UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

CLAIRTON BULIGON

Implementação de Recuperação por Retorno de Aplicações Distribuídas baseada em *Checkpoints* Coordenados

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Profa. Dra. Ingrid Jansch-Pôrto
Orientadora

Prof. Dr. Sérgio Luis Cechin
Co-orientador

Porto Alegre, junho de 2005.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Buligon, Clairton

Implementação de Recuperação por Retorno de Aplicações Distribuídos baseada em Checkpoints Coordenados / Clairton Buligon – Porto Alegre: Programa de Pós-Graduação em Computação, 2005.

120 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2005. Orientadora: Ingrid Jansch-Pôrto; Co-Orientador: Sérgio Luis Cechin.

1. Tolerância a Falhas. 2. Sistemas Distribuídos. 3. Recuperação por Retorno. 4. Checkpoint. I. Jansch-Pôrto, Ingrid. II. Cechin, Sérgio Luis. III. Título.

AGRADECIMENTOS

Agradeço primeiramente a Deus, por ter me concedido a oportunidade e capacidade para realização deste trabalho.

À professora Dra. Ingrid Jansch-Pôrto e ao professor Dr. Sérgio Luis Cechin pela orientação imprescindível, pela confiança e pela tolerância aos eventuais atrasos no andamento deste trabalho devido à minha falta de tempo.

Aos meus amigos pela compreensão nos meus inúmeros momentos de "corpo presente", com minha concentração sempre voltada para este trabalho, deixando-os falarem sozinhos. Em especial aos amigos Filipe Brandenburger e Valdo Molina, pelas valiosas dicas sobre Linux, me poupando preciosas horas de pesquisa.

À minha família, em especial aos meus pais Vanda Buligon e José Bruno Buligon, pelo sacrifício que sempre fizeram para garantir a educação de seus filhos. Vocês são o meu exemplo de vida e a vocês dedico meus méritos e o mérito deste trabalho.

Por fim, em dobro à minha amada esposa Aline, pelo apoio incondicional para a realização deste trabalho e pela "tolerância a falhas" em relação a mim, abrindo mão dos nossos preciosos momentos em prol da realização deste trabalho. Só você consegue "estabelecer *checkpoints*" dos nossos melhores momentos. A você dedico minha vida! Agradeço também meus sogros Abílio e Albertina pelo apoio e pela sua filha maravilhosa.

Muito obrigado a todos.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 7

LISTA DE FIGURAS 8

LISTA DE TABELAS 9

RESUMO 10

ABSTRACT 11

1 INTRODUÇÃO 12

1.1 Objetivos 16
1.2 Metodologia do Trabalho 16
1.3 Estrutura da Dissertação 17

2 CONCEITOS E ABORDAGENS 19

2.1 Modelo de Sistema Distribuído 19
2.2 Algoritmo Utilizado 22
2.3 Infra-estrutura Disponível 25
2.3.1 Ambientes simulados 25
2.3.2 Ambientes reais 26
2.4 Alternativa Adotada 27
2.5 Trabalhos Relacionados 28

3 O MODELO DE IMPLEMENTAÇÃO DO SISTEMA 30

3.1 Ambiente de Implementação 30
3.1.1 Modelo de processos 31
3.1.2 Comunicação entre processos 32
3.1.3 *System calls* 33
3.1.4 Espaço de *kernel* e espaço de usuário 34
3.2 Modelo Computacional 34
3.2.1 Aplicação distribuída 34
3.2.2 O sistema de *checkpointing* 35
3.3 Funcionamento do Sistema de *Checkpointing* 37
3.3.1 Situações para estabelecimento de *checkpoints* 38
3.4 Arquitetura do Sistema de *Checkpointing* 38
3.4.1 Demonstração das situações 41
3.5 Seqüência de Execução 42

3.6 Ciclo de Vida do Sistema 45

4 IMPLEMENTAÇÃO 49

4.1 Estrutura Geral 49
4.2 Módulo `CKPT` 50
4.2.1 Biblioteca `CRAK` 51
4.2.2 Interceptação 55
4.2.3 Interface de acesso 59
4.2.4 Recuperação de *sockets UDP* 60
4.3 Processo `CK` 61
4.4 Processo `MONITOR` 63
4.5 Mensagens 66
4.6 Operações de Acordo 67
4.6.1 Estabelecimento de linhas de recuperação consistentes 68
4.6.2 Rotação de coordenadores 70
4.6.3 Detecção de defeitos e procedimentos relacionados 72
4.6.4 Recuperação 74
4.7 Da Descrição Formal para o Código-fonte 76
4.8 Limitações e Aspectos Pendentes 79

5 ANÁLISE DE RESULTADOS 81

5.1 Métricas e Variáveis 81
5.2 Aplicações 82
5.3 Definição das Execuções e da Coleta de Dados 84
5.4 Avaliação dos Resultados 86
5.4.1 Fator de interceptação 86
5.4.2 Fator de *checkpoint* 87
5.4.3 Fator de mensagens geradas 88
5.4.4 Sobrecarga do sistema de *checkpointing* 90
5.5 Tempo de recuperação 91

6 CONCLUSÃO E TRABALHOS FUTUROS 93

REFERÊNCIAS 96

ANEXO A DESCRIÇÃO DA ESPECIFICAÇÃO DO ALGORITMO 100

ANEXO B CONSIDERAÇÕES DO AUTOR DA BIBLIOTECA `CRAK` 105

APÊNDICE A CONSTANTES 106

APÊNDICE B ESTRUTURAS DE DADOS E VARIÁVEIS 108

APÊNDICE C FUNÇÕES DO MÓDULO `CKPT` 110

APÊNDICE D FUNÇÕES DO PROCESSO `CK` 111

APÊNDICE E CONSUMO DE RECURSOS DE REDE 113

APÊNDICE F CÁLCULO DO Nº DE REPETIÇÕES NECESSÁRIAS 115

APÊNDICE G RESULTADOS DAS SIMULAÇÕES REALIZADAS 117

APÊNDICE H MENSAGENS SALVAS E TEMPO POR *CHECKPOINT* 118

APÊNDICE I MENSAGENS INTERCEPTADAS 119

APÊNDICE J CONTEÚDO DO CD-ROM 120

LISTA DE ABREVIATURAS E SIGLAS

API	Application Program Interface
BSD	Berkeley Software Distribution
CCM	CORBA Component Model
CPU	Central Processing Unit
CRAK	Checkpoint/Restart As a Kernel Module
DCOM	Distributed Component Object Model
EJB	Entreprise Java Beans
FIFO	First-In First-Out
FTP	File Transfer Protocol
GUID	Group Unique Identifier
IDE	Integrated Disk Electronics
IP	Internet Protocol
IPv4	Internet Protocol versão 4
MPI	Message Passing Interface
NS	Network Simulator
PID	Process Identifier
PVM	Parallel Virtual Machine
PWD	Piecewise Deterministic
SSD	Solid State Disks
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol / Internet Protocol
TLA	Temporal Logic of Actions
UDP	User Datagram Protocol
UID	Unique Identifier
VMA	Virtual Memory Area
WWW	World Wide Web

LISTA DE FIGURAS

Figura 2.1: Modelo de computação distribuída 22
Figura 3.1: Modelo de aplicação distribuída 35
Figura 3.2: Processo da aplicação distribuída em ambiente sem *checkpointing*. 35
Figura 3.3: Processo da aplicação distribuída em ambiente com *checkpointing*..... 37
Figura 3.4: Arquitetura do sistema de *checkpointing* em um nodo do sistema. 39
Figura 3.5: Passos para o usuário executar as aplicações. 43
Figura 3.6: Ciclo de vida do sistema de *checkpointing*. 46
Figura 4.1: Distribuição dos componentes do sistema. 50
Figura 4.2: Função da biblioteca para salvamento de estados dos processos..... 51
Figura 4.3: Função da biblioteca para recuperação processos..... 54
Figura 4.4: Interceptação de *system calls*. 56
Figura 4.5: Alterações na tabela *sys_call_table*. 57
Figura 4.6: Funções de interceptação 58
Figura 4.7: Interface do processo **`MONITOR`**..... 64
Figura 5.1: Operações da aplicação sintética..... 83
Figura 5.2: Comparação no tamanho dos *checkpoints*. 87
Figura 5.3: Sobrecarga do sistema de *checkpointing* 90

LISTA DE TABELAS

Tabela 5.1: Perfil das aplicações 83
Tabela 5.2: Sobrecarga imposta pela interceptação..... 86
Tabela 5.3: Tempo gasto no estabelecimento dos *checkpoints* 87
Tabela 5.4: Taxa de mensagens salvas em cada *checkpoint*..... 88
Tabela 5.5: Número de mensagens geradas..... 89
Tabela 5.6: Sobrecarga das mensagens de controle 89
Tabela 5.7: Sobrecarga do sistema de *checkpointing* 90

RESUMO

A recuperação por retorno baseada em *checkpointing* é largamente usada como técnica de tolerância a falhas. O modelo complexo de sistemas distribuídos tem motivado o desenvolvimento de diversos algoritmos na tentativa de encontrar soluções mais simples e eficientes. Os processos que formam o sistema distribuído podem coordenar suas operações para garantir que o conjunto de *checkpoints* locais componha um estado global consistente (linha de recuperação). A partir desse estado, no caso de ocorrência de falhas, o sistema pode ser recuperado e a computação retomada a partir de um momento anterior ao da manifestação da falha, evitando o retrocesso para o estado inicial da computação e prevenindo a ocorrência de prejuízos com a perda de todo processamento até então realizado.

No Grupo de Tolerância a Falhas da UFRGS foi proposto recentemente um algoritmo que é voltado para aplicações que executam em sistemas distribuídos assíncronos que se comunicam exclusivamente pela troca de mensagens. Ele opera com salvamento coordenado de *checkpoints* (não bloqueando as aplicações) e prevê o tratamento de mensagens órfãs e perdidas. Os mecanismos do algoritmo sugerem que nenhuma alteração deveria ser realizada no código das aplicações, criando a possibilidade de implementação transparente sob o ponto de vista dos usuários e dos programadores das aplicações. Como o algoritmo não requer o bloqueio das aplicações, a sobrecarga imposta pelos mecanismos à execução livre de falhas é pequena. Além disso, o processo de recuperação tende a ser efetuado rapidamente, uma vez que é garantida a existência de uma linha de recuperação consistente, facilmente identificada.

Este trabalho apresenta as decisões de projeto, a implementação, os resultados e a avaliação de desempenho desse algoritmo. A avaliação das alternativas de implementação resultou na decisão de uma implementação então realizada diretamente sobre o sistema operacional Linux, sem recorrer a protocolos auxiliares para garantir a execução dos serviços e sem a necessidade de adaptações no código das aplicações nem no código do sistema operacional. Adicionalmente, os resultados comprovaram a expectativa inicial de que o algoritmo causaria pouca sobrecarga no sistema (menos de 2%), embora ele ainda apresente alta dependência do tamanho dos *checkpoints* salvos.

Palavras-Chave: Tolerância a falhas, sistemas distribuídos, recuperação por retorno, *checkpoint*.

Implementing Rollback-Recovery Coordinated Checkpoints

ABSTRACT

The rollback-recovery technique based on previous checkpoints is largely used for fault tolerance. The complexity of distributed system models has motivated the development of new algorithms that offer simpler, more efficient and more transparent solutions. The processes in the distributed system may coordinate their actions to ensure that the set of local checkpoints form a consistent global state (recovery line). From that state, in the case of failures, the system can be rolled back to this recovery line and the computation can be restarted from this consistent state, which avoids the return to the initial state of the computation and prevents the loss of previous processing so far.

In our Fault Tolerance Group, rollback-recovery algorithm was proposed, which focuses on asynchronous distributed system, based on message passing. It operates with coordinated non-blocking checkpointing and ensures the treatment of orphan and lost messages. The mechanisms of the algorithm suggest that no changes should be made over the application code, which leads to an implementation that is transparent from the point of view of users and applications programmers. As it does not block applications, there is little overhead during failure-free operation. Furthermore, the rollback procedure tends to be quick, as a recovery line always exists and can be easily identified.

This work discusses the implementation of this algorithm, including the project decisions, the performance evaluation and the test results. The evaluation of the implementation options led to the choice of a Linux based implementation, directly on the kernel, with no support given by specialized protocols, no need to patch the kernel source. User transparency was also accomplished, as no change in the application source code is needed. Our experiments with the implemented prototype showed that the algorithm really has a very low performance overhead (less than 2%), but it is still highly dependent on the size of stored checkpoints.

Keywords: Fault tolerance, distributed systems, rollback-recovery, checkpoint.

1 INTRODUÇÃO

Os recentes avanços tecnológicos têm proporcionado um aumento considerável na capacidade de processamento dos sistemas. O desenvolvimento de poderosos microprocessadores e sistemas de comunicação de alta velocidade, associado às vantagens econômicas no emprego de vários elementos de processamento de maneira ótima para se conseguir maior poder computacional, tem incentivado a utilização de sistemas distribuídos em larga escala (ÖZSU; VALDUREZ, 2001). Não é de hoje que numerosas pesquisas são direcionadas em busca de novas tecnologias para explorar os limites computacionais dos recursos desses sistemas, que já não são tão raros como outrora, quando o processamento em larga escala limitava-se ao uso de espaçosos e caros supercomputadores, cujo acesso era privilégio de poucos.

Sistemas distribuídos são atualmente explorados por inúmeras aplicações, incluindo sistemas cliente-servidor, processamento de transações, computação científica, sistemas agregados (*clusters*), computação em grade (g*rids*), entre tantas outras. Entretanto, o conseqüente aumento na complexidade desses sistemas tornou-os cada vez mais suscetíveis à ocorrência de falhas, o que tem motivado o desenvolvimento de diferentes técnicas para melhorar as características de confiabilidade e disponibilidade oferecidas às aplicações que exploram sistemas distribuídos.

Exemplos dessas técnicas são: transações, comunicação em grupo e recuperação por retorno (ELNOZAHY *et al*., 2002). A primeira técnica é direcionada para aplicações orientadas a dados, como banco de dados e sistemas de arquivos. A segunda propõe uma abstração de um sistema de comunicação ideal, sobre o qual os programadores podem assumir confiabilidade em suas aplicações. A terceira técnica, explorada no presente trabalho, está focada em aplicações que executam por um longo período de tempo, como computação científica, e aplicações voltadas para área de telecomunicações. Essa técnica tem por objetivo a redução da perda de processamento devido à ocorrência de falhas.

Várias definições de sistemas distribuídos têm sido apresentadas na literatura (TANENBAUM, 2002). Para os propósitos do presente trabalho, consideram-se sistemas distribuídos como uma coleção de processos que se comunicam exclusivamente através da troca de mensagens enviadas através de uma rede. Durante a execução livre de falhas, os processos efetuam periodicamente o armazenamento de pontos de recuperação (*checkpointing*) em uma memória que sobrevive às falhas previstas no modelo, denominada memória estável. Estes pontos de recuperação, denominados de *checkpoints*, contêm informações sobre os processos e sobre o canal de comunicação, necessárias para, em caso de ocorrência de falhas, garantirem a retomada do processamento. Os pontos de recuperação correspondem a estados intermediários, obtidos antes da manifestação da falha e posteriormente ao início da execução, reduzindo assim a quantidade de computação perdida.

As informações de recuperação incluem, no mínimo, o estado dos processos participantes. Diferentes aplicações ou protocolos de recuperação podem requerer informações adicionais, tais como o estado dos periféricos de entrada e saída, eventos que ocorram durante o processamento ou mensagens trocadas entre os processos e o meio externo. Sistemas concorrentes baseados em troca de mensagens apresentam desafios à recuperação por retorno porque as mensagens induzem dependências entre os processos durante a execução livre de falhas. Em caso de falhas, processos que não falharam podem ser forçados a retroceder a computação, causando a propagação dos efeitos do retorno da computação. Em alguns cenários, esta propagação pode provocar inclusive o retorno para o estado inicial da computação, causando o chamado "efeito dominó" (RANDELL, 1975), que deve ser evitado.

O uso de recuperação por retorno para garantir tolerância a falhas em sistemas distribuídos é uma técnica bastante pesquisada. Segundo Elnozahy *et al.* (2002), estudos recentes têm desafiado algumas premissas sobre as quais muitos protocolos de recuperação por retorno se baseiam. Os primeiros protocolos de recuperação por retorno foram propostos na década de 80, quando a velocidade de processamento e a vazão dos meios de comunicação implicavam em alta sobrecarga imposta por esses protocolos, principalmente quando comparados com o custo de acesso à memória estável (BHARGAVA; LIAN; LEU, 1990). As mensagens de controle necessárias para sincronização entre os múltiplos nodos do sistema também acarretavam uma sobrecarga considerável ao sistema. A sobrecarga gerada pelas mensagens de controle dos protocolos e pelo tempo gasto para o salvamento dos *checkpoints* prejudicou a aceitação dessa técnica de tolerância a falhas por sistemas comerciais durante muitos anos (ELNOZAHY *et al*., 2002). Entretanto, recentemente a velocidade dos processadores, a vazão dos canais de comunicação e a velocidade de acesso à memória estável aumentaram consideravelmente, possibilitando que a recuperação por retorno se tornasse uma boa alternativa para implementação de tolerância a falhas em sistemas distribuídos.

