Ambiente para a Visualização de Protocolos de *Checkpointing*

Relatório Final*- Processo 03/00483-0

Orientando: Raphael Marcos Menderico

Orientadora: Profa. Dra. Islene Calciolari Garcia

Fevereiro de 2004

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 está disponível de acordo com a filosofia de *software* livre.

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

*A entrega do relatório foi antecipada em dois meses, visto que está acompanhada de um pedido de renovação

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 [3]. Um protocolo de *checkpointing* coordena a seleção de *checkpoints* pelos processos de maneira a garantir a construção de *checkpoints* globais consistentes [4, 10].

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 setas horizontais representam a execução dos processos com o decorrer do tempo, as setas diagonais representam as mensagens e os quadrados, *checkpoints*.

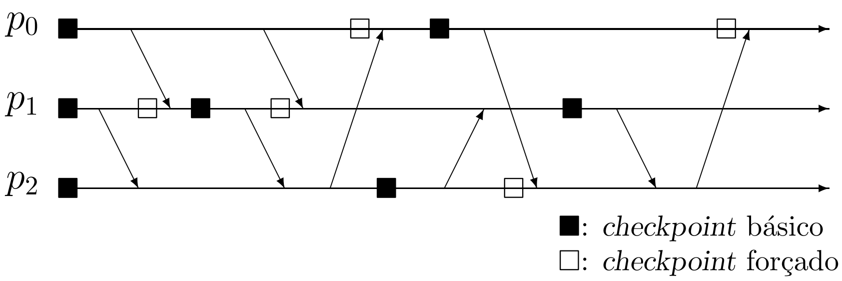

Figura 1: Diagrama espaço-tempo ilustrando um protocolo quase-síncrono

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 [10]. Esta abordagem

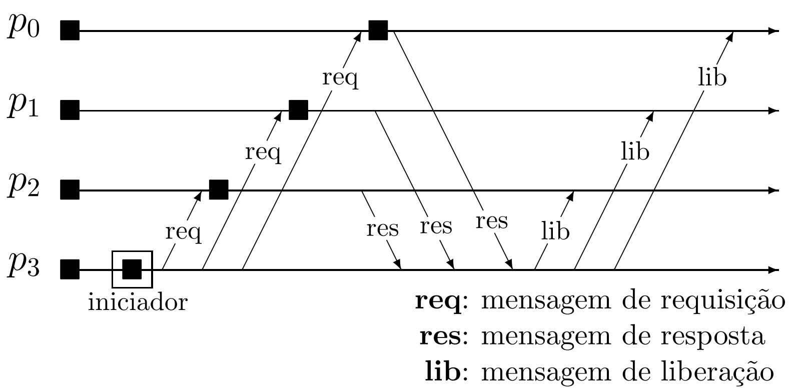

Figura 2: Diagrama espaço-tempo ilustrando um protocolo síncrono bloqueante

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 [14, 10].

A Figura 2 ilustra o comportamento de um protocolo síncrono bloqueante [6]. 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 [2].

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 e 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.

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

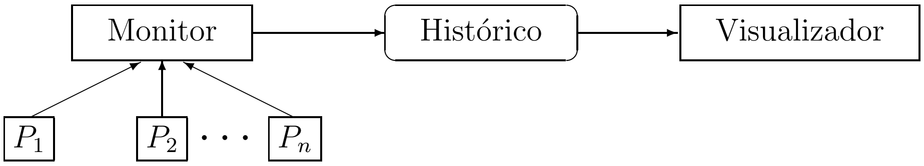

Figura 3: Arquitetura do ambiente

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.

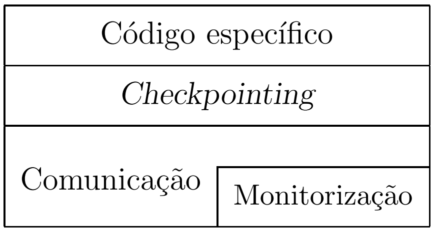

Figura 4: Estrutura de um processo da aplicação

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

Aqui inicia-se a descrição das atividades realizadas e dos resultados obtidos durante o desenvolvimento do projeto. Na Seção 3.1, apresentaremos a estrutura interna de cada processo, descrevendo as camadas de comunicação(Seção 3.1.1), checkpointing (Seção 3.1.2) e de código específico (Seção 3.1.3); a Seção 3.2 detalha o processo monitor; a Seção 3.3 mostra o funcionamento do visualizador e a Seção 3.4 encerra esse tópico apresentando a forma como o conteúdo foi disponibilizado na Internet.

