Ambiente para a Visualização de Protocolos de *Checkpointing*

Relatório Parcial - Processo 03/00483-

Orientando: Raphael Marcos Menderico
Orientadora: Profa. Dra. Islene Calciolari Garcia

Outubro de 2003

Resumo

Checkpoints são utilizados em sistemas distribuídos para auxiliar tarefas como monitorização e recuperação por retrocesso de estado. No entanto, entender o comportamento de um protocolo de *checkpointing* pode ser uma tarefa bastante complexa visto que o seu funcionamento depende da interação entre vários processos. Nesse projeto, apresentamos a especificação e implementação de um ambiente que permita a visualização gráfica de protocolos para *checkpointing*. Esse ambiente auxilia a depuração dos protocolos durante a fase de implementação dos mesmos, além de servir como ferramenta de pesquisa, permitindo a exploração de detalhes do comportamento dos protocolos. O código-fonte desenvolvido para o ambiente será disponibilizado de acordo com a filosofia de *software* livre.

Palavras-chave: Algoritmos distribuídos, tolerância a falhas, *checkpointing*, visualização de algoritmos

1 Introdução

Soluções para vários problemas em sistemas distribuídos, como monitorização e recuperação por retrocesso de estado, estão baseadas na utilização de *checkpoints*. Um *checkpoint* é um estado de um processo que foi selecionado para ser gravado em memória estável. Um *checkpoint* global consistente representa uma fotografia da computação distribuída que poderia ter sido obtida por um observador onisciente externo [2]. Um protocolo de *checkpointing* coordena a seleção de *checkpoints* pelos processos de maneira a garantir a construção de *checkpoints* globais consistentes [3, 6].

Entender o comportamento de um protocolo de *checkpointing* pode ser uma tarefa complexa, visto que temos de levar em conta a interação entre vários processos. As descrições dos protocolos presentes na literatura estão sempre acompanhadas de diagramas espaço-tempo semelhantes aos ilustrados nas Figuras 1 e 2. Nestes diagramas, as linhas horizontais representam a execução dos processos com o decorrer do tempo, as setas menores representam as mensagens e os quadrados, *checkpoints*.

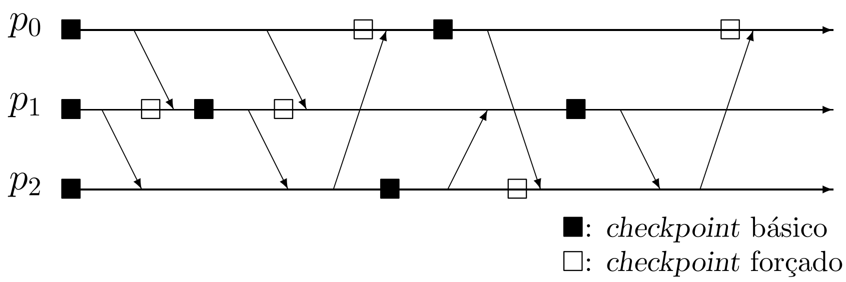

Figura 1: Diagrama espaço-tempo ilustrando um protocolo quase-síncrono

A Figura 1 ilustra um protocolo quase-síncrono de *checkpointing*, no qual podemos distinguir dois tipos de *checkpoints*: os que são selecionados de maneira autônoma pelos processos, denominados *checkpoints* básicos, e os que são induzidos pelo protocolo, denominados *checkpoints* forçados. No cenário exibido, os processos são induzidos a gravar um *checkpoint* forçado imediatamente antes de todo evento de recepção de mensagem [6]. Esta abordagem é muito simples, mas claramente induz um número bastante alto de *checkpoints* forçados. A literatura da área apresenta vários protocolos mais complexos que permitem reduções significativas no número de *checkpoints* forçados [7, 6].