Ainda segundo Elnozahy *et al.* (2002), a recuperação por retorno possui muitos enfoques quanto à transparência desta técnica, sob o ponto de vista da aplicação ou dos programadores do sistema. O sistema de *checkpointing* pode contar com a aplicação para decidir "quando" e "o quê" salvar na memória estável, transferindo toda a responsabilidade para o programador da aplicação; a transparência pode limitar-se a algumas características de uso, provendo apenas alguns mecanismos que auxiliam a estruturar a aplicação; ou ser mais abrangente, não requerendo qualquer intervenção por parte da aplicação ou do programador. Neste caso, o sistema automaticamente obtém os pontos de recuperação de acordo com alguma política pré-estabelecida e os utiliza para recuperar a aplicação se uma falha ocorrer. A vantagem desta abordagem é permitir que os programadores se concentrem plenamente sobre suas atividades-fim, sendo liberados de tarefas complexas que podem levar a erros na implementação dos mecanismos de tolerância a falhas nas aplicações. Além disso, é garantida a inserção de tolerância a falhas em aplicações convencionais, de maneira plenamente transparente.

Elnozahy classifica a implementação de recuperação por retorno em sistemas distribuídos em duas categorias: recuperação baseada em *checkpoint* e recuperação baseada em *log*. Nos *logs* são armazenadas todas as operações que causam modificações sobre o sistema, com a perspectiva de poder desfazer ou refazer estas operações quando for necessário. A primeira baseia-se exclusivamente nos *checkpoints* para obter tolerância a falhas e será a abordagem adotada no presente trabalho.

A recuperação por retorno distribuída baseada em *log* necessita que os processos tenham um comportamento tipo *PWD* (efetuam um processamento determinístico iniciado com eventos não determinísticos) e é uma alternativa para aplicações que interagem com o mundo externo. Ela combina *checkpointing* com o salvamento de eventos não determinísticos, normalmente associados às mensagens geradas internamente pelo sistema e mensagens trocadas com o mundo externo. Essa categoria de recuperação por retorno classifica-se em três tipos: pessimista, otimista e causal. Em *log* pessimista, a aplicação é bloqueada até que todos os eventos sejam salvos em memória estável e antes que seus efeitos possam ser percebidos pelos demais processos ou pelo mundo externo. Em *log* otimista, a aplicação não é bloqueada e os eventos são temporariamente armazenados em memória volátil. Posteriormente o conteúdo da memória volátil é salvo em memória estável. *Log* causal é uma combinação dos dois anteriores. As três classes também diferem no que diz respeito à implementação de coleta de lixo e na interação com o mundo externo.

A recuperação por retorno baseada em *checkpoints* implementa tolerância a falhas exclusivamente através do salvamento periódico do estado da aplicação. Periodicamente, são estabelecidos *checkpoints* locais contendo informações sobre o estado da aplicação, necessárias para executar sua recuperação em caso de falha. Quando uma aplicação é distribuída, o **estado global consistente** é aquele que foi estabelecido em execução livre de falhas e sem perda mensagens enviadas de um processo para outro. Um ***checkpoint* global consistente** (também chamado de linha de recuperação) é formado pelo conjunto de todos os *checkpoints* locais que constituem o estado global consistente da aplicação, e sua existência é requisito para reiniciar a execução da aplicação, após a ocorrência de falhas.

Em relação ao estabelecimento das linhas de recuperação, os protocolos de recuperação baseados exclusivamente em *checkpointing* são subdivididos em três categorias: **não coordenados, coordenados e controlados pela comunicação** (ELNOZAHY *et al*., 2002). Em seguida, esses protocolos serão brevemente descritos.

Nos protocolos não coordenados os processos participantes estabelecem seus *checkpoints* locais individualmente. Na recuperação, esses *checkpoints* são processados até formarem um *checkpoint* global consistente, o qual é utilizado para a recuperação da aplicação. A vantagem dessa abordagem é permitir que os processos estabeleçam seus *checkpoints* de acordo com sua conveniência. A contrapartida é que, durante a recuperação, as dependências entre os *checkpoints* podem causar o "efeito dominó", obrigando a aplicação a retroceder para seu estado inicial. Além disso, cada um dos processos deve manter múltiplos *checkpoints*, resultando em sobrecarga na utilização da memória estável. Por último, os processos podem estabelecer *checkpoints* inúteis, que não contribuem para formar um *checkpoint* global consistente.

Nos protocolos coordenados, o *checkpointing* é conduzido de tal maneira que o conjunto de *checkpoints* individuais sempre resulta em um *checkpoint* global consistente. Tipicamente os processos usam mensagens de controle adicionais para sincronizar as atividades de *checkpointing*. Isso simplifica a utilização da memória estável, uma vez que é necessário manter um único *checkpoint*, removendo aqueles que fazem parte de *checkpoints* globais obsoletos. Adicionalmente, protocolos coordenados estão livres do "efeito dominó", pois sempre existirá um *checkpoint* global consistente para recuperar a aplicação em caso de falhas. Esses protocolos podem ser ainda classificados como **bloqueantes** e **não bloqueantes**.

Protocolos coordenados bloqueantes são tipicamente protocolos de múltiplas fases. Na primeira fase é assegurado o "congelamento" dos canais de comunicação entre os processos da aplicação. Uma vez que os canais de comunicação estão vazios, pode-se efetuar o estabelecimento dos *checkpoints*: a consistência estará trivialmente garantida, pois não existem mensagens caracterizadas como "em trânsito". Em seguida, os canais de comunicação podem ser liberados. A principal desvantagem de protocolos bloqueantes é a latência imposta pelo estabelecimento dos *checkpoints*, o que causa uma sobrecarga considerável durante execuções livres de falhas.

No caso dos protocolos não bloqueantes, não é necessário "congelar" os canais de comunicação entre os processos, durante os procedimentos de *checkpointing*. Entretanto, isso implica na utilização de mecanismos adicionais para prevenir que mensagens "em trânsito" possam tornar o *checkpoint* global inconsistente. Se os canais forem confiáveis e assegurarem ordem FIFO, esse problema pode ser evitado, bastando que as mensagens de coordenação sejam todas processadas antes de qualquer mensagem da aplicação, fazendo com que cada processo estabeleça um *checkpoint* antes de processar qualquer mensagem da aplicação enviada após as mensagens de coordenação. Por outro lado, se os canais não assegurarem ordem FIFO, marcadores ou índices podem ser inseridos nas mensagens de maneira a dirigirem os receptores a estabelecerem seus *checkpoints*, quando necessário.

Por último, protocolos controlados pela comunicação evitam o "efeito dominó". Para isso, permitem que os processos estabeleçam seus *checkpoints* independentemente, gerando *checkpoints* locais, e *checkpoints* induzidos, denominados *checkpoints* forçados. Neste tipo de protocolo, informações de controle do mesmo são anexadas em todas as mensagens enviadas pela aplicação. O receptor de cada mensagem da aplicação utiliza essas informações para determinar se deve ou não estabelecer um *checkpoint* forçado visando formar um *checkpoint* global consistente. O *checkpoint* forçado deve ser estabelecido antes que a aplicação processe a mensagem recebida, o que pode incorrer, sob o ponto de vista da aplicação, em aumento na latência da transmissão de uma mensagem. Ao contrário dos protocolos coordenados, não são utilizadas mensagens especiais de coordenação.

Diferentes algoritmos foram propostos na literatura (ELNOZAHY *et al*., 2002) para *checkpointing* e recuperação de aplicações em ambientes distribuídos, mas pouco se conhece a respeito de implementações destas propostas, principalmente em se tratando de implementações comerciais. Os poucos estudos experimentais disponíveis na literatura, que serão abordados posteriormente na Seção 2.5, têm mostrado que a programação de protocolos de recuperação com pouca sobrecarga em execuções livres de falhas são facilmente praticáveis, principalmente com os recursos computacionais disponíveis atualmente. Ao mesmo tempo, os estudos mostram que a principal dificuldade na implementação desses protocolos recai na complexidade em manipular a recuperação propriamente dita (ELNOZAHY; ZWAENEPOEL, 1992). A complexidade associada à resolução de problemas de consistência e garantia de transparência para aplicações que incorporam recuperação de processos são bons exemplos de pontos em aberto que ainda merecem serem discutidos.

Em vista desta complexidade, é comum encontrar-se implementações restritas a ambientes fechados, baseados em infra-estrutura especializada, como em ambientes MPI (*Message Passing Interface*), sistemas agregados (*clusters*), ou nos recentes ambientes em grade (g*rids*), onde existe infra-estrutura física (*hardware*) e lógica (*software*) coordenando o processamento e garantindo qualidade nos serviços. Em ambientes distribuídos convencionais, onde não se dispõe desta infra-estrutura, trabalhos de

implementação de *checkpointing* geralmente concentram-se no desenvolvimento de bibliotecas e conjuntos de primitivas que implementam mecanismos de tolerância a falhas. Há, portanto, carência de implementações completas de recuperação sobre as quais possam ser executadas aplicações distribuídas.

1.1 Objetivos

O objetivo central do presente trabalho é implementar e avaliar o desempenho de um algoritmo de recuperação por retorno em sistemas distribuídos, levando em consideração aspectos de transparência e inexistência de infra-estrutura especializada. Visa-se a concepção de um **sistema de *checkpointing*** autônomo, que possa ser facilmente executado no sistema operacional como serviço complementar, possibilitando aos usuários executarem as aplicações distribuídas em conjunto com esse sistema, de modo a torná-las tolerantes a falhas sem necessidade de incorporar qualquer adaptação em seus códigos.

O trabalho propõe abrangência suficiente para explorar aspectos conceituais envolvendo recuperação por retorno em sistemas distribuídos até aspectos de usabilidade, relacionados a mecanismos empregados na recuperação propriamente dita das aplicações. Dessa forma, além da implementação de *checkpointing*, deverá ser possível proceder à recuperação de acordo com premissas estabelecidas. O objetivo desta tarefa é mostrar que, através do uso adequado dos recursos disponíveis e a implementação dos mecanismos propostos pelo algoritmo adotado, é possível obter-se uma recuperação facilmente praticável e de baixo custo, mesmo sem a utilização de infra-estrutura especializada.

Adicionalmente, visa-se obter subsídios suficientes para complementar a avaliação do algoritmo desenvolvido dentro do contexto do Grupo de Tolerância a Falhas da UFRGS, buscando enriquecer os trabalhos de pesquisa efetuados pelo Grupo com um trabalho voltado exclusivamente para a experimentação de um algoritmo resultante de uma proposta que gerou uma tese de doutorado. O algoritmo de recuperação escolhido, desenvolvido por Cechin (2002), opera por retorno, é do tipo coordenado e explora técnicas de protocolos controlados pela comunicação, é não determinístico[1] e não bloqueia a aplicação.

1.2 Metodologia do Trabalho

Partiu-se da investigação de trabalhos relacionados à implementação de *checkpointing* e recuperação de processos em ambientes **não** distribuídos, tais como Plank (1995), Pinheiro (1999), Zhong (2001), Zandy (2003) e Sudakov (2003); e de trabalhos relacionados desenvolvidos dentro do contexto do Grupo de Tolerância a Falhas da UFRGS, como os trabalhos de Fontoura (2002), Silva (2002) e Cechin (2002), com a expectativa de reunir informações suficientes para identificar a possibilidade de implementação de um sistema de *checkpointing* para aplicações distribuídas em um sistema operacional, preferencialmente em um sistema operacional comercialmente utilizado nos dias atuais.

---

[1] Quando o sistema retornar o estado da aplicação para um estado previamente armazenado e retomar o processamento, este pode não ser igual àquele que levou à falha.

A investigação desses trabalhos relacionados visou explorar as técnicas e soluções empregadas para os problemas decorrentes da implementação, levando em conta os objetivos propostos.

A partir da análise da especificação do algoritmo empregado, foram identificados os mecanismos necessários, convencionais ou não, a serem implementados. A viabilidade de implementação desses mecanismos foi observada a partir da investigação inicial e testes dos diferentes ambientes e dos trabalhos relacionados, se necessário com programação e validação de cada mecanismo para assegurar que seriam praticáveis. A partir dessa análise, foi possível delimitar o nível de compatibilidade do sistema a ser implementado com as aplicações distribuídas, bem como o nível de portabilidade obtido.

Uma vez certificada a viabilidade de implementação de cada mecanismo, no ambiente mais adequado, foi construído um modelo conceitual que representasse o sistema como um todo (sistema de *checkpointing*, ambiente de execução e aplicações) buscando organizar as atividades posteriores de programação.

A programação envolveu a adaptação de bibliotecas e programação dos componentes complementares que, conectados, formaram o sistema como um todo. Ao final da programação, testes foram realizados para depuração do sistema.

Com a conclusão do sistema de *checkpointing*, foram definidos diferentes cenários de execução das aplicações, buscando exercitar os diferentes comportamentos de aplicações reais. Um exemplo destes comportamentos pode ser expressado por cenários em que as aplicações consomem grande quantidade de recursos de processamento, de rede ou de memória. A partir da definição desses cenários, foi definido um conjunto adequado de seqüências de execução para cada cenário, buscando obter resultados quantitativos e qualitativos, usados como subsídios para avaliar o desempenho do algoritmo e, sobretudo, da própria implementação.

1.3 Estrutura da Dissertação

A presente dissertação está organizada conforme segue.

O Capítulo 2 apresenta os aspectos conceituais das abordagens encontradas na literatura, relacionados ao modelo de ambiente distribuído adotado e ao ambiente de implementação, juntamente com seu suporte para a construção dos mecanismos que fazem parte do sistema a ser desenvolvido. Primeiramente, serão apresentados o modelo de sistema distribuído e definições dos conceitos de tolerância a falhas envolvidas nos trabalhos de implementação. Após a apresentação do algoritmo empregado, serão apresentados os resultados de uma investigação sobre a disponibilidade de infra-estrutura para implementação e a alternativa adotada como base dos trabalhos de programação. Por fim, serão apresentados alguns trabalhos relacionados investigados na literatura.

O capítulo 3 aborda as principais conseqüências das estratégias de implementação adotadas. O objetivo deste capítulo é estabelecer os conceitos fundamentais sobre o funcionamento esperado do sistema implementado, a ser usada no capítulo referente à implementação.

O capítulo 4 descreve o desenvolvimento do sistema de *checkpointing*. Durante a apresentação das soluções desenvolvidas, serão feitas referências aos correspondentes mecanismos que compõem o algoritmo. Procurou-se também estabelecer uma correspondência entre a implementação e a especificação do algoritmo, buscando reforçar

a relação direta entre elas. Por fim, os problemas enfrentados nos trabalhos de implementação são relacionados e discutidos.

O capítulo 5 apresenta os resultados das medições dos experimentos efetuados com o sistema desenvolvido. A análise desses resultados e as medidas de desempenho também são apresentadas e discutidas.

Encerrando o trabalho, o capítulo 6 apresenta as conclusões extraídas através da implementação e das medidas de desempenho obtidas. Também são apresentadas sugestões de trabalhos futuros que podem dar seguimento ou aperfeiçoamento ao presente trabalho.

Em anexo, é disponibilizado no Apêndice J um CD-ROM contendo o código-fonte completo do presente trabalho e instruções complementares de uso do sistema de *checkpointing* para auxiliar futuras avaliações ou mesmo o uso deste como ponto de partida para trabalhos relacionados.

2 CONCEITOS E ABORDAGENS

Uma vez que o foco do presente trabalho é implementar um sistema de *checkpointing* em ambiente distribuído, é necessário definir precisamente o modelo que será adotado, o qual será referenciado ao longo do texto. Da mesma forma, é necessário identificar o ambiente de implementação e o suporte disponível para a construção dos mecanismos que fazem parte do sistema.

Primeiramente, serão apresentados neste capítulo o modelo de sistema distribuído e definições dos conceitos de tolerância a falhas envolvidos na implementação do sistema de *checkpointing*, tais como modelo de processos, comunicação entre os processos, falhas toleradas, memória estável e detecção de defeitos. Em seguida, o algoritmo a ser implementado será apresentado, suficientemente detalhado para permitir a identificação dos mecanismos a serem implementados. Estes, por sua vez, também serão analisados quanto a sua viabilidade de implementação nas plataformas pesquisadas. Após a apresentação da plataforma adotada, os trabalhos de implementação mais significativos serão analisados, buscando identificar as diferentes abordagens de cada autor.

