vários fatores.

Existe uma distinção entre os *checkpoints* selecionados espontaneamente pela aplicação, denominados *checkpoints básicos*, e os *checkpoints* induzidos pelo protocolo. Em príncipio, um protocolo ideal deveria induzir o menor número possível de *checkpoints* e ainda assim garantir a utilidade de todos os *checkpoints* básicos. Infelizmente, a especificação deste protocolo parece ser impossível, pois dependeria de conhecimento sobre o futuro da execução da aplicação [23]. Cabe aos projetistas considerar um compromisso entre número de *checkpoints* induzidos, complexidade da manutenção da informação de controle e possibilidade de *checkpoints* inúteis. Não temos conhecimento de uma comparação cuidadosa envolvendo todos estes fatores e os protocolos citados na Seção 2.2. Algumas avaliações consideram apenas protocolos semelhantes [4, 32, 36] ou classes de protocolos [23].

Além dos fatores citados, uma outra preocupação deve ser considerada pelos projetistas de sistemas distribuídos tolerantes a falhas. A construção de *checkpoints* globais consistentes é uma tarefa ortogonal à escolha dos *checkpoints* pelos componentes da aplicação. Quando esta construção precisa ser feita de maneira incremental (adicionando-se *checkpoints* dos componentes um a um), só são conhecidos algoritmos eficientes para as classes de protocolos que induzem o maior número de *checkpoints* [21, 23]. Desta forma, a utilização de protocolos mais flexíveis requer maior cuidado na construção de *checkpoints* globais consistentes.

Dentro do contexto de recuperação de falhas por retrocesso de estado, um outro fator importante é o espaço necessário para o armazenamento de *checkpoints*. É desejável que os *checkpoints obsoletos*, ou seja, *checkpoints* anteriores à linha de recuperação da aplicação possam ser removidos, através de coleta de lixo. Wang *et. al* [33] determinaram qual é o limite máximo de espaço requerido e propuseram um algoritmo para a remoção de *checkpoints* obsoletos e inúteis. O resultado deles é voltado para a abordagem assíncrona e, portanto, inclui a quase-síncrona. No entanto, padrões gerados por protocolos quase-síncronos podem possuir limites inferiores de espaço requerido e/ou possibilitar algoritmos mais simples para a coleta de lixo.

3 Visões Progressivas de Computações Distibuídas

Uma visão progressiva de uma computação distribuída é formada por uma seqüência de estados (*checkpoints*) globais consistentes que poderia ter ocorrido nesta ordem durante a computação [14]. Nesta Seção, introduzimos visões progressivas de computações distribuídas e exploramos algumas áreas relacionadas à tolerância a falhas em que esta teoria pode ser aplicada. Nosso objetivo é demonstrar que esta abordagem pode contribuir para um melhor entendimento e para a obtenção de soluções melhores para problemas ligados a estados globais consistentes e protocolos quase-síncronos para *checkpointing*.

3.1 Z-Precedência entre *Checkpoints*

Intuitivamente, visões progressivas permitem que se compute um *checkpoint* global consistente mais recente partindo-se do último *checkpoint* global consistente conhecido. Em contraste, as abordagens exploradas tradicionalmente pelos algoritmos existentes na literatura fazem uso de uma das seguintes técnicas: (i) computam um *checkpoint* global consistente utilizando um conjunto inicial de *checkpoints* e o transformam em um *checkpoint* global consistente através da inclusão de *checkpoints* de outros processos [21] ou (ii) computam um *checkpoint* global consistente retrocedendo de um estado global inconsistente [10].

Devido à motivação de se construir visões progressivas de computações distribuídas, obtivemos uma re-interpretação das *zigzag paths* propostas por Netzer e Xu [26], que denominamos de *Z-precedência* entre *checkpoints* [14]. A Z-precedência é uma generalização da relação de precedência causal proposta por Lamport [20] cujo significado é que um *checkpoint* a Z-precede um checkpoint b caso a deva ser *observado antes* de b para todas as visões progressivas de uma determinada computação distribuída.