3.1 Implementação dos processos

3.1.1 Camada de comunicação

A classe `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*.

Essa classe 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 [7] 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.

Os processos da aplicação distribuída foram modelados como uma máquina de esstado finita com quatro estados, conforme mostrado na figura 5:

- **Aplicação:** Onde os processos executam o seu código específico e enviam mensagens.

- **Tira checkpointing básico:** Decorrente de uma solicitação da camada de código específico, a camada de checkpointing realiza a gravação do estado do processo e efetua as checagens necessárias. Nesse estado, pode ocorrer o bloqueio do processamento, decorrente dessa checagem. Nesse caso, o processo muda para o estado de bloqueio.

- **Bloqueio:** Estado onde o processo somente analisa as mensagens de controle e coloca todas as mensagens da aplicação em uma fila. A partir desse processamento, a camada de checkpointing decide se o processo continua nesse estado, devolve o controle para a aplicação ou aguarda o recebimento de uma mensagem da aplicação.

- **Aguarda recebimento da mensagem:** Estado em que o processo aguarda a chegada de alguma mensagem em seus canais de entrada ou na fila montada durante o bloqueio, para que a camada de checkpointing analise o conteúdo da mensagem e decida se ela deve ser entregue para a aplicação ou se um bloqueio é gerado. Se for gerado, muda para o estado de bloqueio.

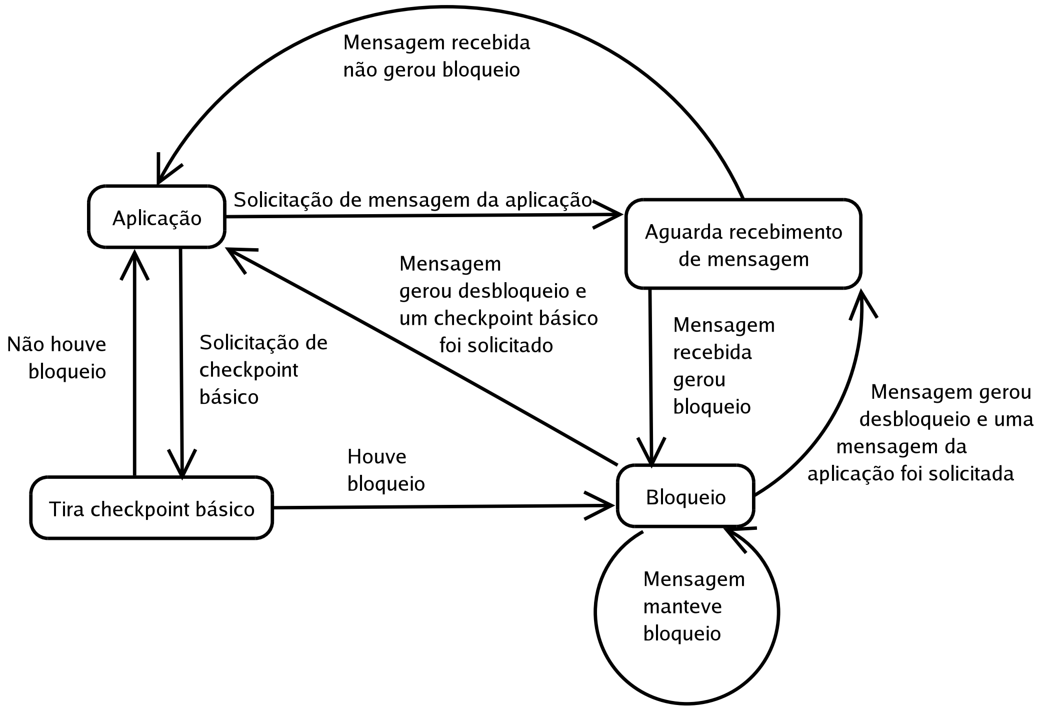

Figura 5: Máquina de estados utilizada para modelar o funcionamento dos processos

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.

3.1.2 Camada de checkpointing

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 utilizado.