2.1 Modelo de Sistema Distribuído

Segundo Jalote (1994), existem duas maneiras de visualizar um sistema distribuído: a partir dos componentes físicos e a partir do processamento, ou da computação. Os dois pontos de vista equivalem, respectivamente, ao modelo físico e ao modelo lógico.

O modelo lógico representa a computação realizada, sob a perspectiva do usuário, onde os serviços distribuídos executam formando um único sistema. O modelo físico envolve os processadores, ou nodos, sobre os quais os serviços são executados, e o meio de comunicação entre esses nodos. Um dos objetivos de tolerância a falhas em sistemas distribuídos é garantir a disponibilidade do sistema, definido no modelo lógico, nos casos de ocorrência de falhas nos componentes do modelo físico.

Para o presente trabalho, considera-se o modelo de sistema distribuído aquele formado pelo modelo físico e lógico. Entretanto, o trabalho será focado exclusivamente nos serviços do modelo lógico, sobre os quais serão implementados mecanismos que buscam garantir tolerância a falhas na execução de aplicações distribuídas. Os componentes do modelo físico corresponderão aos computadores e aos equipamentos de interligação desses computadores. Já os componentes do modelo lógico corresponderão ao sistema operacional instalado nesses computadores e aos serviços disponíveis no sistema operacional que possibilitam a distribuição do processamento.

Baseado nas definições acima e obedecendo à definição adotada pelo autor do algoritmo utilizado (CECHIN, 2002), o modelo de sistema distribuído será formado por um conjunto de processadores, interligados através de uma rede de comunicação, que

efetua um processamento distribuído. Os processadores equivalem aos computadores, ou nodos, onde a computação é realizada pelos processos da aplicação distribuída. Além disso, para efeitos de simplificação do modelo, cada processador executará um único processo da aplicação.

Esses processos, distribuídos entre os computadores, efetuam alguma computação e comunicam-se exclusivamente pela troca de mensagens através da rede de comunicação. Dessa forma, não haverá nenhuma outra forma de comunicação entre os processos, tais como memória compartilhada, relógios sincronizados, etc. As velocidades relativas dos processadores, tanto no que diz respeito à velocidade de computação quanto à velocidade de processamento das mensagens enviadas e recebidas, não são limitadas e não são necessariamente iguais entre si. Da mesma forma, a disponibilidade de recursos (quantidade de memória, espaço em disco, vazão da rede, etc.) não deverá ser igual entre os processadores. Tudo isso para garantir a utilização de recursos não especializados, disponíveis em qualquer ambiente informatizado de uso geral.

Entre os recursos disponíveis nos processadores, um deles merece destaque: a memória estável. Entende-se por memória estável aquela que, quando necessário um desligamento do processador ou quando uma falha ocorrer no mesmo, todas as informações nela confiadas ficarão protegidas e estarão disponíveis posteriormente. No modelo adotado neste trabalho, a memória estável será utilizada para o salvamento dos *checkpoints*. Considerou-se os recursos de disco rígido dos processadores como alternativa para implementação de memória estável[2].

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

Os processadores e a rede de comunicação podem falhar, causando algum tipo de inconsistência no processamento da aplicação distribuída, sendo necessária, portanto, a sua recuperação a partir de um estado consistente previamente estabelecido. Esses eventos, entretanto, serão tratados como defeitos nos processos do sistema. Dessa forma, assume-se que os processos podem sofrer colapso (*crash*) causado por falhas no ambiente de execução. Defeitos de colapso, segundo o modelo definido por Cristian (1991), causam a parada total do processamento ou a perda do estado interno dos processos. Esses defeitos serão percebidos pelo detector quando o processo parar de se comunicar por um intervalo de tempo pré-estabelecido ou quando o próprio processo, por ter sofrido alguma falha ou gerado alguma operação não tolerada pelo sistema operacional, tiver sua execução incondicionalmente interrompida.

O comportamento do detector de defeitos implementado neste trabalho é similar ao do modelo *lazy*, apresentado Fezter *et al.* (2001). Neste modelo, o protocolo utiliza as mensagens da aplicação para “poupar” mensagens de detecção, as quais somente são utilizadas quando não houver comunicação entre os processos após um limite de tempo

[2] O uso de disco rígido como solução para memória estável não corresponde rigorosamente à definição de memória estável, pois não apresenta garantia para o colapso do meio de armazenamento. Entretanto, é adequado para os propósitos do presente trabalho, que se concentram na prototipação e demonstração de viabilidade técnica.

estabelecido. São utilizados os tempos locais (*hardware clock*) como referência para calcular esse limite. Para detectar processos suspeitos de falha, o processo que executa a monitoração armazena em um *array* a identificação dos processos monitorados e o instante de tempo local em que a última mensagem foi recebida destes processos. A cada mensagem recebida do processo monitorado, o seu respectivo instante de tempo é atualizado no *array*. Um processo monitorado passa a ser suspeito quando dele não for recebida nenhuma mensagem após um determinado período de tempo, momento em que mensagens específicas de detecção são utilizadas para sondar o respectivo processo. Caso não haja resposta à sondagem, a suspeita é confirmada e assume-se que o processo monitorado sofreu colapso, sendo então iniciados os mecanismos de recuperação.

Para a implementação dos mecanismos de recuperação, é necessário ainda que as falhas sejam do tipo temporárias, ou seja, seu efeito desaparece após algum tempo. O efeito da falha pode desaparecer, por exemplo, devido ao fato do defeito ter sido reparado ou do componente defeituoso ter sido substituído (SCHLICHTING, 1983). Falhas temporárias podem corresponder, por exemplo, a situações em que somente o processo da aplicação distribuída apresentou defeito, sendo finalizado e substituído por outro, a partir do último *checkpoint* consistente. Ou ainda, problemas momentâneos de particionamento da rede ou mesmo desligamento forçado de um dos processadores do sistema distribuído, com posterior re-ligamento, causado por problemas de interrupção de energia. Em todos os casos, deverá ser possível executar os mecanismos de recuperação buscando a retomada do processamento da aplicação a partir de um momento anterior ao da manifestação da falha. A execução desses mecanismos poderá ser automática, executada pelo próprio sistema de *checkpointing*, ou deverá sofrer interferência do usuário do sistema.

A aplicação distribuída é formada por um conjunto finito de processos distribuído entre os processadores. A computação distribuída realizada pela aplicação é modelada por uma seqüência de eventos geralmente associados à troca de mensagens. A aplicação distribuída poderá ser executada tanto no ambiente original quanto no ambiente acrescido do sistema de *checkpointing*. Dessa forma, é necessário um nível de transparência que não requeira adaptações da aplicação, a qual não perceberá a presença do sistema de *checkpointing.*

O sistema de *checkpointing*, por sua vez, também faz parte do modelo de computação distribuída, tendo parte de seus módulos inseridos como uma camada intermediária, entre a aplicação e o sistema operacional. Faz parte do sistema de *checkpointing*, da mesma forma que fazem parte das aplicações, um conjunto de processos e componentes, distribuídos entre os nodos do sistema. Basicamente o sistema de *checkpointing* é formado por três módulos: módulo de recuperação, módulo de detecção de defeitos e módulo de interceptação.

No módulo de recuperação são implementados os mecanismos para o estabelecimento das linhas de recuperação e, em caso de falhas, a restauração dos estados dos processos e a retomada das atividades. A detecção de defeitos deve informar sobre a ocorrência destes nos processos da aplicação e no meio de comunicação. Além disso, deve informar sobre a restauração de um componente defeituoso. O módulo de interceptação, integrado ao módulo de comunicação, é responsável pelos mecanismos de interceptação e anexação de informações de controle às mensagens da aplicação.

O módulo de comunicação é responsável pela troca de informações: envio, recebimento e entrega de mensagens. Nesse módulo estão presentes os mecanismos

básicos de troca de mensagens providos pelo sistema operacional associados a mecanismos de interceptação de mensagens da aplicação, necessários para o funcionamento do algoritmo utilizado.

Os processos da aplicação podem assumir o papel de transmissor ou de receptor das mensagens, as quais serão interceptadas e processadas pelo sistema de *checkpointing*. Na Figura 2.1 é apresentada uma simplificação do modelo de computação distribuída, onde estão representados os processos da aplicação e os módulos que fazem parte do sistema de *checkpointing*.

Da mesma forma que os processos da aplicação, o sistema de *checkpointing* também troca mensagens para executar suas operações. Dessa forma, as mensagens geradas e processadas pelo processo da aplicação serão denominadas **mensagens da aplicação** e as mensagens geradas pelo sistema de *checkpointing* serão denominadas **mensagens de sistema ou mensagens de controle**. Alternativamente, o sistema de *checkpointing* faz uso das mensagens da aplicação como meio de transporte das informações de controle, através da interceptação dessas mensagens e da anexação de informações na transmissão e remoção na recepção.

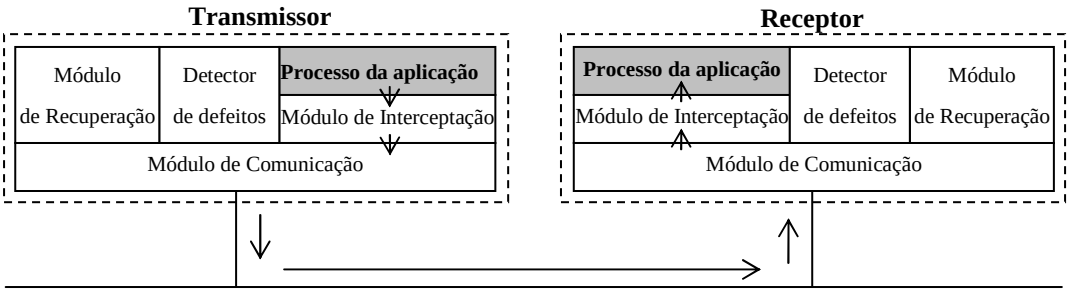


Figura 2.1: Modelo de computação distribuída

Na transmissão, uma mensagem gerada pela aplicação e passada para o módulo de comunicação é interceptada pelo módulo de interceptação, processada e devolvida para o módulo de comunicação. Este, por sua vez, encarrega-se de transmitir a mensagem pela rede de comunicação.

Na recepção, uma mensagem recebida pelo módulo de comunicação do receptor é interceptada e processada pelo módulo de interceptação, antes de ser entregue para o processo da aplicação.

O módulo detector de defeitos atua tanto nos processos da aplicação quanto na comunicação, informando ao módulo de recuperação sobre a identificação de um defeito ou recuperação desses componentes.

2.2 Algoritmo Utilizado

O algoritmo desenvolvido por Cechin (2002) em sua tese de doutorado veio ao encontro da motivação proposta para o presente trabalho, sendo o algoritmo adotado para implementação, buscando-se a comprovação de sua eficiência e das suas características de comportamento. Associado ao fato do algoritmo ter sido desenvolvido dentro do Grupo de Tolerância a Falhas do Instituto de Informática da UFRGS (diferencial no que se refere ao acesso aos trabalhos relacionados e à proximidade dos pesquisadores envolvidos), destacou-se a oportunidade de implementar um algoritmo recentemente

proposto, o qual ainda não possuía subsídios práticos (baseados em resultados reais) que possibilitassem complementar a sua avaliação.

Outro fator que favoreceu a escolha do algoritmo foi a preocupação manifestada pelo autor com sua implementação. Ao longo do teu trabalho, o autor cita alternativas para implementar os mecanismos utilizados no algoritmo. Além disso, a partir de uma análise superficial do algoritmo, foi possível identificar a possibilidade de implementação de todos os mecanismos em ambientes já conhecidos, como no sistema operacional Linux. O mesmo não ocorreu na investigação de outros algoritmos, como os apresentados e avaliados na tese de Cechin e outros encontrados na literatura, citados por Elnozahy *et al.* (2002). Em poucos casos, há uma preocupação com a viabilidade de implementação dos algoritmos e cumprimento das premissas estabelecidas, sendo a apresentação dos modelos teóricos e a prova de consistência os principais objetivos.

O trabalho de implementação apresentado ao longo do texto deve obedecer à especificação do algoritmo escolhido, utilizada como base para consolidar os mecanismos que fazem parte do sistema de *checkpointing*. O sistema deve implementar todos os mecanismos básicos que fazem parte do algoritmo adotado. Mecanismos auxiliares poderão ser implementados utilizando especificações próprias.

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

Para tolerar falhas na coordenação, o autor sugere a implementação de rotação de coordenadores, onde, a cada rodada, um processo diferente assume a tarefa de coordenação. Nessa sugestão, os processos deverão ser numerados segundo a sua ordem de criação, a qual determinará a ordem de rotação dos coordenadores, sendo que, após o último, retorna-se ao primeiro, formando uma lista circular. Dessa forma, a cada estabelecimento de um novo ponto de recuperação, apenas um processo será responsável pela tarefa de coordenação. A contrapartida é garantir que todos os outros processos identifiquem inequivocamente quem é o coordenador.

Ainda como operação auxiliar, após o estabelecimento das linhas de recuperação, o algoritmo indica os momentos adequados a partir dos quais é possível descartar os pontos de recuperação que deixaram de serem úteis, ou seja, a coleta de lixo. Essa coleta é acionada pelo processo coordenador ao término do estabelecimento de cada linha de recuperação e reduz consideravelmente a necessidade de memória utilizada pelo sistema para armazenamento dos pontos de recuperação, uma vez que descarta aqueles que podem ser substituídos por outros mais recentes.

Por definição, o algoritmo opera com duas linhas de recuperação em memória estável. Uma, identificada no algoritmo por `CurrCP`, corresponde à linha de recuperação em construção. Ela vai existir durante o período de tempo em que os processos estiverem estabelecendo uma nova linha de recuperação. A outra, identificada por `PrevCP`, corresponde à linha de recuperação mais antiga e portanto estável (não está sendo alterada). Ao término do estabelecimento de cada nova linha de recuperação (`CurrCP`),

ela substitui a antiga, sendo efetuada a atualização de `PrevCP` (`PrevCP` = `CurrCP`). Com isso, ela assume a identificação da nova linha de recuperação estabelecida e, concomitantemente, permite o descarte da antiga, a qual poderá ser removida da memória estável pela operação de coleta de lixo. Dessa forma, na recuperação, vai existir pelo menos uma linha de recuperação no sistema.

Como parte dos mecanismos de consistência do algoritmo, uma vez que o algoritmo não assume a existência de canais FIFO, todas as mensagens (mensagens da aplicação e mensagens de sistema) têm associado um índice que identifica qual foi o último ponto de recuperação estabelecido pelo seu processo transmissor, denominado “índice do intervalo do ponto de recuperação”, correspondente ao identificador `CurrCP` citado no parágrafo anterior. Ao receber uma mensagem, o receptor verifica este índice. Se for maior que o seu próprio índice, então um novo ponto de recuperação deve ser estabelecido antes da entrega da mensagem para a aplicação. Caso contrário, a mensagem é entregue sem que sejam realizados outros procedimentos. A associação desse índice às mensagens da aplicação deverá ser efetuada de forma transparente, adicionando o índice nas mensagens no momento de sua emissão e removendo-o na recepção, antes de serem entregues para a aplicação.

O algoritmo foi projetado para permitir a troca de mensagens da aplicação simultaneamente às mensagens de coordenação, necessárias ao estabelecimento das linhas de recuperação. Isso significa que, durante as tarefas de coordenação, não há a necessidade de bloqueio da aplicação e tampouco o bloqueio do canal de comunicação para evitar troca de mensagens entre os processos distribuídos que compõem a aplicação.

O controle de mensagens perdidas e de mensagens órfãs também é efetuado pelo algoritmo. As mensagens podem ser perdidas por falha no meio de comunicação ou devido ao próprio mecanismo de recuperação. O algoritmo determina o salvamento de todas as mensagens enviadas pela aplicação. Dessa forma, caso haja necessidade de retransmissão de uma mensagem da aplicação caracterizada como perdida, esta poderia ser recuperada.

A existência de mensagens órfãs pode ocorrer nos casos em que, no momento da recuperação da aplicação a partir de uma linha de recuperação previamente estabelecida, existam mensagens registradas como recebidas pelo processo receptor mas não há o correspondente registro de que elas tenham sido enviadas pelo emissor correspondente. O algoritmo resolve o problema das mensagens órfãs através de mecanismos de consistência que evitam que elas ocorram.

O retorno consistente é garantido pela existência de uma linha de recuperação previamente estabelecida pelo algoritmo. Os mecanismos de recuperação serão disparados a partir da detecção de defeitos causados por falhas previstas no modelo ou nas situações de retomada do processamento interrompido intencionalmente.