Ao contrário das *zigzag paths* que são definidas a partir de seqüências de mensagens, a Z-precedência é definida exclusivamente a partir de *checkpoints* e precedência causal entre *checkpoints*. Consideramos que esta escolha é coerente com o nível de abstração proporcionado pelos *checkpoints* (eventos e mensagens correspondem a uma abstração de nível mais baixo) e torna a Z-precedência facilmente aplicável a outros modelos computacionais. Em particular, aplicamos este conceito ao modelo de objetos e ações atômicas [35], estendendo nosso trabalho de construção assíncrona de estados globais consistentes neste modelo [11, 12].

Outra vantagem da Z-precedência é o fato dela ser formada por uma composição de relações causais entre *checkpoints*, sendo simples escrever provas de propriedades utilizando indução no tamanho desta composição. Esta característica nos ajudou a escrever as provas de correção para os algoritmos que propusemos para a construção de visões progressivas [11, 14].

Acreditamos que o conceitos de visão progressiva e Z-precedência também possam vir a ser úteis para o desenvolvimento de novos protocolos quase-síncronos para *checkpointing* e algoritmos para avaliação de predicados globais.

3.2 Desenvolvimento e Avaliação de Protocolos

Um dos principais pontos a ser considerado quando se desenvolve ou se avalia um protocolo quase-síncrono é o fato deste protocolo garantir ou não que todos os *checkpoints* selecionados sejam úteis (possam fazer parte de pelo menos um *checkpoint* global consistente). Uma abordagem empregada para se provar esta característica é demonstrar a possibilidade de se construir um *checkpoint* global consistente a partir de qualquer *checkpoint* do padrão [1]. No entanto, construir *checkpoints* globais consistentes de maneira incremental não é uma tarefa simples para todas as classes de protocolos quase-síncronos [21, 23].

Outra abordagem consiste em garantir que o padrão de *checkpoints* gerado por um protocolo quase-síncrono possui a propriedade de *precedência virtual* [17]. Tal propriedade garante que os *checkpoints* podem ser rotulados utilizando-se relógios lógicos que obedeçam às seguintes restrições: intervalos de *checkpoints* conectados por mensagens são rotulados

de maneira não-decrescente e o relógio lógico de um processo deve aumentar após um envio de mensagem. Uma terceira abordagem consiste em garantir que determinados padrões de comunicação que poderiam ocasionar *checkpoints* inúteis não ocorrem [3, 27, 29].

A construção de visões progressivas nos permitiu encontrar outra abordagem, que associa a ausência de *checkpoints* inúteis a um cenário global, denominado *passo* de uma visão progressiva. Um passo tem como objetivo partir de um *checkpoint* global consistente, $\hat{\Sigma}$, e construir um outro *checkpoint* global consistente, $\hat{\Sigma}'$, mais recente que o primeiro. Idealmente, gostaríamos de poder utilizar apenas os *checkpoints* que sucedem $\hat{\Sigma}$ de maneira imediata no padrão de *checkpoints* para a construção de $\hat{\Sigma}'$. Na ausência de *checkpoints* inúteis isto é sempre possível; caso contrário, os *checkpoints* inúteis devem ser descartados [14].

Desta forma, para provar que um protocolo quase-síncrono não permite a seleção de *checkpoints* inúteis, basta mostrar que, dado um *checkpoint* global consistente, o padrão de *checkpoints* gerado pelo protocolo sempre permite a execução de um passo sem descartes [11]. Este cenário engloba características das três abordagens anteriores, contribuindo para o estabelecimento de um arcabouço único para avaliação dos protocolos quase-síncronos presentes na literatura. Acreditamos que este resultado também possa vir a ser útil para a proposta de novos protocolos.

3.3 Avaliação de Predicados Instáveis

Visões progressivas de computações distribuídas podem ser facilmente utilizadas para a avaliação de predicados estáveis, enquanto sua aplicação para a avaliação de predicados instáveis requer cuidados adicionais. Predicados instáveis, de maneira geral, só podem ser avaliados na presença de um *reticulado* da computação [9], que contém todos os estados globais pelos quais a computação pode ter passado. Entretanto, quando se conhece a estrutura do predicado, algoritmos mais eficientes podem ser empregados [15].