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 seguintes protocolos:

- Protocolos Quase-Síncronos ZPF (Z-Path Free)
  - CASBR (Checkpoint-After-Send-Before-Receive) [14, 10]
  - CAS (Checkpoint-After-Send) [14, 10]
  - CBR (Checkpoint-Before-Receive) [14, 10]
  - NRAS (No-Receive-After-Send) [14, 10]
  - FDI (Fixed-Dependency-Interval) [14, 10]
  - FDAS (Fixed-Dependency-After-Send) [14, 10]
  - RDT-Partner [14]
  - BHMR [14]
- Protocolos Quase-Síncronos ZCF (Z-Cycle Free)

- BCS [14, 10]
- BCS Aftersend [14]
- BCS-Partner [14]
- HMNR [14]
- Lazy-BCS [14]
- Lazy-BCS-Aftersend [14]
- Lazy-BCS-Partner [14]
- BQC [14]

• Protocolos Síncronos Bloqueantes

- Koo e Toueg [6]
- Leu e Bhargava [8]

3.1.3 Camada de código específico

A primeira aplicação específica implementada para testar o sistema gerencia um *token* único em um sistema distribuído 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 usam os métodos disponibilizados pela camada de *checkpointing* e fornecem 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. Primeiramente o visualizador realiza 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. Posteriormente, os objetos são desenhados 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.