Para verificar a correção do algoritmo, Cechin optou pela utilização da lógica temporal (TLA) de Lamport (1994), usada na especificação dos mecanismos e na demonstração de sua correção. No Anexo A, é reproduzida a especificação do algoritmo. A prova de correção, bem como maiores detalhes do algoritmo, podem ser obtidos na publicação original do trabalho (CECHIN, 2002).

A partir das propriedades do algoritmo, foi possível identificar os principais mecanismos a serem implementados e que deverão delimitar o cenário do presente trabalho.

O primeiro mecanismo refere-se ao salvamento do estado dos processos da aplicação distribuída. Esse mecanismo deverá capturar e salvar, em memória estável, as informações necessárias para a recuperação da aplicação em caso de falha. Além disso, deverá ser realizado de forma transparente, sem a necessidade de reescrita ou adaptação das aplicações. Da mesma forma, deverá efetuar o processo inverso, ou seja, recuperar as aplicações a partir dos *checkpoints* previamente estabelecidos.

O segundo mecanismo está relacionado aos conceitos de consistência do algoritmo, mais precisamente à manipulação das mensagens da aplicação. Como exposto anteriormente, o tratamento de mensagens perdidas sugere o salvamento das mensagens enviadas pela aplicação para posterior retransmissão, caso necessário. Associado a esse mecanismo, também é sugerida a anexação de informações de controle às mensagens da aplicação. Isso significa que as mensagens da aplicação deverão ser interceptadas, de forma transparente, para que sejam devidamente processadas, de acordo com as premissas do algoritmo.

Por último, estão os mecanismos de coordenação empregados pelo algoritmo. Estes deverão ser executados paralelamente à execução da aplicação, também de forma transparente ao programador, e deverão implementar as ações do algoritmo, além de executar e gerenciar os dois mecanismos anteriores.

2.3 Infra-estrutura Disponível

A definição do ambiente de implementação do algoritmo e de condições de funcionamento (em oposição a uma implementação genérica) tornou-se obrigatória para viabilizar a programação do sistema de *checkpointing*. Alguns mecanismos empregados pelo algoritmo influenciaram na escolha do ambiente de implementação e das ferramentas de apoio. Decisões como a escolha do sistema operacional, adoção alternativa de bibliotecas e ambientes de simulação, estratégias de programação e definição do perfil das aplicações-alvo são requisitos que, previamente definidos, limitam o escopo do trabalho e viabilizam sua realização.

Com base nessas considerações, no modelo de sistema distribuído e nos objetivos estabelecidos, fez-se necessário determinar, a partir de uma investigação da infra-estrutura disponível, um ambiente adequado para implementação e execução dos mecanismos de tolerância a falhas que comportam o sistema de *checkpointing*.

Nesta seção, serão apresentadas alternativas identificadas de infra-estrutura para implementação do sistema proposto, procurando estabelecer um paralelo com os mecanismos exigidos pelo algoritmo.

2.3.1 Ambientes simulados

Um caminho possível a ser seguido é a experimentação do algoritmo sobre um ambiente simulado, ou seja, sobre uma ferramenta que simule o ambiente distribuído definido no modelo. Para tanto, existem algumas ferramentas de simulação cujo uso foi cogitado para simular o ambiente de execução definido. Exemplos dessas são as ferramentas Neko (URBÁN; DÉFAGO; SCHIPER, 2001), Cesium (ALVAREZ; CRISTIAN, 1997) e NS – *Network Simulator* (NS, 2005). Entretanto, a avaliação preliminar destas demonstrou sua inadequação para estudos detalhados de *checkpointing* pois, apesar de fornecerem primitivas para a implementação de alguns mecanismos de tolerância a falhas e/ou sistemas distribuídos, não dispõem de facilidades relacionadas à

implementação de *checkpoints* e recuperação do estado dos processos, operações básicas, porém complexas de serem implementadas, exigidas pelo algoritmo. O suporte para simulação de falhas nestas também é relativamente incipiente, como demonstrado em Trindade (2003) e Rodrigues (2005), de tal forma que não deveria ser despendido um esforço significativo na complementação do ambiente básico.

Outro ponto negativo no que diz respeito ao uso de ferramentas de simulação é a disponibilidade de aplicações. Por possuírem objetivos exclusivos de simulação, normalmente essas ferramentas limitam as experimentações a execuções de funções ou rotinas, ao invés de aplicações concretas, exercitando os componentes restritos ao contexto do ambiente simulado, muitas vezes ignorando problemas que se manifestam em ambientes reais. Nos casos em que aplicações são implementadas sobre os simuladores, essas devem ser implementadas na mesma linguagem de programação da própria ferramenta, em alguma linguagem pré-definida ou ainda utilizando funções restritas ao ambiente. Essas características, por si só, podem restringir o uso de bibliotecas auxiliares, exigindo a programação de todos os mecanismos sugeridos pelo algoritmo e distanciando essa execução de sua implementação verdadeira.

2.3.2 Ambientes reais

A implementação do algoritmo utilizando os recursos disponíveis no próprio sistema operacional, sem o uso de ferramentas de simulação ou alguma infra-estrutura especializada, revelou-se como uma alternativa a ser investigada. Dos sistemas operacionais disponíveis, foram investigados os mais difundidos atualmente: MS Windows e Linux.

Para o primeiro, pouco se localizou na literatura a respeito de trabalhos relacionados à recuperação por retorno em sistemas distribuídos. Entretanto, através de uma consulta nas propriedades desse sistema operacional, concluiu-se ser uma alternativa viável para uma implementação completa, uma vez que é possível implementar todos os mecanismos exigidos pelo algoritmo. Baseado em Win32 API, que é a interface das aplicações com o sistema operacional, é possível implementar todas as funcionalidades exigidas pelo algoritmo através de alguma linguagem de programação (NEBBET, 2000), tais como interceptação de mensagens da aplicação e salvamento e recuperação do estado dos processos.

Ao contrário do primeiro, existe uma riqueza de propostas e trabalhos baseados no sistema operacional Linux. Apesar de algumas implementações de *checkpointing* existentes em Linux serem bastante restritas pelo uso de infra-estrutura especializada (SANKARAN *et al.*, 2003), seu código aberto e sua arquitetura amplamente divulgada, como apresentado por Beck *et al.* (1999), Bar (2000) e Rubini (1999), fizeram desse sistema operacional um ponto de partida bastante confortável para o desenvolvimento do sistema de *checkpointing* proposto.

No contexto do presente trabalho, onde o algoritmo impõe o uso de mecanismos não convencionais, algumas alternativas de implementação em Linux foram investigadas visando verificar sua viabilidade. Primeiramente foram identificadas alternativas para a implementação de mecanismos de interceptação de mensagens e manipulação de processos das aplicações. Um exemplo dessas alternativas é a utilização de chamadas de sistema (*system call*), como a chamada `ptrace`, que permite a um processo o controle total sobre outro processo (BECK *et al.*, 1999). Detalhes dessa, e de outras técnicas de interceptação, bem como avaliações de desempenho podem ser encontradas no trabalho

de Fontoura (2002), onde foi apresentado um estudo sobre as possíveis abordagens de captura de informações da aplicação. Desse estudo, merece destaque a abordagem de interceptação de chamadas de sistema em espaço de *kernel*, bastante adequada para implementar a interceptação de mensagens da aplicação de forma transparente e sem implicar em uma sobrecarga significativa para o sistema operacional.

Quanto às alternativas para implementação dos mecanismos de *checkpoint* e recuperação de processos, foram encontradas informações suficientes para sustentar a possibilidade de uma implementação transparente sob o ponto de vista das aplicações. Isso significa que existem técnicas que permitem, por exemplo, o salvamento do estado de uma aplicação sem a necessidade de alterar o seu código-fonte, para este fim.Visando obter embasamento para desenvolver código-fonte específico ao salvamento de pontos de recuperação e sua retomada posterior, foram estudadas algumas bibliotecas reunidas no sítio *Checkpointing.org* (2003). São exemplos destas: `libckpt` (PLANK, 1995), `epckpt` (PINHEIRO, 1999), `CRAK` (ZHONG; NIEH, 2001), `ckpt` (ZANDY, 2003) e `chpox` (SUDAKOV; MESHCHERYAKOV, 2003). Todas têm código-fonte disponível, são implementadas em linguagem C/C++ e sobre o sistema operacional Linux, explorando diferentes abordagens. `libckpt`, por exemplo, implementa uma alternativa em espaço de usuário, exigindo alteração do código-fonte e re-compilação da aplicação. Por outro lado, `epckpt` e `CRAK` permitem a utilização transparente em diferentes níveis: em espaço de *kernel* como parte do sistema operacional, em camada entre o sistema operacional e a aplicação ou como processo independente, em espaço de usuário. As bibliotecas foram utilizadas como fonte de estudo e ponto de partida para a implementação dos mecanismos de *checkpoint* sugeridos pelo algoritmo ou agregando-se suas funções na forma original.

Ainda no contexto de ambientes reais, cuja programação do sistema de *checkpointing* é baseada exclusivamente nos recursos disponíveis no sistema operacional, identificou-se dois níveis alternativos de programação: espaço de usuário e espaço de *kernel*. Alessandro Rubini (1999) e Richard Stones (1999) utilizam essa classificação para diferenciar o desenvolvimento de programas que devem executar, respectivamente, no espaço de memória reservado para os programas de usuário ou no espaço de memória reservado para rotinas de *kernel*. No primeiro, somente usuários privilegiados têm acesso a determinados recursos, o que equivale à proteção do sistema operacional contra códigos mal elaborados ou maliciosos. No segundo, o acesso aos recursos do sistema é completo. Entretanto, exige um profundo conhecimento do *kernel* por parte do programador para que este possa implementar suas rotinas, uma vez que não existem bibliotecas de funções nesse contexto, facilidade amplamente encontrada em espaço de usuário.

2.4 Alternativa Adotada

Os resultados da investigação das alternativas disponíveis determinaram a implementação do sistema de *checkpointing* diretamente sobre o sistema operacional Linux. A disponibilidade de bibliotecas que oferecem parte dos mecanismos necessários, como *checkpointing* e recuperação de processos, pôde ser explorada para reduzir consideravelmente os esforços de implementação do sistema. Nesse aspecto, a principal decisão, que praticamente definiu o ambiente de desenvolvimento, foi a escolha da biblioteca `CRAK` como ponto de partida para os trabalhos de programação.

Considerou-se, portanto, a manipulação do estado dos processos da aplicação como o ponto mais delicado do trabalho, que demandaria a maioria dos esforços caso fosse

implementada na sua totalidade. Além disso, a biblioteca destacou-se entre as alternativas por ser uma implementação em espaço de *kernel* e por manipular a maior quantidade de recursos alocados pelos processos. A primeira característica facilita o uso de forma transparente, sem a necessidade de adaptações nas aplicações. A segunda característica amplia o cenário das aplicações com as quais o sistema é compatível.

A partir de adaptações na biblioteca adotada, a serem abordadas no Capítulo 4, foi possível implementar os mecanismos de interceptação de mensagens da aplicação com base nas técnicas sugeridas por Fontoura, também de forma transparente. Dessa forma, com a utilização de uma única biblioteca, foram implementados dois dos mecanismos básicos do algoritmo: salvamento e recuperação de processos e interceptação e processamento de mensagens da aplicação. Restaram apenas os mecanismos de gerência do algoritmo, os quais não exigem a existência de bibliotecas especializadas e, portanto, poderiam ser inseridos de acordo com a conveniência. Por questões de facilidade, estes últimos mecanismos foram implementados sob a forma de um processo, em espaço de usuário.

É importante observar que, de acordo com algumas premissas do algoritmo, tal como a comunicação baseada exclusivamente na troca de mensagens, e na forma como o sistema de *checkpointing* foi implementado, existirão limites nos níveis de compatibilidade do sistema com as aplicações existentes. A caracterização das aplicações e eventuais restrições em seus perfis serão apresentadas na Subseção 3.2.1. Exemplos dessas aplicações serão apresentados na Seção 5.2.

2.5 Trabalhos Relacionados

Existem na literatura trabalhos envolvendo técnicas de *checkpointing* e recuperação de aplicações distribuídas (ELNOZAHY *et al.*, 2002). Grande parte dos trabalhos concentra-se na especificação e construção de algoritmos, buscando reduzir o número de operações requeridas em busca da consistência e sincronização dos *checkpoints* e redução da sobrecarga imposta em cenários livres de falhas. Alguns trabalhos preocupam-se também com a recuperação propriamente dita, propondo mecanismos que facilitem a retomada do processamento após a manifestação dos defeitos.

Muitos destes trabalhos carecem de implementações para subsidiar a avaliação dos mesmos. Existem poucos estudos relacionados à avaliação de *checkpointing* consistente e desempenho dos respectivos algoritmos em cenários sujeitos a ocorrência de falhas. Dos trabalhos de implementação identificados, a maioria baseia-se no uso de cenários idealizados sobre arquiteturas especializadas (*MPI*, *Grids*, *Corba*, etc.) para a execução das aplicações distribuídas, o que abstrai grande parte dos problemas enfrentados em ambientes de uso geral.

Krishnan e Gannon (2004), por exemplo, implementam *checkpointing* e recuperação de aplicações distribuídas que executam sobre uma arquitetura em grade (*grid*). O trabalho envolve a construção de APIs adicionais para implementação do *checkpointing*, que é particularizada para a arquitetura proposta (em grade) e baseada em componentes distribuídos. Essas APIs podem ser usadas pelos programadores das aplicações para adaptá-las, possibilitando salvar e recuperar os seus estados. Os resultados desse trabalho permitem concluir que a sobrecarga imposta pelo sistema de *checkpointing* cresce linearmente com o número de componentes da aplicação distribuída (*workers*) e com a quantidade de dados (tamanho dos *checkpoints*) a serem salvos.

Existem ainda sistemas que implementam *checkpointing* distribuído para aplicações baseadas em MPI e PVM. Sankaran *et al.* (2003) e Stellner (1996) implementam sistemas desse tipo. Essa abordagem exige que as aplicações executem em sistemas agregados (*clusters*), o que dificulta a portabilidade das aplicações.

Outra abordagem encontrada na literatura trata da implementação de bibliotecas de *checkpointing* distribuído para serem agregadas em sistemas já existentes, tais como o sistema Condor (BASNEY; LIVNY; TANNENBAUM, 1997) e a linguagem Orca (KAASHOEK *et al.*, 1992). É possível encontrar também implementações relacionadas a *checkpointing* em trabalhos envolvendo persistência de componentes nas arquiteturas tradicionais baseadas em componentes, como CCM (*CORBA Component Model*), EJB (*Enterprise Java Beans*) e DCOM (*Distributed Component Object Model*).

Por outro lado, Bhargava *et al.* (1990) e Elnozahy (ELNOZAHY; ZWAENEPOEL, 1992a) efetuaram experimentos buscando a avaliação de algoritmos de *checkpointing* distribuído em cenários desprovidos de infra-estrutura especializada. Enquanto o primeiro concluiu que as mensagens utilizadas para sincronização do protocolo constituem a principal sobrecarga no sistema, o segundo concluiu que o gargalo era decorrente da escrita dos *checkpoints* em memória estável, propondo alternativas para reduzir essa sobrecarga. A sobrecarga observada nesses trabalhos variou de 3% para *checkpointing* otimista até 5,8% para *checkpointing* consistente. Ambos os trabalhos utilizaram aplicações que manipulavam pouca quantidade de memória, gerando *checkpoints* modestos cujo volume de informações variava entre 4 e 48 kilobytes. Não foram implementados mecanismos de recuperação das aplicações, restringindo-se a análises de execuções em cenários livre de falhas.

O presente trabalho procura mostrar a viabilidade de implementar *checkpointing* distribuído, de forma transparente e sem a utilização de infra-estrutura especializada para garantia dos serviços. A transparência referenciada aqui indica que não é necessário inserir ou adaptar código nas aplicações para definição ou controle dos mecanismos de recuperação. Os resultados que serão apresentados no Capítulo 5 mostrarão que, mesmo sem qualquer otimização nos mecanismos que correspondem ao gargalo do sistema, a sobrecarga máxima foi de 1.73% em cenários equivalentes aos utilizados por Bhargava e Elnozahy. Além de gerar resultados reais que possibilitam complementar a avaliação do algoritmo utilizado, o presente trabalho também implementa mecanismos para garantir a recuperação das aplicações distribuídas a partir dos *checkpoints* estabelecidos, o que também contribui para comprovar os mecanismos de consistência que compõem o algoritmo implementado.