Acreditamos ser possível a modificação dos nossos algoritmos para a construção de visões progressivas para que se possa obter um reticulado *resumido* da aplicação, ou seja, um reticulado formado apenas por *checkpoints* globais consistentes. Com a colaboração dos componentes da aplicação para a seleção de *checkpoints* adequados, o reticulado resumido poderá ser útil para avaliação eficiente de predicados pertencentes a algumas classes específicas de predicados instáveis. Uma classe de provável aplicação seria a de predicados globais instáveis formados pela conjunção de predicados locais.

4 Monitorização e Recuperação de Falhas

Nesta Seção, apresentamos duas arquiteturas de *software* para tolerância a falhas que se beneficiam dos resultados obtidos com a teoria e os algoritmos para construção de visões progressivas [11, 14]. A primeira delas é voltada para a monitorização de aplicações distribuídas, enquanto a segunda propõe a integração dos mecanismos de monitorização e recuperação de falhas por retrocesso de estado.

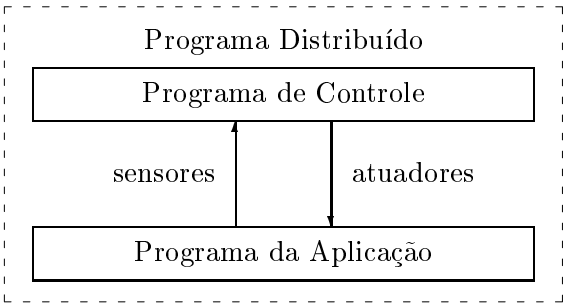

Figura 1: Arquitetura de *software* para monitorização.

4.1 Monitorização utilizando Visão Progressiva

Respeitando uma política de divisão de tarefas, consideramos que um *programa distribuído* é composto pela superposição de dois sistemas reativos: o *programa da aplicação* e o *programa de controle* (Figura 1) [6, 24]. O programa da aplicação implementa aspectos funcionais do programa distribuído, enquanto o programa de controle implementa aspectos de gerência e reconfiguração. Por exemplo, considere um sistema de arquivos replicados como um programa distribuído. Neste caso, o programa da aplicação implementa as funcionalidades básicas de sistemas de arquivos, enquanto o programa de controle é responsável pela gerência das réplicas.

Para facilitar a implementação do programa de controle, consideramos sua divisão em dois módulos (Figura 2): o *fotógrafo*, responsável pela construção progressiva de *checkpoints* globais consistentes [14] e o *monitor*, responsável pela avaliação de predicados globais [8] e atuação sobre a aplicação.

Os componentes do programa da aplicação podem cooperar com o programa de controle através da seleção de *checkpoints* (estados de interesse). Considerando o exemplo dado, um estado de interesse pode refletir um conjunto de alterações em uma réplica. Os componentes podem respeitar um protocolo quase-síncrono [23], de maneira a reduzir ou eliminar a ocorrência de *checkpoints* inúteis.

Um protótipo desta arquitetura para o modelo de processos e mensagens foi implementado em Java[1] [16, 30] por Gustavo Maciel Dias Vieira [31]. Cabe notar que esta arquitetura também é válida para outros modelos computacionais, como o modelo de objetos e ações atômicas [12].

4.2 Integração de Recuperação de Erros e Monitorização

Grande parte da teoria e dos protocolos quase-síncronos desenvolvidos para a obtenção de *checkpoints* globais consistentes foi projetada tendo a recuperação de falhas por retrocesso de estado como alvo [2, 10, 23, 34, 36]. Como vimos na Seção anterior, nossa abordagem inicial foi utilizar estes protocolos para a construção de visões progressivas para monitorização.

[1] Java is a trademark of Sun Microsystems, Inc.