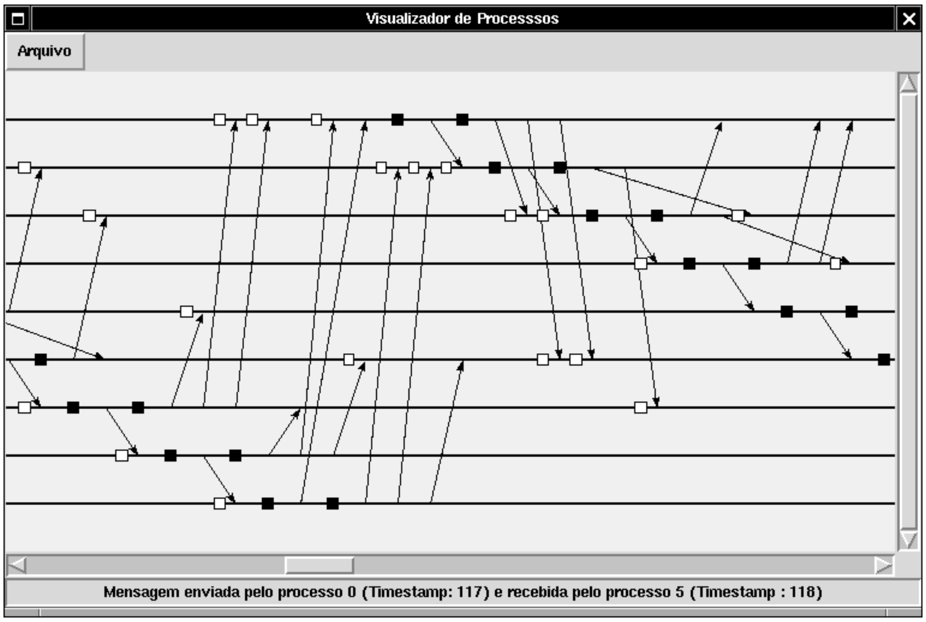

Figura 7: Imagem gerada na execução do protocolo BCS

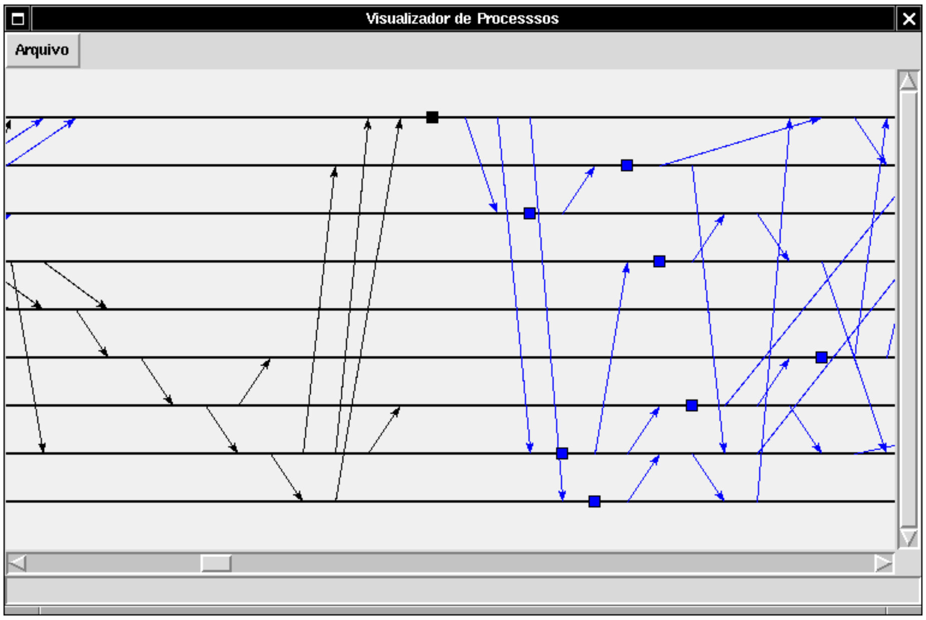

Figura 8: Imagem gerada na execução do protocolo KT

3.4 Disponibilização do conteúdo na Internet

Uma página na Internet foi construída para disponibilizar a aplicação para *download*, de acordo com a filosofia de software livre [1]. No mesmo local encontra-se um conjunto de *screenshots*, um pequeno manual, o código fonte da aplicação e os relatórios enviados a FAPESP. Esse conteúdo será atualizado conforme o projeto for finalizado. O endereço do *site* é `http://www.ic.unicamp.br/~ra009702/iniciacao` e a página inicial é mostrada na Figura 9.

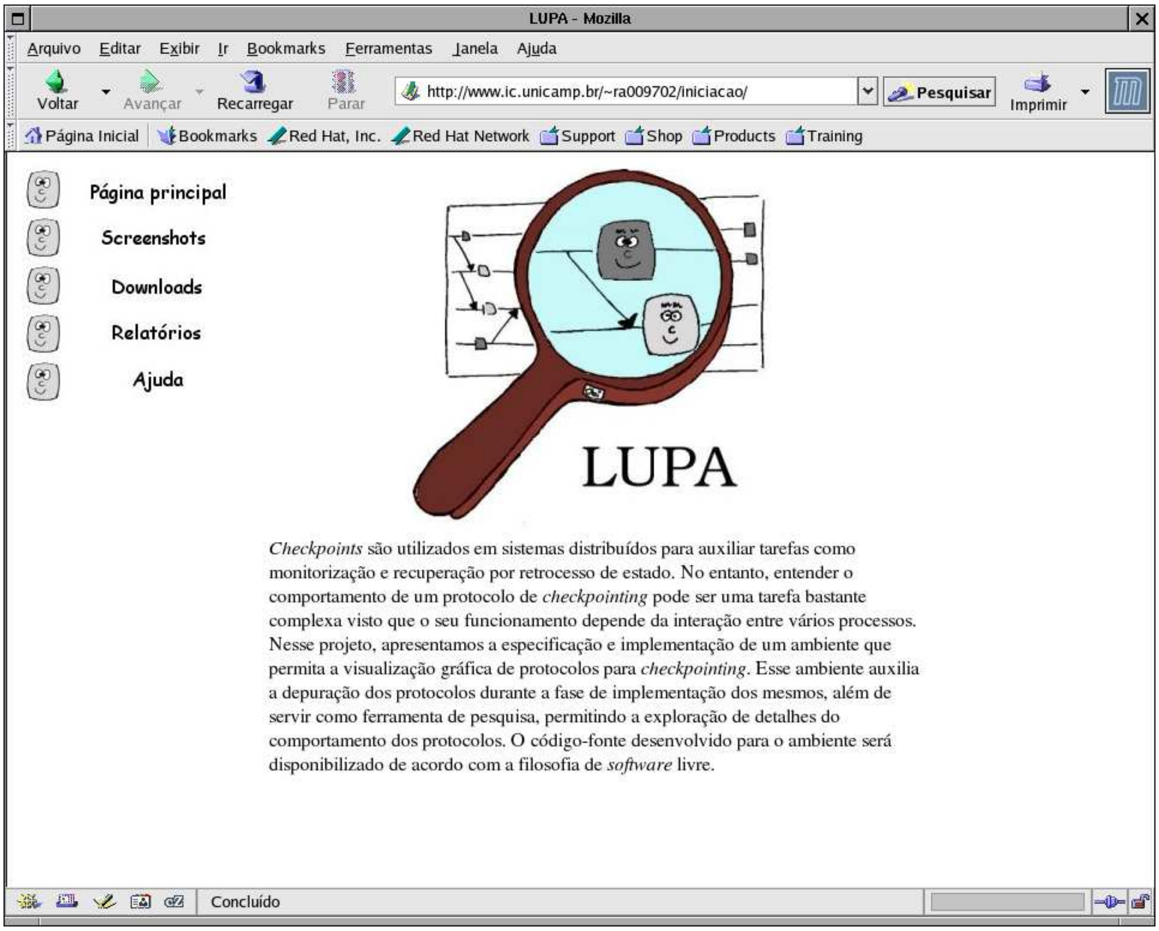

Figura 9: Site utilizado para disponibilizar a aplicação na internet

3.5 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 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 na seçã 3.1.1) que realiza a conversão junto com a camada de comunicação.