3 O MODELO DE IMPLEMENTAÇÃO DO SISTEMA

O modelo adotado na implementação leva em consideração os seguintes aspectos conceituais: ambiente de implementação, representando o ambiente de desenvolvimento e execução do sistema, formado basicamente pela plataforma operacional e sua infra-estrutura; a forma de utilização do sistema de *checkpointing* pelo usuário desse ambiente, envolvendo a instalação e os passos para utilização do sistema de *checkpointing* na execução de suas aplicações; o modelo, ou arquitetura, do sistema desenvolvido, caracterizando todos os seus componentes e identificando a localização de seus módulos nos diferentes espaços do sistema operacional; e por fim, o ciclo de vida do sistema de *checkpointing* representando o conjunto de ações executadas pelo mesmo, desde sua inserção no sistema operacional até sua remoção. Espera-se estabelecer uma linguagem sobre o funcionamento do sistema, a ser usada no capítulo referente à implementação.

3.1 Ambiente de Implementação

O ponto de partida adotado para a implementação e execução de tolerância a falhas foi a definição do ambiente de desenvolvimento adotado. Entende-se por ambiente de implementação o sistema operacional e sua infra-estrutura: modelo de funcionamento, comunicação, ferramentas de desenvolvimento, bibliotecas, documentação, compiladores, etc.

A partir da investigação apresentada no Capítulo 2, optou-se pela implementação desse trabalho sobre a plataforma Linux. Essa escolha levou em consideração principalmente a disponibilidade de bibliotecas que poderiam ser agregadas com o objetivo de reduzir o esforço de programação. A linguagem de programação adotada foi C/C++. Sua escolha foi influenciada pela familiaridade já existente, pela possibilidade de integração com bibliotecas auxiliares e por ser compatível com programação em nível de *kernel*.

Não é objetivo desse trabalho apresentar um estudo detalhado sobre o sistema operacional Linux, tampouco sobre linguagens de programação existentes. Serão apresentados, nessa seção, somente os recursos diretamente relacionados com a implementação e com o modelo de sistemas distribuídos definido no Capítulo 2. Baseado naquele modelo e nos requisitos do algoritmo escolhido, deve-se definir o conceito de processo, componente básico de uma aplicação de usuário segundo Tanenbaum (2001), formas de comunicação entre processos, chamadas de sistema (*system calls*) e níveis de execução do sistema operacional. Uma apresentação mais detalhada sobre esses e demais recursos estão reunidas no trabalho de Fontoura (2002).

3.1.1 Modelo de processos

O modelo de sistema distribuído definido por Cechin (2002) para especificar seu algoritmo envolve o conceito de processos. Esses, por sua vez, são os componentes que executam o processamento da aplicação distribuída e comunicam-se através da troca de mensagens. Segundo a visão de Tanenbaum (2001), processo é o conceito central de qualquer sistema operacional. É uma abstração de um programa em execução consumindo recursos do sistema. Ainda, um processo também é uma instância de um programa em execução (MICHELL; OLDHAM; SAMUEL, 2001).

Um processo em Linux é implementado seguindo os mesmos conceitos citados anteriormente. Cada processo possui um identificador (*PID – Process ID*), gerenciado pelo *kernel* do sistema operacional. Esse identificador e demais informações sobre o processo são armazenados em uma estrutura fundamental chamada `task_struct`. A partir dessa estrutura é possível ao *kernel* do Linux gerenciar os segmentos de memória do processo, os registradores, os estados de execução e demais recursos alocados pelo processo. A criação e extinção dos processos são dinâmicas, e estes evoluem de acordo com a necessidade de execução de novos programas. Um processo pode ser criado pelo sistema operacional ou por outro processo. Em ambos os modos, podem ser usados diferentes chamadas de sistema, tais como `fork` e `execve`. A chamada `execve` é normalmente invocada quando uma aplicação é iniciada pelo usuário.

Há duas maneiras possíveis para que um processo termine sua execução: terminação invocada pela aplicação que o criou, classificada nesse trabalho como terminação normal; ou terminação forçada pelo sistema operacional. A primeira ocorre quando a aplicação encerra o processamento e finaliza sua execução. Por exemplo, a função `main` de um programa implementado em C/C++ invoca o comando `exit` ou `return`. A segunda pode ocorrer quando há necessidade de encerrar a execução de uma aplicação antes que ela termine normalmente ou quando a aplicação gera algum tipo de erro ou operação ilegal, situação em que o *kernel* se encarrega de abortá-la. Um exemplo de como forçar a finalização de um processo é através do envio de mensagens especiais para o processo, denominadas *signals*. Uma mensagem especial `SIGKILL` enviada para um processo causa sua imediata finalização.

O mesmo mecanismo de sinais usados para forçar a finalização de um processo pode ser usado para controlar sua execução. A versão 2.4 do *kernel* do Linux adotada no presente trabalho possui 62 diferentes tipos de mensagens *signals*. Dessas, são empregadas, além da mensagem `SIGKILL`, mensagens `SIGSTOP` e `SIGCONT`, utilizadas respectivamente para forçar uma pausa momentânea no processamento da aplicação e para interromper essa pausa. Maiores detalhes do uso desses mecanismos serão apresentados no próximo capítulo.

Um processo pode fazer parte de uma tarefa de sistema ou de tarefa de usuário. A forma de execução é a mesma, o que diferencia uma classe de outra são os níveis de permissão de acesso aos recursos. Um processo executando em nível de sistema, normalmente, possui acesso total aos recursos do sistema operacional e faz parte do próprio sistema operacional. Um processo executando em espaço de usuário possui acesso aos recursos associados ao contexto do usuário que o criou. Independente dessa distinção, todos são gerenciados da mesma forma pelo sistema operacional.

Uma aplicação pode utilizar um único processo ou ser composta por vários processos que executam em conjunto uma determinada tarefa, tornando-a mais robusta (MICHELL;

OLDHAM; SAMUEL, 2001). Nessa situação, os processos adicionais podem ser criados ou finalizados durante a execução da aplicação, de acordo com a sua necessidade. Em se tratando de uma aplicação distribuída, os processos componentes estão distribuídos entre diferentes computadores e comunicam-se na medida em que a computação evolui.

Para o contexto do presente trabalho, no que diz respeito ao número de processos das aplicações, restringiu-se àquelas formadas por um único processo em cada computador. Essa restrição foi adotada por questões de simplicidade na implementação dos mecanismos necessários.

3.1.2 Comunicação entre processos

Comunicação é a transferência de dados entre processos. Essa transferência pode ser implementada de várias formas, variando de acordo com o sistema operacional. Nesse trabalho, serão citadas cinco maneiras adotadas pelo Linux para implementar comunicação entre processos (STONES; MATTEW, 1999): memória compartilhada, memória mapeada, *pipe*, *FIFO* e *sockets*.

- **Memória compartilhada** permite aos processos comunicarem-se através da escrita e leitura em uma área de memória acessível a ambos os processos da aplicação.
- **Memória mapeada** é similar à memória compartilhada, exceto pelo fato da área de memória estar associada com um arquivo.
- ***Pipe*** é um canal serial que permite comunicação unidirecional entre processos.
- ***FIFO*** (*first-in-first-out*), também denominado de *named-pipe*, é similar a *pipe*, exceto pelo fato de estar associado com um arquivo.
- ***Sockets*** permite comunicação bidirecional entre processos, normalmente utilizado quando os processos estão localizados em diferentes computadores.

Destas cinco alternativas acima, com base no modelo de sistema distribuído apresentado no capítulo anterior, será adotado somente o mecanismo de *sockets* para a comunicação entre processos. Escolheu-se este mecanismo devido a sua representatividade em sistemas distribuídos, além de ser bastante utilizado por aplicações como *telnet*, *rlogin*, *FTP*, *talk* e *WWW*.

A criação de um canal de comunicação baseado em *sockets* está condicionada à especificação de três parâmetros: tipo de comunicação, endereçamento e protocolo. O tipo de comunicação controla como será tratada a transmissão de dados. Comunicação orientada à conexão garante a entrega das informações na ordem em que elas foram transmitidas. Se os pacotes enviados são perdidos ou reordenados por problemas de comunicação, o receptor automaticamente solicita a retransmissão para o emissor.

Comunicação não orientada à conexão, também referenciada por comunicação orientada a datagramas, não garante a ordem de entrega dos pacotes nem a própria entrega. Os pacotes podem chegar ao receptor em ordem diversa a usada no seu envio, e os pacotes perdidos por problemas de rede não são retransmitidos. Esse controle é transferido para o nível de aplicação.

O endereçamento especifica onde os dados serão escritos ou de onde serão lidos. Uma comunicação baseada em *sockets* é formada por no mínimo dois endereços: o endereço do emissor e o endereço do receptor. Em endereçamento de Internet, conhecido como IP

(*Internet Protocol*), também baseado em *sockets*, um endereço é formado pelo identificador do computador (endereço IP) e por uma porta de comunicação. O uso de uma porta específica possibilita que o mesmo computador possa ter diferentes endereços e conseqüentemente estabelecer mais de uma conexão.

O terceiro parâmetro especifica como os dados serão transmitidos. Exemplos são o *TCP/IP*, *AppleTalk* e *UNIX*. Nem todas as combinações dos três parâmetros introduzidos aqui são implementadas na prática. Atualmente, o protocolo *TCP/IP* consolidou-se na ampla maioria das aplicações desenvolvidas para Internet.

Em Linux, a comunicação via *sockets* é utilizada pelas aplicações através de chamadas de sistema. Normalmente, as aplicações fazem uso de bibliotecas específicas que implementam a interface entre a aplicação e estas chamadas. Cada combinação de parâmetros *sockets* está associada a um conjunto de chamadas de sistema.

O presente trabalho desenvolverá suporte limitado a aplicações que utilizam *sockets* e comunicação não orientada a conexão. Essa restrição é justificada pelas características da biblioteca de *checkpoint* utilizada, apresentada no próximo capítulo.

3.1.3 *System calls*

O *kernel* é o *software* do núcleo do sistema operacional, que executa em modo protegido e com acesso privilegiado aos registradores do *hardware* (BAR, 2000). Nele são executadas as funções que fornecem recursos para as aplicações de usuários. Alguns sistemas operacionais empregam a arquitetura de *microkernel*, onde o *software* de gerenciamento dos recursos não faz parte do núcleo e é carregado e executado por demanda, não estando necessariamente em memória. Por outro lado, alguns sistemas operacionais empregam uma arquitetura monolítica, onde todo o *software* de gerenciamento de recursos faz parte do núcleo. Esse último é comumente empregado em implementações *UNIX*, como *BSD* e *Linux*.

Em Linux, o *kernel* não é simplesmente um processo em execução no sistema. É o *software* base do sistema operacional, que controla o escalonamento dos processos em um esquema multitarefa. Ele também provê um conjunto de rotinas de acesso aos recursos, constantemente armazenadas em memória, as quais são acionadas pelos processos de usuário. Essas rotinas são acessadas pelos processos de usuário através de chamadas de sistema (*system calls*). Entretanto, o programador de uma aplicação pode fazer uso de bibliotecas de funções, que implementam toda a interface com as rotinas de sistema. Estas bibliotecas, por sua vez, fazem uso das chamadas de sistema para acesso aos recursos necessários.

Chamadas de sistema são, portanto, procedimentos especiais que transferem o controle do processamento para o *kernel*. Linux implementa cerca de 200 chamadas de sistema diferentes. No arquivo `/usr/include/asm/unistd.h` estão listadas todas as chamadas de sistema em Linux. Algumas destas são exclusivas para uso interno pelo sistema operacional, enquanto outras são usadas somente em implementações de bibliotecas especializadas.

O *kernel* é capaz de identificar o processo de usuário que executou uma determinada chamada de sistema. Essa identificação é feita através do identificador do processo (PID) e está armazenada na variável `current->pid`.

3.1.4 Espaço de *kernel* e espaço de usuário

Em Linux, assim como nos demais sistemas operacionais, os processos podem estar alocados no espaço de *kernel* e no espaço de usuário (RUBINI,1999; MATTHEW; STONES, 1999). A diferença se refere ao endereçamento de memória reservado para o *kernel* e o endereçamento reservado para os processos de usuário.

Essa divisão também é observada quando se trata de programação. Pode-se desenvolver código que será compilado e executado no espaço de *kernel* ou no espaço de usuário. O desenvolvimento em espaço de *kernel* pode ser realizado alterando-se o próprio código-fonte ou através da implementação de módulos, também referenciados por *device drivers* (RUBINI, 1999), que podem ser inseridos e removidos dinamicamente no sistema. O desenvolvimento em espaço de usuário corresponde ao método tradicional de desenvolvimento de aplicações.

Rubini (1999) apresenta em detalhes estas duas abordagens, também estudadas e comentadas por Fontoura (2002). A característica interessante dessa divisão entre diferentes espaços é que, em espaço de *kernel*, o acesso aos recursos do sistema é completo. Além do acesso aos recursos em nível privilegiado, é possível estender as funcionalidades do sistema operacional de acordo com a necessidade, seja pela alteração do seu código-fonte, seja pela implementação de módulos adicionais.

Em contrapartida, diferentemente da programação em espaço de usuário, onde é possível fazer o uso de bibliotecas de funções para facilitar o acesso aos recursos, no espaço de *kernel* as funções disponíveis são aquelas oferecidas pelo mesmo, o que pode tornar a programação mais trabalhosa dependendo do tipo de recurso que se pretende manipular.

O desenvolvimento do presente trabalho baseou-se na utilização das duas abordagens citadas anteriormente. Para desenvolver os mecanismos de nível mais baixo, como *checkpoint*, recuperação de processos e interceptação de mensagens, utilizou-se programação em espaço de *kernel*, concentrada no desenvolvimento de um módulo de *kernel*. Os demais mecanismos foram implementados em espaço de usuário, na forma de um processo, fazendo uso das funções do módulo de *kernel*.

3.2 Modelo Computacional

Baseado no objetivo central desse trabalho, que é a implementação de um sistema de *checkpointing* para ser incorporado por aplicações distribuídas, é importante que o modelo computacional seja definido. O modelo computacional é formado pela aplicação distribuída e pelo sistema de *checkpointing* inserido como recurso extra. O comportamento desses dois elementos sob o ponto de vista do usuário serão explicados nessa seção.

3.2.1 Aplicação distribuída

Conforme apresentado na Subseção 3.1.1, considera-se neste trabalho como aplicação distribuída toda aplicação composta por um conjunto de processos distribuídos em diferentes computadores, efetuando alguma tarefa em cooperação. A comunicação é baseada exclusivamente na troca de mensagens. Em cada computador reside um único processo dessa aplicação. A Figura 3.1 esquematiza o modelo de aplicação considerado nesse trabalho.

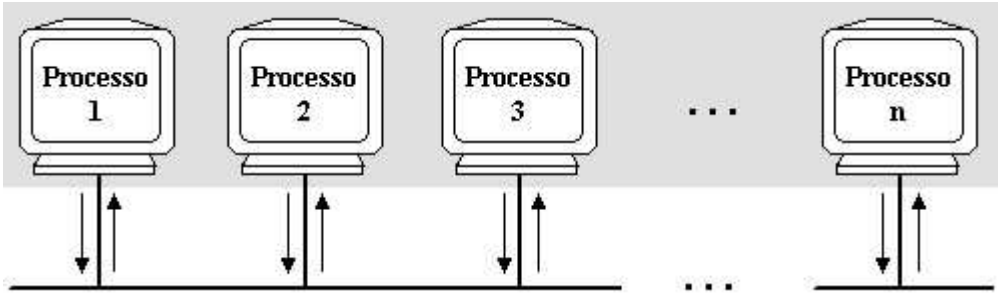


Figura 3.1: Modelo de aplicação distribuída

Como um dos objetivos do trabalho é a utilização de um ambiente distribuído convencional, sem o uso de infra-estrutura especializada, conforme discutido no Capítulo 2, assume-se que a aplicação faz uso dos recursos básicos disponíveis no sistema operacional Linux para fins de comunicação. Dessa forma, a aplicação pode ser executada em qualquer sistema compatível com Linux. A restrição imposta nesse contexto refere-se ao fato de que as aplicações devem utilizar o protocolo *UDP* com *sockets* não orientados a conexão para a troca de mensagens. A Figura 3.2 apresenta o modelo de um processo executando em um computador que utiliza os recursos do sistema operacional, tais como recursos de rede para troca de mensagens.

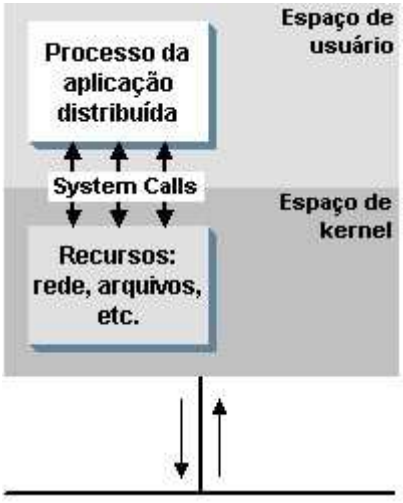


Figura 3.2: Processo da aplicação distribuída em ambiente sem *checkpointing*.