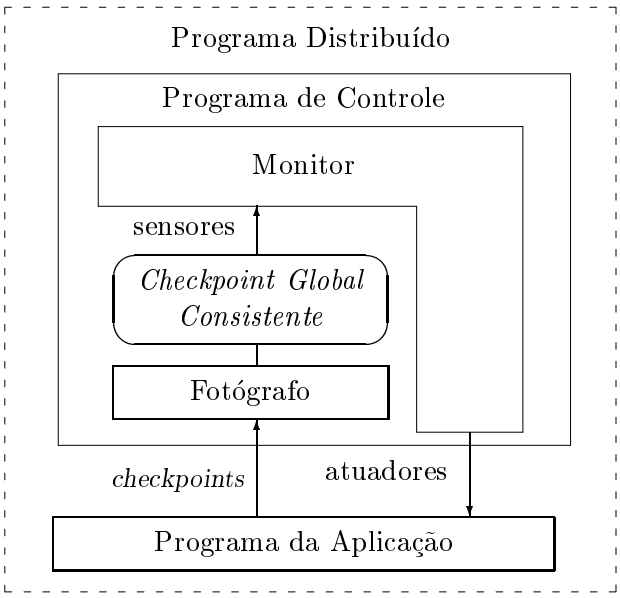

Figura 2: Arquitetura de *software* para monitorização utilizando visão progressiva

Atualmente, estamos investigando a possibilidade de integração destes dois mecanismos. Nosso alvo são aplicações que têm como requisitos básicos a necessidade de monitorização e de recuperação de falhas. A idéia básica é aproveitar o fato de que o monitor conhece um *checkpoint* global consistente para aliviar os componentes da aplicação de tarefas como construção de linhas de recuperação e coleta de lixo de *checkpoints* obsoletos.

A Figura 3 é uma especialização da arquitetura de *software* apresentada na Figura 2 que ilustra a integração entre os mecanismos. O fotógrafo constrói *checkpoints* globais consistentes e os envia para o programa de controle, que é composto pelos sistemas de monitorização e reconfiguração. Quando o sistema de recuperação é notificado de alguma falha, ele propaga as linhas de recuperação para que os componentes da aplicação possam fazer o retrocesso. Linhas de recuperação também são propagadas periodicamente para que possa ser feita a coleta de lixo de *checkpoints* obsoletos.

Uma comparação inicial da arquitetura de *software* proposta com as arquiteturas utilizadas normalmente deve levar em conta as abordagens mencionadas na Seção 2.1. Em adição ao já comentado sobre vantagens e desvantagens da adoção de cada uma dessas abordagens para a recuperação pode-se dizer:

- a abordagem síncrona tem as vantagens de recuperação simples e espaço para armazenamento de *checkpoints* mínimo;
- nas abordagens assíncrona e quase-síncrona a recuperação é mais complexa e involve os seguintes procedimentos: (i) computação do grafo de dependência [10] e (ii) determinação e propagação da linha de recuperação. A coleta de lixo também é mais complexa e sempre requer a computação do grafo de dependência.

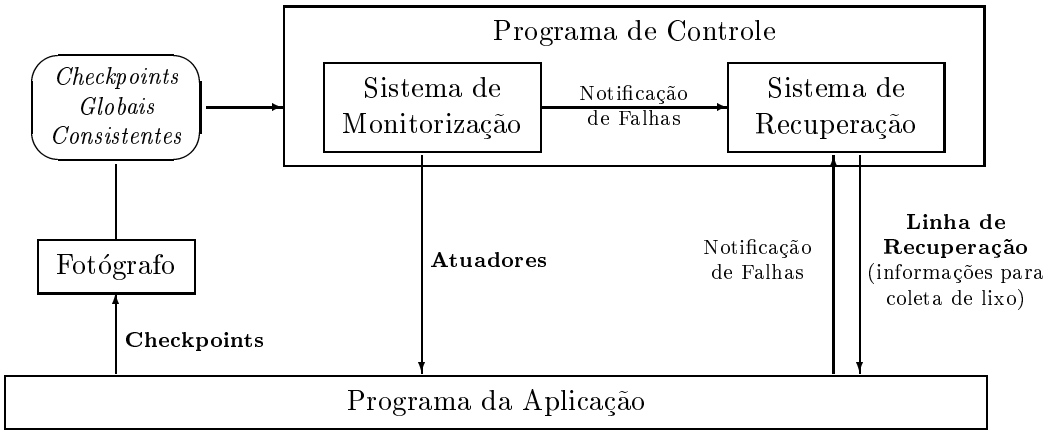

Figura 3: Integração entre monitorização e recuperação com retrocesso.