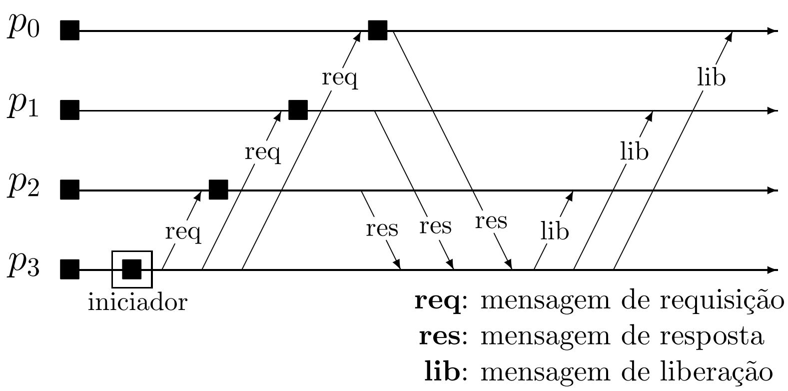

Figura 2: Diagrama espaço-tempo ilustrando um protocolo síncrono bloqueante

A Figura 2 ilustra o comportamento de um protocolo síncrono bloqueante [5]. Nesta classe de protocolos, a obtenção de um *checkpoint* global consistente é coordenada por meio da transmissão de mensagens de controle. Durante a fase de coordenação, os processos interrompem o processamento das mensagens da aplicação e cuidam apenas das mensagens de controle. Um processo denominado iniciador grava um *checkpoint* e envia mensagens de requisição para que os outros processos da aplicação também gravem *checkpoints*. Quando o processo iniciador recebe uma mensagem de cada um dos outros processos indicando que os *checkpoints* foram gravados, ele envia mensagens de liberação para todos eles, de modo que estes possam retomar o processamento da aplicação. Para a classe de protocolos síncronos também existem variações mais complexas descritas na literatura, inclusive protocolos que permitem que a coordenação seja feita sem a interrupção do processamento da aplicação [1].

O objetivo deste trabalho é apresentar a especificação e implementação de um ambiente para a visualização de protocolos de *checkpointing* que permita a construção de diagramas espaço-tempo a partir de uma execução real de uma aplicação distribuída. Esse visualizador auxilia a depuração de protocolos durante a fase de implementação dos mesmos, além de servir como ferramenta de pesquisa, permitindo a exposição de detalhes do comportamento global dos protocolos. É possível visualizar diferenças fundamentais de comportamento entre classes de protocolos e diferenças mais sutis entre protocolos de uma mesma classe. Além disso, os diagramas indicam qual é o padrão de mensagens gerado por uma determinada aplicação de maneira a auxiliar a escolha do protocolo mais adequado para esta aplicação.

O restante deste documento está estruturado da seguinte forma. A Seção 2 detalha o objetivo deste projeto, mostrando a metodologia empregada. A Seção 3 detalha as atividades realizadas durante o período e os resultados obtidos. A Seção 4 encerra esse documento descrevendo as atividades futuras e apresentando o cronograma revisado.

2 Objetivo

O objetivo deste trabalho é a especificação e implementação de um ambiente para a visualização de protocolos de *checkpointing* que permita a construção de diagramas espaço-tempo a partir de uma execução real de uma aplicação distribuída. Neste projeto, o diagrama espaço-tempo é construído estaticamente a partir de um arquivo de histórico gerado pela aplicação.

A Figura 3 ilustra a arquitetura do ambiente, que é composta por uma aplicação distribuída, um *monitor* e um *visualizador*. A aplicação distribuída é formada por um conjunto de processos $P_1 \ldots P_n$ que se comunicam por meio de troca de mensagens e que gravam periodicamente seus estados de acordo com algum protocolo de *checkpointing*. O processo *monitor* recebe dos processos os eventos de interesse para a visualização e os armazena em um arquivo *histórico*. O processo *visualizador* lê este arquivo e constrói um diagrama espaço-tempo referente à execução da aplicação.