Os processos da aplicação distribuída executam necessariamente em espaço de usuário e o acesso a todos os recursos ocorre via chamadas de sistema, através do uso de bibliotecas de funções disponíveis, conforme abordado na Subseção 3.1.4. Por exemplo: a programação em C/C++ de envio de um datagrama normalmente utiliza a função `sendto`, disponível na biblioteca de funções de usuário do compilador *gcc*. Essa função é mapeada pelo compilador para a chamada de sistema `sys_socketcall`, do tipo `SYS_SENDTO`, a qual é executada pelo sistema operacional (BECK *et al.*, 1999). Detalhes dessa composição também são apresentados por Rubini (1999) e comentados por Fontoura (2002).

3.2.2 O sistema de *checkpointing*

Na Subseção 3.2.1 foi apresentado o modelo de aplicação distribuída utilizado nesse trabalho. Considerou-se que a aplicação faz uso dos recursos básicos do sistema

operacional e esse não possui infra-estrutura especializada para garantir os serviços demandados pela aplicação.

Analisando-se o comportamento das aplicações e os recursos de desenvolvimento disponíveis no sistema operacional Linux, observou-se a possibilidade de implementação do sistema de *checkpointing* como componente adicional ao ambiente de execução. Desse modo, a mesma aplicação poderá ser executada com ou sem a presença do sistema de *checkpointing*, isenta de alteração do seu código-fonte ou adaptação do código-fonte do próprio sistema operacional.

A partir dos recursos disponíveis, o desenvolvimento dos componentes do sistema de *checkpointing* foi dividido em duas partes: módulo de *kernel* e aplicação de usuário. No módulo de *kernel*, foram implementadas as operações de mais baixo nível, como interceptação de chamadas de sistema, mecanismos de *checkpoint* e recuperação de processos. A implementação desses mecanismos em espaço de *kernel*, ao contrário de uma implementação em espaço de usuário, permite uma maior otimização e, pela característica de módulos de *kernel*, permite a utilização por usuários sem necessidade de privilégios administrativos. A única intervenção do administrador é no momento da inserção desse módulo no sistema. Uma vez inserido o módulo, o serviço permanece disponível no sistema operacional e somente será executado quando for acionado, não degradando o desempenho do sistema como um todo, exceto por consumir uma pequena quantidade de memória em torno de 25 Kbytes.

Foram dois os motivos pelos quais optou-se pela implementação de parte dos mecanismos em espaço de *kernel*. O principal deles refere-se à possibilidade de escolher as chamadas de sistema a serem interceptadas. Conforme discutido no capítulo anterior, o uso de “ganchos” de *kernel* evita o comprometimento do desempenho do sistema ao permitir a seleção de chamadas de sistema específicas, ao contrário da alternativa de utilização da chamada `ptrace`, que não permite essa seleção, interceptando todas as chamadas de sistema geradas pela aplicação. O outro motivo foi a disponibilidade de uma ferramenta de *checkpoint* e recuperação com código-fonte aberto (ZHONG; NIEH, 2001) e que, apesar de necessitar de algumas adaptações a serem discutidas no próximo capítulo, mostrou-se a alternativa mais adequada dentre as disponíveis.

A gerência das operações do algoritmo foi concentrada no restante da implementação, a qual foi desenvolvida através de programação em espaço de usuário. Essa alternativa foi adotada pois permite que o sistema de *checkpointing* seja acionado pelo usuário de acordo com a sua necessidade, sem a exigência de privilégios administrativos para sua utilização. Além dessa característica, é mais confortável programar em espaço de usuário, onde é possível fazer uso de bibliotecas especializadas para implementar os demais mecanismos do algoritmo.

Portanto, o usuário possui agora duas alternativas para executar sua aplicação distribuída: uma sobre o sistema operacional sem a utilização de tolerância a falhas (Figura 3.2), e outra com a utilização de mecanismos de *checkpointing* e recuperação em caso de falha, ou seja, com tolerância a falhas. A Figura 3.3 mostra a estrutura da segunda alternativa onde, além da aplicação, aparecem os componentes do sistema de *checkpointing*.

Na Figura 3.3, a implementação de tolerância a falhas em espaço de *kernel* concentra-se no módulo **CKPT** e a implementação em espaço de usuário concentra-se no processo **CK**. Como pode ser observado, ambos operam em conjunto e podem ser removidos ou

inseridos sem a necessidade de adaptação da aplicação. Detalhes da implementação desses dois componentes serão apresentados no Capítulo 4.

A implementação também levou em consideração aspectos de operação e usabilidade desse sistema. Sem a presença do sistema de *checkpointing*, o usuário executa sua aplicação distribuída disparando os processos que dela fazem parte, nos respectivos computadores sobre os quais a aplicação vai ser executada. A forma como a aplicação trata o início e o término da computação depende de como foi implementada.

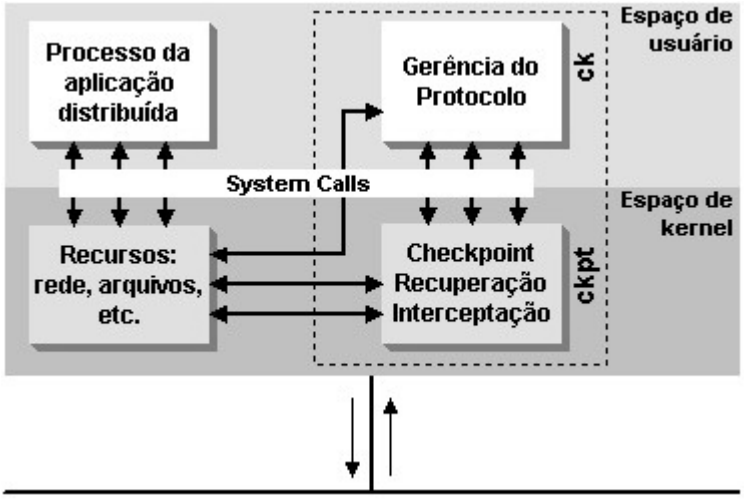


Figura 3.3: Processo da aplicação distribuída em ambiente com *checkpointing*.

Para executar uma dada aplicação com recursos de *checkpointing*, a única alteração nos procedimentos do usuário consiste na necessidade de informar antecipadamente ao sistema que a sua aplicação deverá ser executada com recursos de tolerância a falhas. O sistema de *checkpointing* se encarregará de detectar o início e término da aplicação automaticamente. Nas próximas seções, esse aspecto será discutido em detalhes.

3.3 Funcionamento do Sistema de *Checkpointing*

A presente implementação foi projetada para resultar em um sistema de *checkpointing* que rodasse concomitantemente com as aplicações e que fosse inserido no sistema operacional de forma transparente sob o ponto de vista dos programadores destas. Esse sistema deveria ainda ser independente de protocolos auxiliares, necessários para garantia de determinados serviços durante sua execução.

Dessa forma, foi necessário considerar a implementação parcial ou total dos seguintes componentes, que passaram a integrar o sistema de *checkpointing*: interceptação de mensagens da aplicação, mecanismo de *checkpointing* e recuperação, detector de defeitos, coleta de lixo, listas (*buffers*) de mensagens e comunicação em grupo para coordenação dos processos. Esses mecanismos são indispensáveis para o funcionamento do algoritmo de acordo com sua especificação.

Esta subseção tem o objetivo de apresentar e demonstrar o funcionamento do sistema de *checkpointing* (e seus componentes) baseado nas situações de estabelecimento de *checkpoints* descritas na especificação do algoritmo. Serão inicialmente apresentadas essas situações, seguidas da apresentação da arquitetura do sistema implementado e, por fim, a implementação da execução dessas situações.

3.3.1 Situações para estabelecimento de *checkpoints*

Visando garantir as premissas de consistência, o algoritmo adotado foi projetado para estabelecer *checkpoints* individuais dos processos da aplicação distribuída em duas situações: através de mensagens específicas para esse fim (*checkpoint* requisitado) ou através da detecção da necessidade de estabelecimento baseado no índice do último *checkpoint* estabelecido pelo transmissor, extraído das mensagens interceptadas (*checkpoint* induzido ou forçado). Esse índice é inserido no emissor em todas as mensagens da aplicação pelo mecanismo de interceptação. Através dele é possível dirigir os receptores a estabelecerem seus *checkpoints* sem a necessidade de ordenação das mensagens.

De acordo com a especificação do algoritmo, se necessário, será estabelecido um ***checkpoint* requisitado** quando for recebida uma requisição para esse fim, enviada pelo processo coordenador com o objetivo de estabelecimento de uma linha de recuperação consistente. Ao receber essa requisição e sendo confirmada a necessidade, bloqueia-se o processo da aplicação distribuída e estabelece-se um *checkpoint* do mesmo. Após seu desbloqueio, responde-se ao coordenador. Além disso, após essa operação, todas as mensagens enviadas pelo processo passam a conter a identificação da linha de recuperação proposta pelo coordenador. O estabelecimento do *checkpoint* local será desnecessário quando a identificação da linha de recuperação conhecida pelo processo for igual àquela requisitada pelo coordenador. Nesse caso, somente será remetida pelo processo a resposta ao coordenador, sem executar nenhuma ação adicional.

A partir do índice do último *checkpoint* estabelecido pelo transmissor, inserido nas mensagens enviadas pela aplicação, contendo a identificação da linha de recuperação proposta, é possível determinar a necessidade de estabelecer um ***checkpoint* forçado** antes que o processo da aplicação receba essas mensagens. Essa situação pode ocorrer quando uma mensagem enviada pela aplicação, após seu emissor ter estabelecido um *checkpoint*, for recebida pelo destinatário antes da correspondente requisição de tomada de *checkpoint* local, enviada pelo coordenador. Nesse caso, antecipa-se o salvamento do estado do processo garantindo-se os princípios de consistência especificados no algoritmo, sem necessidade de prover ordenação de mensagens.

Também faz parte desses princípios a necessidade de salvamento das mensagens da aplicação consideradas em trânsito nos canais de comunicação durante o estabelecimento dos *checkpoints*. Essas mensagens são aquelas que foram enviadas pela aplicação mas não se tem a certeza de que elas foram recebidas pelo destinatário, devendo ser inseridas novamente no canal de comunicação em uma eventual recuperação da aplicação. Dessa forma, elas devem fazer parte dos *checkpoints* que formam a linha de recuperação.

3.4 Arquitetura do Sistema de *Checkpointing*

Para um melhor entendimento de como as situações apresentadas anteriormente foram concretizadas na implementação, primeiramente serão apresentados a arquitetura do sistema implementado e os componentes que dele fazem parte, para depois demonstrar como o sistema implementa aquelas situações .

Na Figura 3.4 está ilustrada a arquitetura do sistema de *checkpointing*. Como observado nessa figura, os componentes estão distribuídos em dois blocos, correspondendo à divisão em espaços de programação de *kernel* e de usuário disponíveis em Linux. No primeiro estão ilustrados os componentes implementados no módulo de

kernel **CKPT**, enquanto que no segundo estão ilustrados os demais componentes, implementados no processo **CK**. A comunicação entre os componentes é ilustrada na figura através das setas, sendo a direção delas a indicação da direção do fluxo das informações, repassadas através de variáveis compartilhadas ou através de parâmetros na chamada de funções. A comunicação entre os processos distribuídos que formam o sistema de *checkpointing* (cada um possuindo a mesma arquitetura da Figura 3.4) é realizada exclusivamente através da troca de mensagens.

Essas mensagens, por sua vez, de acordo com o modelo de computação distribuída adotado no presente trabalho, são diferenciadas em dois tipos: mensagens da aplicação e mensagens de controle ou de sistema. Na Figura 3.4, as mensagens **msg1** e **msg2** ilustram, respectivamente, mensagens enviadas e mensagens recebidas pela aplicação. Por outro lado, as mensagens **msg3** e **msg4** ilustram, respectivamente, as mensagens de controle enviadas e recebidas pelo sistema de *checkpointing*. Detalhes das mensagens da aplicação e das mensagens de controle serão apresentados no Capítulo 4.

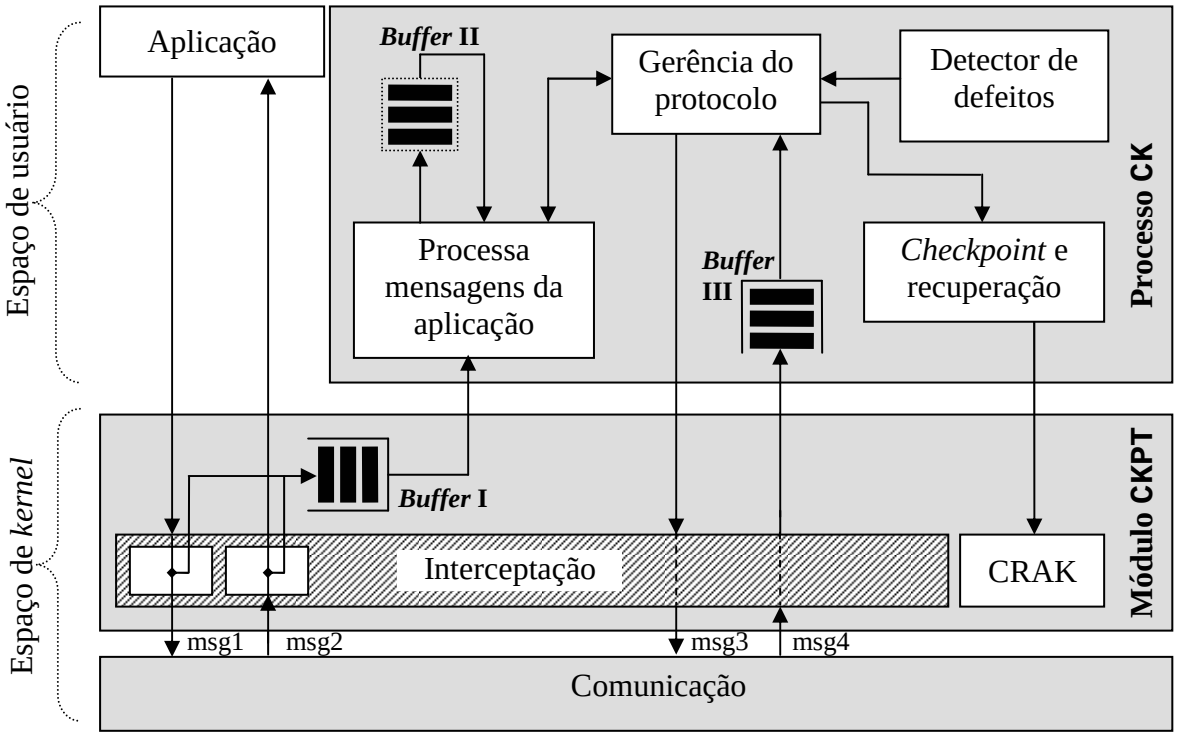


Figura 3.4: Arquitetura do sistema de *checkpointing* em um nodo do sistema.

Como abordado anteriormente, a arquitetura do sistema foi dividida em dois blocos de acordo com os dois diferentes espaços de programação implementados em Linux: espaço de *kernel* e espaço de usuário. A partir dessa divisão, os componentes que fazem parte do sistema de *checkpointing* foram implementados de acordo com a sua função. Nesse sentido, os componentes passivos (aqueles que fornecem algum tipo de serviço) foram implementados em espaço de *kernel* e os componentes ativos (aqueles que executam os serviços dos componentes passivos e gerenciam o funcionamento do algoritmo) foram implementados em espaço de usuário. Além desses componentes, fazem parte da arquitetura implementada três *buffers* (*Buffer* I, *Buffer* II e *Buffer* III ilustrados na Figura 3.4) utilizados pelos componentes para o armazenamento temporário de mensagens da aplicação e mensagens de controle. Desses, o *Buffer* I e III foram implementados em

formato de fila de mensagens. A necessidade desses *buffers* será justificada na medida em que o funcionamento dos componentes for sendo apresentado.

De acordo com a divisão dos blocos, a distribuição dos componentes foi implementada como segue:

1. Componentes passivos (reunidos no modulo **CKPT** em espaço de *kernel*)
    a. *Checkpoint* e recuperação do estado básico de processos (CRAK);
    b. Interceptação de mensagens da aplicação.
2. Componentes ativos (reunidos no processo **CK** em espaço de usuário)
    a. Processa mensagens da aplicação;
    b. Gerência dos *checkpoints* e da recuperação dos processos;
    c. Detector de defeitos;
    d. Gerência do protocolo.