A arquitetura proposta procura reduzir as desvantagens da abordagem quase-síncrona, simplificando a recuperação e a gerência da coleta de lixo, enquanto preserva as vantagens da abordagem quase-síncrona sobre a síncrona para *checkpointing*. O processo de recuperação é simplificado pela introdução do fotógrafo e do monitor, que computam linhas de recuperação em paralelo à aplicação e podem propagá-las imediatamente após a detecção de uma falha. A construção progressiva de linhas de recuperação permite redução do espaço total utilizado para armazenar *checkpoints*. As desvantagens da nossa arquitetura são as seguintes. Na proposta corrente, o monitor é centralizado, podendo representar um gargalo em termos de processamento e um ponto fraco em termos de tolerância a falhas. Esta deficiência pode ser contornada através da replicação do monitor. Finalmente, o custo associado à manutenção do monitor e às mensagens adicionais pode representar uma desvantagem em relação as arquiteturas existentes.

5 Conclusão

Uma visão progressiva de uma computação distribuída é formada por uma seqüência de fotografias do estado da aplicação que poderia ter ocorrido nesta ordem durante a computação. Visões progressivas podem ser construídas a partir de *checkpoints* (estados selecionados pelos componentes da aplicação), formando uma abstração da computação executada. Em face de um resumo do estado da arte em protocolos para *checkpointing* e do levantamento de problemas em aberto na área, argumentamos que o conceito de visões progressivas pode permitir uma avaliação uniforme dos protocolos existentes e contribuir para o desenvolvimento de novos protocolos.

Visões progressivas também podem contribuir para a implementação de mecanismos para tolerância a falhas. Uma das nossas propostas é integrar sistemas de monitorização e sistemas de recuperação de falhas por retrocesso de estado, aproveitando as similaridades entre estes dois sistemas de modo a oferecer uma solução única, mais uniforme e eficiente.

Agradecimentos

Este trabalho recebeu apoio financeiro da Fapesp, processos número 95/1983-8 (Islene Calciolari Garcia) e 96/1532-9 (Laboratório de Sistemas Distribuídos), do CNPq, processo número 145563/98-7 (Islene Calciolari Garcia) e do convênio Pronex/Finep, processo número 76.97.1022.00 (projeto Sistemas Avançados de Informação).

Nossos especias agradecimentos a Gustavo Maciel Dias Vieira por ter implementado e testado nossas idéias e por ter contribuído com vários comentários interessantes.

Referências

[1] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. Consistent checkpointing in message passing distributed systems. Technical Report 2564, INRIA, June 1995.

[2] R. Baldoni, J. M. Helary, A. Mostefaoui, and M. Raynal. A communication-induced checkpoint protocol that ensures rollback dependency trackability. In *IEEE Symp. on Fault Tolerant Computing (FTCS'97)*, 1997.

[3] R. Baldoni, F. Quaglia, and B. Ciciani. A VP-accordant checkpointing protocol preventing useless checkpoints. In *17-th Symposium on Reliable Distributed Systems*, Purdue University, 1998.

[4] R. Baldoni, F. Quaglia, and P. Fornara. An index-based checkpoint algorithm for autonomous distributed systems. Technical Report 07-97, Universitá "La Sapienza", Roma, Italy, Mar. 1997.

[5] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *IEEE 4th Symp. on Reliability in Distributed Software and Database Systems*, pages 207–215, 1984.

[6] L. E. Buzato. *Management of Object-Oriented Action-Based Distributed Programs.* Ph.D. Thesis, University of Newcastle upon Tyne, Department of Computer Science, Dec. 1994.

[7] M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Computing Systems*, 3(1):63–75, Feb. 1985.

[8] R. Cooper and K. Marzullo. Consistent detection of global predicates. *SIGPLAN Notices*, 26(12):167–174, Dec. 1991.

[9] Ö. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

[10] E. N. Elnozahy, D. Johnson, and Y.M.Yang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, 1996.

[11] I. C. Garcia. Estados globais consistentes em sistemas distribuídos. Dissertação de mestrado, Instituto de Computação—Universidade Estadual de Campinas, julho 1998.

[12] I. C. Garcia and L. E. Buzato. Asynchronous construction of consistent global snapshots in the object and action model. In *4th IEEE Int. Conf. on Configurable Distributed Systems (ICCDS'98)*, Annapolis, Maryland, EUA, May 1998. Disponível como Relatório Técnico IC–98–16.