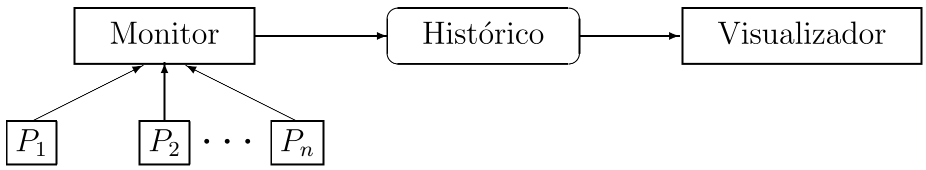

Figura 3: Arquitetura do ambiente

Para que esta arquitetura pudesse ser implementada facilmente, cada processo teve sua estrutura interna dividida em três partes principais, como mostrado na Figura 4: (i) camada de código específico da aplicação, (ii) camada de *checkpointing* e (iii) camada de comunicação. A camada de comunicação é responsável pelo envio e recebimento de mensagens, além de fornecer a infraestrutura básica para a comunicação com o *monitor* (os comandos necessários para realizar a coleta dos dados). A camada de *checkpointing* encapsula os métodos da camada de comunicação, para permitir que cada protocolo decida quando os *checkpoints* devem ser tirados. Na camada de código específico, uma aplicação é escrita utilizando os métodos disponibilizados pela camada de *checkpointing* para envio e recebimento de mensagens e gravação do estado. Esta estrutura permite que se possa testar uma mesma aplicação com vários protocolos e um mesmo protocolo com vários tipos de aplicação.

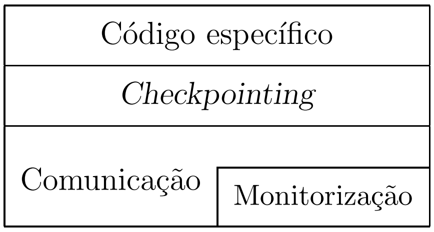

Figura 4: Estrutura de um processo da aplicação

O processo *monitor* tem como tarefa principal processar as mensagens vindas de todos os processos da aplicação e produzir o arquivo *histórico* que permite a construção do diagrama espaço-tempo pelo *visualizador*.

3 Atividades Realizadas

3.1 Implementação dos processos

A Figura 5 mostra o diagrama de classes utilizado para implementar as camadas de comunicação e *checkpointing*. A superclasse `mensagem` implementa a camada de comunicação, provendo a estrutura básica para que os processos possam enviar e receber mensagens entre si. Todas as trocas de mensagens ocorrem através do LAM-MPI, que provê um canal FIFO e confiável entre os processos e entre um processo e o *monitor*.