Além desses componentes é ilustrado na figura, no espaço de usuário, o componente "**Aplicação"**, representando o processo da aplicação distribuída.

No bloco dos **componentes passivos** foram incorporados os recursos da biblioteca CRAK para implementar exclusivamente o salvamento e recuperação do estado básico dos processos. Pela característica passiva desse componente, suas funções deverão ser acionadas para serem executadas e salvarem o estado de um processo ou recuperarem o estado do mesmo.

A partir do código-fonte da biblioteca CRAK, foram implementadas funções para interceptação de chamadas de sistema (*system calls*), as quais formam o serviço de **interceptação** das mensagens da aplicação. Esse serviço é acionado a cada vez que uma mensagem for enviada ou recebida pela aplicação. No caso de interceptação de mensagem enviada pela aplicação, o serviço insere informações de controle (identificador da mensagem e da linha de recuperação) no corpo da mensagem e, antes de devolvê-la ao *kernel* para que seja enviada, insere uma cópia desta no *Buffer* I. Por outro lado, ao interceptar uma mensagem recebida e antes de devolvê-la ao *kernel* para que seja entregue para a aplicação, o serviço remove as informações de controle inseridas na transmissão e insere uma mensagem de controle no *Buffer* I com as informações extraídas do corpo da mensagem da aplicação. Nesse momento, o serviço também verifica, através da comparação do identificador da linha de recuperação interceptada com o seu identificador local, a necessidade de estabelecimento de um ***checkpoint* forçado**. Caso afirmativo, o serviço "congela" o processo da aplicação através de *signals* para que ele não receba nenhuma mensagem até que o seu *checkpoint* seja estabelecido, procedimento este que será executado pelos componentes ativos, a ser explicado posteriormente.

No bloco de **componentes ativos** foram reunidos os demais componentes que implementam a execução do algoritmo, fazendo uso dos serviços dos componentes passivos. Encabeça esse grupo o componente **Processa mensagens da aplicação**, cuja função principal é manter atualizada a lista de mensagens da aplicação caracterizadas como "em trânsito" na especificação do algoritmo. Associado a essa tarefa, esse componente deve também indicar a necessidade de estabelecimento de um ***checkpoint* forçado**.

A manutenção da lista de mensagens da aplicação envolve o consumo do *Buffer* I, armazenando as mensagens caracterizadas como “em trânsito” (mensagens enviadas pela aplicação) no *Buffer* II e sinalizando aos emissores o recebimento das mensagens indicadas como recebidas no *Buffer* I. Essa sinalização é necessária para descartar do *Buffer* II as mensagens que foram recebidas pela aplicação, ou seja, não estão mais “em trânsito”. O conteúdo do *Buffer* II é salvo em memória estável no estabelecimento dos *checkpoints* locais. Além disso, como mecanismo auxiliar, também são encaminhadas nesse componente as retransmissões de mensagens enviadas pela aplicação cuja sinalização de recebimento não foi recebida em um determinado prazo.

No que diz respeito ao estabelecimento de um ***checkpoint* forçado**, esse componente detecta, através da interpretação das informações de controle presentes nas mensagens consumidas do *Buffer* I, que o processo da aplicação foi “congelado” e está aguardando o estabelecimento de um *checkpoint*. A ação tomada nesse sentido envolve o acionamento do componente responsável por essa tarefa.

O componente **Gerência dos *checkpoints* e da recuperação dos processos** é responsável pelo estabelecimento dos *checkpoints* locais, formados pelo estado básico do processo da aplicação, obtido através da biblioteca CRAK, e pelas mensagens “em trânsito”, armazenadas no *Buffer* II. Ele também é responsável pela operação inversa, ou seja, pela recuperação do estado básico de um processo e pela recuperação e retransmissão das mensagens a ele associadas.

O componente **Detector de defeitos** é responsável pelo monitoramento do sistema de *checkpoint* e dos processos da aplicação. Detalhes de implementação e funcionamento desse componente serão apresentados na Subseção 4.6.3.

Por último, responsável pelo funcionamento em conjunto de todos os componentes acima mencionados e pela execução do algoritmo, destaca-se o componente **Gerência do protocolo**. Nesse bloco são processadas as mensagens de controle do algoritmo, gerenciando o estabelecimento de novas linhas de recuperação, a coleta de lixo, a rotação de coordenadores e procedimentos de recuperação propriamente dita. Esse componente pode ter algumas funções acionadas pelos demais componentes como pode também acionar funções de outros componentes. Por exemplo: é através dele que o componente **Processa mensagens da aplicação** efetua a retransmissão de mensagens da aplicação e é por ele que chegam as mensagens de controle sinalizando o recebimento das mensagens enviadas pela aplicação. Por outro lado, ele pode acionar o componente responsável pelo estabelecimento dos *checkpoints* quando uma nova linha de recuperação for requisitada.

Todas as mensagens de controle do algoritmo passam pelo componente Gerência do protocolo. As mensagens de controle recebidas, antes de serem processadas, são armazenadas no *Buffer* III. A justificativa para inserção desse *buffer* é para simplificar a implementação. Ao invés de implementar o recebimento das diferentes mensagens de controle em diferentes funções, optou-se por centralizar o recebimento em um único ponto e, a partir desse ponto, acionar as funções correspondentes a cada mensagem de controle recebida. Essa alternativa facilita o uso de *threads*, que processam em paralelo.

3.4.1 Demonstração das situações

Nesse ponto, já é possível demonstrar como os componentes do sistema de *checkpointing* implementam em conjunto as situações de tomada dos *checkpoints* indicadas na Subseção 3.3.1.

A primeira delas refere-se ao estabelecimento de ***checkpoints* requisitados**. Nessa situação, o componente **Gerência do protocolo** do processo coordenador envia mensagens de controle específicas requisitando o estabelecimento de uma linha de recuperação. Ao receber essa mensagem, o mesmo componente nos processos participantes aciona o componente responsável pelo estabelecimento do *checkpoint* local. Este por sua vez, efetua o "congelamento" do processo da aplicação, o salvamento das mensagens e do estado básico do processo da aplicação e, após "descongelar" o processo da aplicação, aciona o componente Gerência do protocolo para que seja enviada a correspondente resposta ao coordenador.

As fases subseqüentes ao estabelecimento de uma linha de recuperação ficam concentradas no componente Gerência do protocolo, assim como as tarefas complementares, como a coleta de lixo e a rotação de coordenadores.

No caso do estabelecimento de um ***checkpoint* forçado**, a aplicação é "congelada" pelo componente de interceptação antes de receber a mensagem que determinou o *checkpoint*. Ao consumir o *Buffer* I, o componente **Processa mensagens da aplicação** identifica a necessidade de *checkpoint* e imediatamente aciona o componente **Gerência do Protocolo**, que por sua vez aciona o componente responsável pelo estabelecimento do *checkpoint* da mesma forma realizada no caso de tomada de *checkpoint* requisitado, com a diferença que o processo da aplicação já esta "congelado". Ao ser "descongelado", o processo da aplicação continua seu processamento, recebendo a mensagem que acionou o estabelecimento do seu *checkpoint*. Enquanto o processo da aplicação estiver "congelado", todas as mensagens destinadas a ele ficarão armazenadas nos *buffers* da camada de comunicação do sistema operacional, aguardando para serem lidas pelo *kernel* e entregues para o processo, o que somente será realizado quando o processo da aplicação for "descongelado".

3.5 Seqüência de Execução

Outra exigência do presente trabalho refere-se à questão de facilidade de uso do sistema de *checkpointing*. Não é um requisito essencial por não se tratar de um trabalho cujo objetivo seja o desenvolvimento de um produto. Mesmo assim, desenvolveu-se uma solução buscando representar uma seqüência de execução do sistema de maneira a simplificar o acesso pelo usuário.

Idealmente, segundo Urbán, Défago e Schiper (2001), a carga de uma aplicação distribuída deveria possibilitar a inicialização, ao mesmo tempo, de todos os processos que a compõem, em todos os computadores do sistema. O usuário poderia executar o processo mestre em um nodo do sistema e, automaticamente, o sistema deveria executar os processos escravos nos demais nodos, instantaneamente. Em Linux, *scripts* e ferramentas de execução remota, como *rsh*, podem ser usados para facilitar a carga dos processos escravos. Entretanto, não são confiáveis e geralmente atrasam a inicialização da aplicação. No presente trabalho, não houve preocupação com a inicialização ideal da aplicação. Essa tarefa não está relacionada com os objetivos do trabalho e fica a critério do usuário. A preocupação foi direcionada para questões diretamente relacionadas com a consistência do algoritmo.

Uma primeira questão envolvendo a consistência refere-se ao estado inicial. Os requisitos de consistência do algoritmo determinam que as mensagens geradas pela aplicação devem ser consideradas durante o estabelecimento de linhas de recuperação

consistentes. Isso implica que deve existir um estado inicial em que a aplicação não gerou nenhuma mensagem no sistema. Mas o que, na prática, é um estado inicial?

De forma geral, na abordagem teórica, o estado inicial corresponde ao estado onde nenhuma computação foi efetuada, como ponto de partida da execução da aplicação. A partir desse estado e com o cumprimento de algumas condições, a execução da aplicação passa para o estado seguinte, onde já está realizando suas atividades-fins e gerando eventos no sistema tais como troca de mensagens. Em termos práticos, essa trivialidade pode não ocorrer. Existem algumas preocupações a serem levadas em consideração para aproximar a inicialização de uma aplicação do modelo teórico. Questões como: quem inicia a aplicação, como ela será disparada em diferentes nodos, qual será o estado inicial, quais serão os estados intermediários, como a aplicação termina e como sua terminação será detectada (RAYNAL, 1992), e como os mecanismos de tolerância a falhas estarão presentes no sistema possibilitam diferentes implementações, de acordo com os requisitos desejados. A partir do ambiente de implementação escolhido e nas alternativas presentes em Linux que possibilitam a inicialização de uma aplicação em ambiente distribuído, propôs-se uma seqüência de passos de execução, representados na Figura 3.5, como possibilidade para a inicialização pelo usuário da aplicação conjugada com o sistema de *checkpointing*.

Em condições convencionais, sem o uso do sistema de *checkpointing*, a aplicação distribuída deve ser disparada em todos os nodos ou computadores sobre os quais a aplicação vai ser executada. Com o uso do sistema proposto, o usuário continuará disparando sua aplicação da mesma forma, exceto pelo fato de ter que informar antecipadamente a sua intenção de executar a aplicação com recursos de *checkpointing* e recuperação.

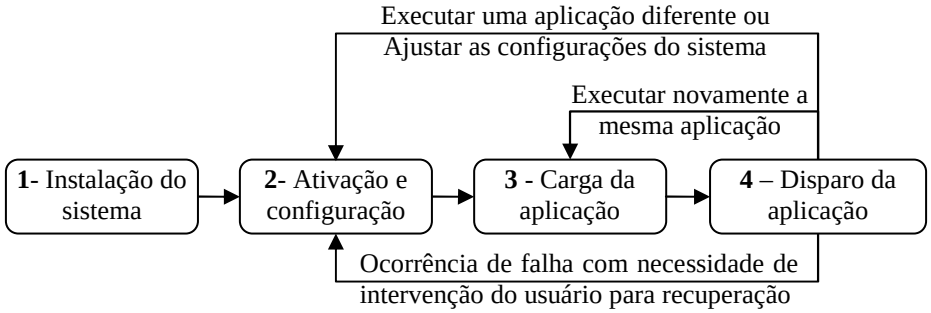


Figura 3.5: Passos para o usuário executar as aplicações.

A seqüência de execução proposta é composta por 4 etapas: (1) instalação do sistema de *checkpointing*; (2) ativação do sistema informando qual aplicação deverá ser gerenciada; (3) carga da aplicação propriamente dita e, por fim, (4) disparo da aplicação. Além destas quatro etapas, em caso de manifestação de erro que impeça a recuperação automática do processamento, conforme explicado na seção anterior, o usuário terá que intervir para disparar novamente o sistema (passo 2). Nesse caso, o sistema irá detectar a existência de uma linha de recuperação e assumirá automaticamente as etapas 3 e 4, com a diferença de que a aplicação continuará sua execução a partir do estado salvo na linha de recuperação correspondente.

Destas quatro etapas, somente a primeira (1) exige que o usuário possua privilégios administrativos no sistema operacional. Nessa etapa, será instalado o módulo **CKPT** e, por tratar-se de inserção de um módulo de *kernel*, o sistema operacional exige privilégios para tal. A contrapartida é que essa etapa será efetuada uma única vez, ficando o módulo **CKPT** disponível para todas as futuras utilizações, por todos os usuários do sistema. A instalação do processo **CK** é trivial, bastando copiar seu arquivo executável para algum diretório do sistema e dar permissão de execução nesse arquivo para todos os usuários.

A segunda etapa envolve a execução do processo **CK**. É através desse processo que o usuário informará a identidade da sua aplicação. Por questões de implementação, definiu-se que, sob o ponto de vista do usuário, a identificação da aplicação corresponde ao nome do seu arquivo executável. Dessa forma, um dos parâmetros informados na carga do processo **CK** poderá ser a identificação da aplicação. Outros parâmetros correspondem ao intervalo entre *checkpoints*, aos prazos estabelecidos para *timeouts*, ambos em segundos, e *flags* indicando ações adicionais a serem executadas pelo sistema. Nessa etapa, o usuário ainda terá duas alternativas: configurar individualmente cada nodo, através de parâmetros na execução do processo **CK**, ou centralizar a configuração através de uma ferramenta desenvolvida exclusivamente para configurar e monitorar o sistema de *checkpointing*, a partir de um único ponto.

Essa ferramenta, denominada de processo **MONITOR**, é acessória ao sistema e foi desenvolvida com propósitos de centralizar e facilitar o controle e monitoração do sistema de *checkpointing*. Ela não exige nenhuma configuração. Ao ser executada, ela detectará automaticamente todos os processos **CK** na rede e exibirá informações sobre cada um. Além de monitorar as informações, é possível inserir, através desse processo, mensagens de controle que causam algumas ações no sistema. Por exemplo, através dela é possível interromper, a qualquer momento, o processamento da aplicação. Da mesma forma, é possível reiniciar o processamento a partir dessa interrupção ou a partir da última linha de recuperação consistente. Estas funcionalidades foram inicialmente implementadas com propósitos de testes do sistema, entretanto observou-se que poderiam ser úteis para os usuários. Maiores detalhes do processo **MONITOR** serão apresentados no Capítulo 4.

Uma vez que o sistema de *checkpointing* esteja instalado e configurado, o usuário pode executar sua aplicação (etapa 3). Entretanto, diferentemente do comportamento normal da aplicação em execuções sem o sistema de *checkpointing*, a aplicação será imediatamente colocada em estado de espera, através da mensagem especial `SIGSTOP` enviada pelo módulo **CKPT** no momento em que o processo da aplicação for carregado para a memória. O mesmo acontecerá com todos os processos da aplicação, em todos os nodos em que ela for executada. Quando o usuário disparar o último processo da aplicação, o sistema de *checkpointing* detectará este fato e a aplicação estará pronta para ser disparada. Além disso, estará configurado o estado inicial, onde a aplicação está carregada na memória mas não executou nenhuma ação.

A partir desse ponto, caso o usuário tenha informado ao processo **CK**, através de parâmetros, que caberia ao sistema disparar automaticamente a aplicação, é enviada para todos os processos componentes uma mensagem especial `SIGCONT`, fazendo com que a aplicação comece de fato sua execução. Caso contrário, se o usuário optou por disparo manual, a aplicação ficará em estado de espera até receber o comando para iniciar a execução. Esse comando poderá ser inserido pelo usuário através do processo **MONITOR**,

em qualquer um dos nodos do sistema. Essa é a última etapa da seqüência de execução, que pode ou não ser efetuada pelo usuário, e corresponde à etapa 4 na Figura 3.4.

Após a aplicação ter iniciado o processamento, e se não houver falhas, não será necessária a interferência do usuário, salvo se a aplicação exigir. Já em casos de ocorrência de falhas, e se o sistema não conseguir recuperar-se automaticamente, haverá a necessidade de intervenção do usuário para solucionar o problema e disparar novamente o sistema.

Quando a aplicação terminar sua execução, o sistema detectará esse término e também encerrará sua execução. Entretanto, o processo **`CK`** continuará ativo na memória, aguardando que uma nova aplicação seja executada. O usuário pode executar a mesma aplicação novamente ou, através do processo **`MONITOR`**, informar aos processos **`CK`** a identidade da nova aplicação e então executá-la. Opcionalmente o usuário pode forçar a finalização dos processos **`CK`** e, se desejar, executar novamente as etapas 2 a 4.