[13] I. C. Garcia and L. E. Buzato. Cortes consistentes em aplicações distribuídas. In *Segundo Workshop em Sistemas Distribuídos (Wosid'98)*, Curitiba, Paraná, Junho 1998. Disponível como Relatório Técnico IC–98–17.

[14] I. C. Garcia and L. E. Buzato. Progressive construction of consistent global checkpoints. Technical Report IC–98–36, Instituto de Computação—Universidade Estadual de Campinas, 1998. Aceito para publicação no ICDCS'99 (19th IEEE International Conference on Distributed Computing Systems), que ocorrerá no período de 31 de maio a 5 de junho, em Austin, Texas, EUA.

[15] V. K. Garg. Observation of global properties in distributed systems. In *IEEE Int. Conf. on Software and Knowledge Engineering*, pages 418–425, Lake Tahoe, Nevada, June 1996.

[16] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Java Series. Addison–Wesley, Sept. 1996. Version 1.0.

[17] J.-M. Helary, A. Mostéfaoui, and M. Raynal. Virtual precedence in asynchronous systems: Concept and applications. In *11th Int. Workshop on Distributed Algorithms (WDAG'97)*, pages 170–184, Saarbrücken, Germany, Sept. 1997.

[18] T. R. K. Venkatesh and H. F. Li. Optimal checkpointing and local recording for domino-free rollback recovery. *Information Processing Letters*, 25(5):295–303, 1987.

[19] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Trans. on Software Engineering*, 13:23–31, jan 1987.

[20] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[21] D. Manivannan, R. H. B. Netzer, and M. Singhal. Finding consistent global checkpoints in a distributed computation. In *IEEE Trans. on Parallel and Distributed Systems*, pages 623–627, June 1997.

[22] D. Manivannan and M. Singhal. A low-overhead recovery technique using quasi-synchronous checkpointing. In *16th Int. Conf. on Distributed Computing Systems*, pages 100–107, May 1996.

[23] D. Manivannan and M. Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. Technical Report OH 43210, Department of Computer and Information Science, The Ohio State University, 1997.

[24] K. Marzullo, R. Cooper, M. Wood, and K. Birman. Tools for distributed application management. Technical Report 90-1136, Cornell University, June 1990.

[25] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[26] R. H. B. Netzer and J. Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. on Parallel and Distributed Systems*, 6(2):165–169, 1995.

[27] F. Quaglia, R. Baldoni, and B. Ciciani. A checkpointing protocol based on a minimal characterization of the no-z-cycle property. Technical Report 01-98, Dipartimento di Informatica e Sistemistica, Universita' di Roma "La Sapienza", Italy, Jan. 1998.

[28] B. Randell. System structure for software fault tolerance. *IEEE Trans. on Software Engineering*, 1(2):220–232, June 1975.

[29] D. L. Russell. State restoration in systems of communicating processes. *IEEE Trans. on Software Engineering*, 6(2):183–194, Mar. 1980.

[30] Sun Microsystems Computer Corporation, Mountain View, CA, USA. *Java API Documentation*, Dec. 1996. Version 1.1.

[31] G. M. D. Vieira and L. E. Buzato. Determinação de estados globais consistentes em sistemas distribuídos. Technical Report IC–99–09, Instituto de Computação, Universidade Estadual de Campinas, Jan. 1999.

[32] Y. M. Wang. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Trans. on Computers*, 46(4):456–468, Apr. 1997.

[33] Y. M. Wang, P. Y. Chung, I. J. Lin, and W. K. Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Trans. on Parallel and Distributed Systems*, 6(5):546–554, May 1995.

[34] Y. M. Wang and W. K. Fuchs. Lazy checkpoint coordination for bounding rollback propagation. In *IEEE Symp. on Reliable Distributed Systems (SRDS'93)*, pages 78–85, Oct. 1993.

[35] S. M. Wheater and D. L. McCue. Configuring distributed applications using object decomposition in an atomic action environment. In *Int. Workshop on Configurable Distributed Systems*, pages 33–44. IEE (UK), Imperial College, UK, Mar. 1992.

[36] J. Xu and R. H. B. Netzer. Adaptive independent checkpointing for reducing rollback propagation. In *IEEE Symp. on Parallel and Distributed Processing*, pages 754–761, Dec. 1993.