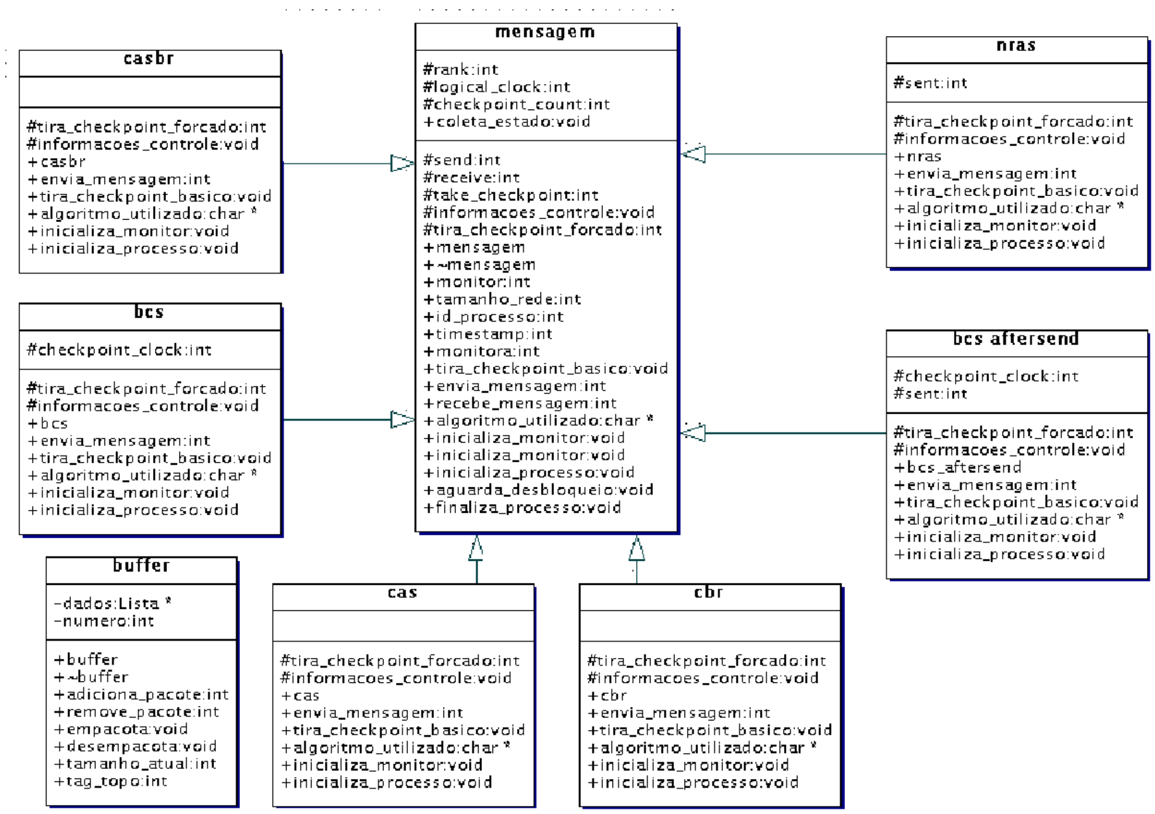

Figura 5: Diagrama de classes

A classe `mensagem` também é responsável por enviar os eventos de interesse para o monitor. Esses eventos são aqueles que contêm um checkpoint ou uma mensagem associada a eles. Para garantir a ordenação, um sistema de relógios lógicos foi implementado, com o *timespamp* sendo enviado de carona junto com as mensagens trocadas entre os processos. Esse relógio garante que as relações causais sejam atendidas na hora de criar a imagem a ser visualizada.

Uma outra classe auxiliar, chamada `buffer`, também atua na camada de comunicação, embora seja utilizada por todas as demais camadas. Essa classe permite o envio de dados de diversos tipos e tamanhos (no caso, por todos os tipos básicos aceitos pelo LAM-MPI) em uma única mensagem, como se fosse um pacote. Assim, os dados de controle podem ser enviados utilizando uma única chamada de método, independente do conteúdo da mensagem original. O tratamento de tipos é feito pela API do LAM-MPI, enquanto que o empacotamento é realizado pela classe, para garantir que o processo que recebe a mensagem consiga ler todo o seu conteúdo.

A camada de checkpointing corresponde a um conjunto de classes, cada uma implementando um protocolo diferente. Essa implementação é feita através de herança, a partir da superclasse `mensagem`, que possui métodos virtuais que são sobrescritos por cada uma das classes-filho. Os protocolos fazem as chamadas dos métodos herdados da superclasse para enviar e receber mensagens, alterando esses métodos para incluir as regras inerentes a cada protocolo (como alteração de variáveis e a obtenção de um *checkpoint* forçado), tornando a implementação independente do sistema de comunicação implementado.