Durante a execução das aplicações, foi constatado que um processo qualquer fica bloqueado quando uma mensagem é enviada por ele para outro processo que não está mais executando. 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 [3] para garantir que os canais estejam sem nenhuma mensagem com entrega pendente.

A camada de comunicação foi projetada inicialmente para atender os requisitos dos protocolos de *checkpointing* quase-síncronos. Quando iniciou-se a implementação dos protocolos síncronos, diversas alterações foram realizadas de forma que essa mesma classe suportasse ambas as classes de protocolos. Essas alterações incluem a diferenciação entre mensagens da aplicação e mensagens de controle do protocolo e o bloqueio da aplicação quando for solicitado pela camada superior. A principal dificuldade encontrada durante essa etapa foi projetar essas alterações de forma a atender aos novos requisitos sem que as funcionalidades anteriores fossem perdidas.

Por fim, a interface com o usuário não permitia que seu tamanho fosse alterado pelo usuário. Após pesquisar o funcionamento do ambiente, descobriu-se que pequenas alterações na construção da interface permitiriam essa nova funcionalidade. Embora a solução para esse problema seja conhecida, a interface ainda não foi alterada. Todas as alterações na interface ocorrerão futuramente, conforme descrito na Seção 4.

4 Cronograma e Atividades Futuras

A Tabela 1 mostra o cronograma de trabalho, indicando as tarefas já concluidas e as pendentes. Até o momento, os trabalhos seguiram o cronograma e por isso ele não foi alterado. Em seguida, os protocolos RDT-Minimal [5], VR-Minimal-Simples [13], BCS-SNB [12] e o protocolo síncrono proposto por Prakash e Singhal [11] serão implementados, finalizando o conjunto de protocolos disponíveis na aplicação.

A interface com o usuário será alterada, para tornar seu uso mais simples e amigavel, e para incluir novas funcionalidades, como estatísticas a respeito do número de checkpoints e de mensagens durante a execução de uma determinada aplicação, assim como a construção de linhas indicando cortes globais consistentes (ou linhas de recuperação) para os algoritmos da classe ZPF que possuam vetores de dependências [15] e, também, para os da classe ZCF baseados em relógios lógicos [10]. As estatísticas permitirão realizar comparações entre diferentes algoritmos para uma mesma aplicação e as linhas de recuperação permitirão obter ao menos um estado global consistente para cada *checkpoint* obtido pela aplicação, facilitando a depuração da mesma.

Os algoritmos `AsynchSpanningTree` [9] e `Drinkink Philosophers` [9] serão implementados, para mostrar o funcionamento do ambiente com outras aplicações.

Por fim, segue em anexo a proposta de prorrogação da bolsa, detalhando a implementação de um ambiente de animação de algoritmos baseado no trabalho atual e na construção progressiva de *checkpoints* globais consistentes.

	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] GNU General Public License. Disponível em `http://www.gnu.org/licenses/gpl.html`.

[2] 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.

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

[4] 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.

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

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

[7] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[8] P.-J. Leu and B. Bhargava. Concurrent Robust Checkpointing and Recovery in Distribuited Systems. In *In Proceedings of the 4th IEEE International Conference on Data Engineering*, pages 154–163, 1988.

[9] N. A. Lynch. *Distribuited Algorithms*. Morgan Kaufmann Publishers, Inc, 1996.

[10] 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.

[11] R. Prakash and M. Singhal. Minimal Global Snapshot and Failure Recovery using Infection. Technical Report OSU-CISRC-12/93-TR42, Department of Computer Science, The Ohio State University, 1993.

[12] T. C. Sakata. BCS-SNB: Um Protocolo de Checkpointing Síncrono Não Bloqueante. In *Workshop de Teses e Dissertaçoes (WTC), Latin-American Symposium on Dependable Computing(LADC)*, pages 73–78, São Paulo, SP, Oct. 2003.

[13] T. C. Sakata, I. C. Garcia, and L. E. Buzato. Checkpointing Síncrono Bloqueante Minimal com Iniciadores Concorrentes. In *Simpósio Brasileiro de Redes de Computadores*, pages 681–696, Natal, Rio Grande do Norte, May 2003.

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

[15] 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

Islene Calciolari Garcia