Cada algoritmo também é responsável por informar a camada inferior quais dados que devem ser transportados de carona em cada mensagem e quais as informações inerentes ao protocolo que devem ser coletadas quando um *checkpoint* for tirado, sem que a camada superior tenha conhecimento sobre o que ocorre internamente ao algoritmo, permitindo a troca do protocolo sem interferir na aplicação que está sendo executada. Até o momento, foram implementados os protocolos quase-síncronos BCS, BCS Aftersend, CASBR (Checkpoint-After-Send-Before-Receive), CAS (Checkpoint-After-Send), CBR (Checkpoint-Before-Receive) e NRAS (No-Receive-After-Send) [7, 6].

A aplicação específica implementada para testar o sistema gerencia um *token* único em um sistema distribuido e envia mensagens aleatórias para os processos enquanto não estão com o token. Ela foi escolhida por ser uma aplicação simples, mas que gera *checkpoints* forçados em diversos algoritmos, permitindo testar as funcionalidades das camadas inferiores. As aplicações específicas chamam os métodos disponibilizados pela camada de *checkpointing* e devem fornecer uma forma de coletar o seu estado, seja porque ela solicitou o armazenamento de um checkpoint básico ou porque a camada inferior decidiu tirar um checkpoint forçado.

3.2 Monitorização e criação do histórico

Um processo especial, chamado monitor, recebe todas as mensagens de controle enviadas pelos processos, analisa os dados enviados e escreve um arquivo de histórico, para posterior avaliação pelo visualizador. Cada tipo de evento especificado possui uma sintaxe diferente. Por enquanto, as opções de exibição são escritas no mesmo arquivo. O monitor utiliza a superclasse `mensagem` para receber os eventos, já que as mensagens com os dados não passam pelas camadas superiores e, portanto, não possuem os relógios e as informações de controle. Um exemplo do arquivo de histórico construido está na Figura 6.

```
DEF FALSE
PROC 0
PROC 1
PROC 2
CHK 2 0 0 FALSE 3
ESTADO 30000 1 0
ESTADO 2 1 0
ESTADO 1 1 0
CHK 1 0 0 FALSE 3
ESTADO 30000 1 0
ESTADO 2 1 0
ESTADO 1 1 0
CHK 0 0 0 FALSE 3
ESTADO 30000 1 0
ESTADO 2 1 0
ESTADO 1 1 0
MSG 0 1 1 2
CHK 1 1 3 FALSE 3
ESTADO 30000 1 1
ESTADO 2 1 2
ESTADO 1 1 1
```

Figura 6: Exemplo de arquivo de histórico

3.3 Visualização

A última etapa é a visualização do arquivo de histórico. A primeira etapa do visualizador é realizar a interpretação do arquivo. Segundo o protocolo estabelecido, cada linha determina um único comando, mas podem existir comandos que dependam do comando da linha anterior. Assim, uma máquina de estados foi implementada para interpretar o arquivo e transformá-lo em um conjunto de objetos que descreva o desenho adequadamente. A última etapa é desenhar os objetos na tela, relacionando-os com os objetos criados, para que as informações adequadas possam ser apresentadas quando o usuário interagir com o diagrama (percorrê-lo com o mouse ou clicar sobre um checkpoint). Exemplos de imagens obtidas com o visualizador são mostrados nas Figuras 7 e 8.

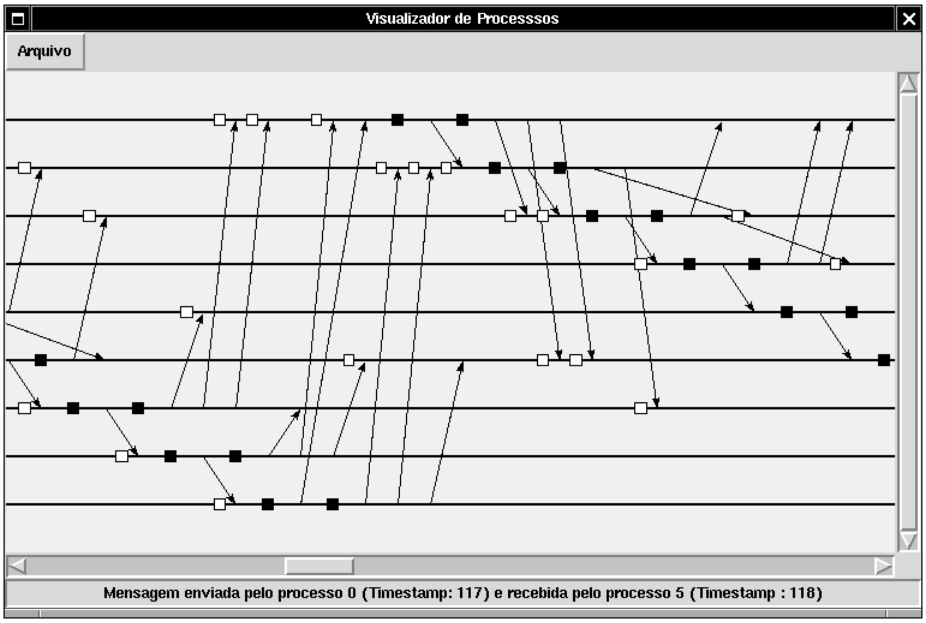

Figura 7: Imagem gerada na execução do protocolo BCS

3.4 Dificuldades encontradas

Uma das dificuldades encontradas durante a implementação foi a conversão de tipos entre os dados. No princípio, constatou-se que dados heterogêneos deveriam ser enviados, pois a

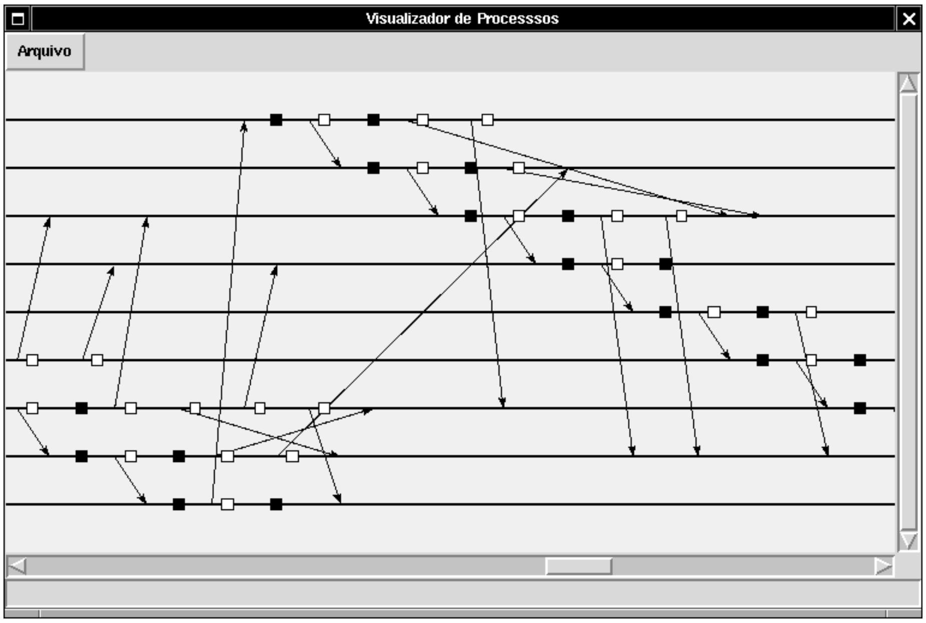

Figura 8: Imagem gerada na execução do protocolo CAS

aplicação poderia enviar informações de todos os tipos possíveis nas mensagens e o relógio lógico era obrigatoriamente um número. Uma primeira tentativa foi converter todos os tipos para um tipo comum (no caso, caracter), mas problemas nessa conversão (como a identificação de um inteiro qualquer como fim de sequência) fizeram com que essa estratégia fosse abandonada.

O próximo passo foi utilizar as bibliotecas de conversão do LAM-MPI, mas, para respeitar o encapsulamento proposto pelas classes, seria necessário que somente a camada inferior (responsável pela comunicação) conhecesse essas bibliotecas. Assim, se fosse necessário, por qualquer motivo, trocar o meio de comunicação, essa troca poderia ser feita sem interferir nas aplicações já escritas. Para isso, foi implementada a classe `buffer` (descrita acima) que realiza a conversão junto com a camada de comunicação.

Outro problema foi encontrado na hora da detecção pelos processos do término da aplicação. Um processo qualquer fica bloqueado quando uma mensagem é enviada por ele para outro processo que não está executando mais. Essa é uma característica do *middleware* utilizado (LAM-MPI). Dessa forma, os processos devem entrar em consenso de que nenhuma mensagem está nos canais, para posteriormente encerrarem a sua execução. Se o fizerem antes, ao menos um processo não conseguirá parar o seu processamento, provocando erros. A solução, nesse caso, foi utilizar uma abordagem semelhante ao algoritmo de Chandy-Lamport [2] para garantir que os canais estejam sem nenhuma mensagem com entrega pendente.

4 Cronograma e Atividades Futuras

A Tabela 1 mostra o cronograma de trabalho atualizado, indicando as tarefas já concluidas e as pendentes. Até o momento, os trabalhos seguiram o cronograma e por isso ele não foi alterado. Nos próximos três meses, serão implementados outros protocolos quase-sincronos baseados em vetores de dependência que obedecem à propriedade de Rollback-Dependency Trackability (RDT) [8, 4]. Juntamente com essa etapa, será realizado o estudo dos protocolos síncronos. A princípio, será implementado e visualizado apenas o protocolo proposto por Koo e Toueg (KT) [5].

Concluída a etapa de estudos e a visualização dos protocolos quase-síncronos, será iniciada a implementação e visualização do protocolo KT. Os três últimos meses estão reservados para implementação de melhorias e outras funcionalidades na interface com o usuário e para a implementação de outras aplicações de teste. Por fim, a aplicação resultante será disponibilizada em um *site*, juntamente com o código fonte, seguindo a filosofia de *software* livre.

	2003								2004			
	Mai	Jun	Jul	Ago	Set	Out	Nov	Dez	Jan	Fev	Mar	Abr
1	√	√										
2	√	√										
3		√	√									
4			√	√	√							
5				√	√	√						
6						√	■	■				
7							■	■				
8									■	■		
9										■	■	■
10										■	■	■

√ Etapa conclúida
■ Etapa não concluída

1. Estudo de protocolos quase-síncronos para *checkpointing*
2. Estudo do ambiente para a visualização gráfica
3. Implementação de uma aplicação utilizando o protocolo BCS
4. Especificação da interface entre o monitor e o visualizador
5. Construção do primeiro protótipo
6. Implementação de protocolos quase-síncronos RDT
7. Estudo de protocolo síncrono KT
8. Implementação do protocolo KT
9. Melhorias para a interface de visualização
10. Implementação de outras aplicações de teste

Tabela 1: Cronograma de trabalho

Referências

[1] G. Cao and M. Singhal. Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems. *IEEE Trans. on Parallel and Distributed Systems*, 12(2):157–172, Feb. 2001.

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

[3] 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.

[4] I. C. Garcia, G. M. D. Vieira, and L. E. Buzato. RDT-Partner: An Efficient Checkpointing Protocol that Enforces Rollback-Dependency Trackability. In *Simpósio Brasileiro de Redes de Computadores*, Florianópolis, Santa Catarina, May 2001.

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

[6] D. Manivannan and M. Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):703–713, July 1999.

[7] G. M. D. Vieira. Estudo comparativo de algoritmos para *Checkpointing*. Dissertação de mestrado, Instituto de Computação—Universidade Estadual de Campinas, Dec. 2001.

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

Raphael Marcos Menderico