Sob o ponto de vista do usuário, as etapas citadas anteriormente resumem a seqüência de execução de suas aplicações quanto optar pelo uso conjugado com o sistema de *checkpointing* para torná-las tolerantes à falhas. Com base na prática de uso dos sistemas operacionais atuais, entende-se que a maioria das aplicações é executada pelos usuários na seqüência apresentada na Figura 3.4. Podem existir situações, por exemplo, em que a aplicação faça parte do sistema e já esteja carregada na memória. Nesse caso, não existiria o passo 3 e o usuário teria de informar ao sistema os identificadores de cada processo (*PIDs*) antes de disparar sua aplicação.

3.6 Ciclo de Vida do Sistema

Na seção anterior, foram discutidas as etapas necessárias para utilização do sistema, sob o ponto de vista do usuário. Nesta seção, serão discutidos os mecanismos implementados e que são executados durante aquelas etapas. Ao conjunto de ações executado pelo sistema, desde sua instalação até sua remoção, deu-se o nome de ciclo de vida do sistema (segundo a nomenclatura definida por Pressman (1995)), representado na Figura 3.6.

A primeira etapa refere-se à instalação do sistema. Por estar dividida em duas partes, uma em um módulo de *kernel* e outra em um processo de usuário, a instalação basicamente resume-se à inserção da primeira parte no sistema operacional. Pela característica de módulos de *kernel* (RUBINI, 1999), são necessários somente privilégios administrativos para o usuário encarregado dessa atividade. Uma vez inserido no sistema, o módulo passa a interceptar todas as chamadas de sistema `sys_socketcall` geradas pelos processos de usuário. Mas, como não fará nenhum processamento, simplesmente repassando as mensagens adiante, a interferência no sistema é mínima (FONTOURA, 2002). Da mesma forma, a remoção desse módulo também deve ser realizada por usuários administradores.

A inserção ou remoção do módulo de *kernel* pode ser realizada sem necessidade de recarregar o sistema operacional. Aplicações que geram as chamadas de sistema interceptadas pelo módulo podem estar em execução durante estas operações.

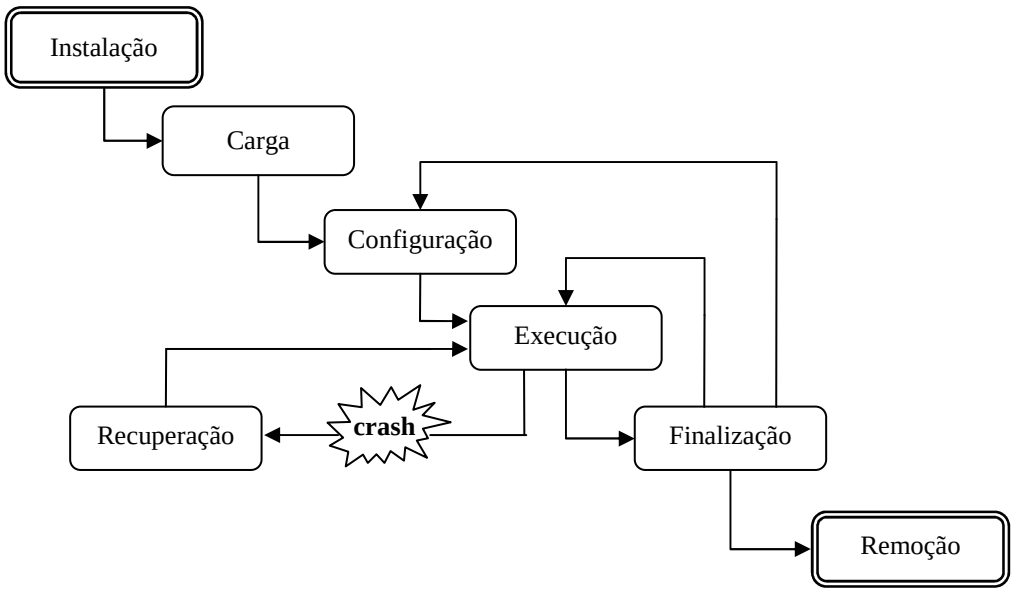


Figura 3.6: Ciclo de vida do sistema de *checkpointing*.

Uma vez inserido no sistema, o módulo de *kernel* estará pronto para ser utilizado pelo processo **CK**. O processo **CK**, por sua vez, somente funcionará com a presença do módulo de *kernel*, pois é através dele que as ações iniciais do sistema são executadas. Como explicado nas seções anteriores, a instalação do processo **CK** resume-se em copiá-lo para o disco e configurar as permissões adequadas, de modo que os usuários possam executá-lo. O mesmo vale para o processo **MONITOR**, que poderá, opcionalmente, ser utilizado pelos usuários.

É através do processo **CK** que o usuário irá configurar e solicitar que sua aplicação seja executada com recursos de tolerância a falhas. Essa configuração pode ser definida durante a execução do processo **CK** ou posteriormente através do processo **MONITOR**, mas sempre antes de executar a aplicação propriamente dita.

Durante a inicialização do processo **CK**, a primeira ação será verificar a presença do módulo **CKPT** no sistema. Caso não esteja presente, o usuário será notificado da necessidade de sua instalação no sistema operacional. Em seguida, ele informará ao módulo de *kernel* a identidade da aplicação fornecida pelo usuário. É a partir dessa identificação que o módulo de *kernel* irá interceptar a carga e finalização da aplicação. Quando o usuário iniciar a aplicação, será gerada a chamada de sistema `sys_execve` contendo o nome do executável, o qual será interceptado pelo módulo **CKPT**. Nesse momento, o sistema operacional já terá atribuído um identificador (*PID*) para o processo da aplicação, o qual será identificado pelo módulo **CKPT**. Também nesse momento, a aplicação será colocada em estado de espera, conforme abordado anteriormente.

Uma vez de posse do identificador do processo, este será informado ao processo **CK** pelo módulo **CKPT** o qual passará a interceptar e processar todas as chamadas de sistema `sys_socketcall` geradas por aquele processo, assim que for desbloqueado pelo processo **CK**.

Enquanto o módulo **CKPT** não identificar a aplicação (período em que o usuário estiver disparando os processos da sua aplicação nos nodos do sistema), o processo **CK** permanecerá verificando a presença de outros processos **CK**, em execução no demais nodos do sistema, através de mensagens específicas. O objetivo é estabelecer o grupo de processos **CK** do sistema. Na medida em que o usuário for executando os processos **CK** nos nodos da rede, o grupo vai se formando e, durante essa etapa, todos os processos **CK** trocam mensagens entre si para entrarem em acordo sobre quem será o coordenador inicial. Por definição, e também por sugestão do autor do algoritmo (CECHIN, 2002), escolhe-se inicialmente como coordenador o processo que possuir o menor identificador.

Levando-se em consideração que a experimentação realizada no presente trabalho, a ser apresentada no Capítulo 5, restringiu-se à utilização de equipamentos de uma mesma sub-rede, onde todos os *hosts* devem possuir o mesmo número de rede (TANENBAUM, 1996), definiu-se o endereço IP dos nodos da rede como identificador do processo **CK** no grupo. Portanto, será coordenador da primeira rodada o processo cujo IP for menor. Essa decisão será difundida através de mensagens específicas e todos deverão concordar, uma vez que a definição do grupo é requisito para o próximo passo do algoritmo: início da execução da aplicação.

Quando o usuário tiver executado a aplicação no último nodo da rede em que tenha previamente executado o processo **CK** e após todos os processos **CK** terem chegado ao acordo sobre o coordenador, o sistema como um todo estará pronto para desbloquear a aplicação e iniciar o processamento especificado e os procedimentos de *checkpointing*. O usuário pode ter optado por desbloquear a aplicação manualmente através de comando via processo **MONITOR**, o que será feito neste momento. Se essa opção não tiver ocorrido, o próprio processo coordenador dará a ordem para que todos os processos **CK** desbloqueiem localmente os processos da aplicação.

A partir de então, iniciam-se os procedimentos do algoritmo: processamento das mensagens interceptadas pelo processo da aplicação, *checkpoint* em intervalos determinados pelo usuário, recuperação em caso de erros, rotação de coordenadores, coleta de lixo e detecção de defeitos. Exceto a recuperação propriamente dita, os demais procedimentos citados anteriormente serão executados enquanto a aplicação estiver executando ou enquanto não ocorrerem falhas, sendo interrompidos quando os módulos **CKPT** detectarem que a aplicação terminou, situação em que a aplicação gera uma chamada de sistema específica, também interceptada pelo sistema.

Em caso de ocorrência de falhas e da respectiva detecção, o sistema interrompe o processamento da aplicação e aguarda a sinalização para recuperar o processamento a partir da última linha de recuperação previamente estabelecida. Quando cessar o efeito da falha, o sistema poderá, em alguns casos, recuperar-se automaticamente, sem a intervenção do usuário. Em outros casos, onde o próprio sistema de *checkpointing* for afetado pelo efeito da falha, poderá ser necessária a intervenção do usuário para colocar o sistema em execução novamente. É exemplo de uma dessas situações a interrupção de energia em algum nodo, ficando praticamente impossível tomar alguma ação até que o nodo seja religado.

A recuperação para um estado consistente, anterior ao da ocorrência da falha, somente será efetuado quando todos os processos **CK** voltarem a operar e após chegarem a um acordo sobre o restabelecimento do sistema. Isso será realizado mediante a troca de mensagens entre os processos **CK**, da mesma forma em que decidem, durante a carga do

sistema, quem será o coordenador. A diferença é que nessa etapa todos os processos **CK** sabem quem eram os demais componentes do grupo e somente admitirão que a falha foi extinta quando todos os demais membros do grupo responderem positivamente às requisições. Caso a identidade do coordenador for perdida durante a falha, o grupo escolherá novamente o novo processo coordenador.

Uma vez restabelecido o grupo e decidida a recuperação, caberá ao coordenador a tarefa de ordenar o retorno para a última linha de recuperação consistente. Todo processo **CK** procederá ao retorno para o último *checkpoint* válido, inclusive efetuando a retransmissão de todas as mensagens armazenadas nesse *checkpoint* (registradas como enviadas mas não registradas como recebidas). A partir de então, a aplicação continua o processamento desde este último estado consistente, até seu término ou até a manifestação de outra falha. Os mecanismos implementados para recuperação serão explicados em detalhas no Capítulo 4.

Por fim, o último estágio do ciclo de vida do sistema de *checkpointing* inicia a partir do momento em que a aplicação encerra suas atividades. Essa transição é detectada pelo sistema através da interceptação da chamada de sistema `sys_exit`, a qual é gerada quando um processo encerra em condições normais de execução.

Nesse estágio, todo o sistema de *checkpointing* (módulo **CKPT** e processo **CK**) permanece ativo aguardando novas ações do usuário, o qual pode decidir por encerrar o sistema ou executar novas aplicações. Através do processo **MONITOR**, o usuário pode enviar comandos para informar ao sistema a identidade da nova aplicação a ser executada. Da mesma forma, poderá solicitar que o sistema seja encerrado, situação em que, através de um comando centralizado no processo **MONITOR**, todos os processos **CK** são finalizados. Os módulos **CKPT** permanecem ativos, mas não processam nenhuma mensagem interceptada.

A remoção completa do sistema dar-se-á pela remoção do módulo **CKPT** do sistema. Novamente, essa atividade deve ser executada por usuários com privilégios administrativos e sem a necessidade de recarga do sistema operacional.

4 IMPLEMENTAÇÃO

A partir do algoritmo e sua especificação, da definição do ambiente de desenvolvimento e do modelo de sistema distribuído, foi possível dar início às atividades de programação do sistema. O objetivo deste capítulo é apresentar a implementação dos mecanismos que compõem a solução adotada para a confecção do sistema de *checkpointing*.

Na medida em que a implementação for apresentada, serão retomadas todas as abordagens propostas nos capítulos anteriores referentes aos mecanismos do algoritmo, bem como restrições impostas pelo ambiente de desenvolvimento e alternativas adotadas para contornar estas restrições. Buscando auxiliar o entendimento das soluções, serão expostos trechos do código-fonte e descrição formal do algoritmo. Informações mais significativas do código-fonte poderão ser encontradas nos anexos e apêndices, ao final desse trabalho, assim como a descrição formal do algoritmo. O código completo e demais instruções de compilação e utilização do sistema fazem parte do CD-ROM que integra este trabalho (Apêndice J).

Inicialmente, será mostrada a estrutura geral do sistema representando os componentes desenvolvidos e a comunicação entre eles. Em seguida, serão detalhados esses componentes e as funções implementadas em cada um deles. Depois, serão apresentados os mecanismos do algoritmo implementados com o uso desses componentes, e o mapeamento da especificação para a implementação, buscando comprovar que a especificação foi obedecida. Por fim, serão citados os problemas observados durante a implementação e possíveis soluções para contorná-los.

4.1 Estrutura Geral

O sistema implementado é formado por um processo de usuário (processo **`CK`**), um módulo de *kernel* (módulo **`CKPT`**) e um agente de monitoramento (processo **`MONITOR`**), todos desenvolvidos na plataforma Linux, exclusivamente em linguagem C/C++. Os dois primeiros, carregados nos nodos da rede, compõem o sistema de *checkpointing* propriamente dito. O último representa unicamente uma alternativa de ferramenta centralizadora da gerência e configuração dos dois primeiros. A Figura 4.1 tenta representar, em uma abordagem didática, os três componentes em sua relação com o ambiente operacional distribuído, bem como a interface de comunicação entre eles. Cada um dos conjuntos representa um nodo processador da rede, ou seja, um computador.

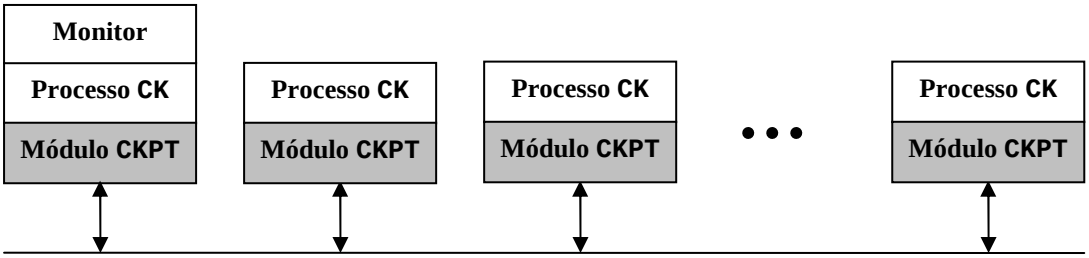


Figura 4.1: Distribuição dos componentes do sistema.

A comunicação local entre o processo **CK** e o módulo **CKPT** é realizada através de uma interface implementada na forma de funções em C/C++ no módulo de *kernel*. Essa comunicação é unidirecional, "de cima para baixo" (esta expressão está associada à representação usada na Figura 4.1), sendo sempre o processo **CK** quem as invoca e o módulo **CKPT** quem as provê, ou executa, devolvendo os resultados com o retorno das chamadas. O módulo **CKPT**, uma vez acionado, gera eventos que poderão interferir na execução do processo **CK**. Entretanto, esse comportamento não caracteriza um tipo de comunicação direta, pois o processo **CK** não disponibiliza funções para o módulo de *kernel* invocá-las. Detalhes dessas funções serão apresentados posteriormente.

Pode-se caracterizar o módulo **CKPT** como agente passivo e o processo **CK** como agente ativo, uma vez que a atividade do primeiro depende de ações geradas pelo segundo e da ocorrência de eventos externos, tais como chamadas de sistema geradas pelas aplicações. O processo **CK** é uma aplicação *multithread*, onde estão implementadas a gerência dos mecanismos de sistema, a geração de eventos, as chamadas de sistema e as chamadas de funções disponíveis no próprio módulo **CKPT**.

A comunicação entre os componentes de diferentes nodos do sistema é realizada exclusivamente através de troca de mensagens. Para tal, são utilizados *sockets UDP* (STONES, 1999), cujas mensagens são encapsuladas em datagramas e o endereçamento de cada nodo é formado pelo endereço IP da estação associado a uma porta específica.

Da mesma forma, o processo **MONITOR** também utiliza mensagens encapsuladas em datagramas para executar suas funções de gerência e monitoração. Sua comunicação é direcionada exclusivamente para os processos **CK**, os quais possuem uma função específica para recebimento e processamento das requisições do processo **MONITOR**.

4.2 Módulo CKPT

O módulo **CKPT** é um módulo de *kernel* em Linux. Módulos de *kernel* executam em espaço de *kernel* e normalmente são utilizados para implementação de *device drivers*, buscando complementar funções do sistema operacional não disponibilizadas por padrão. Conceitualmente, os *device drivers* são divididos em três classes: módulos de caracter, módulos de bloco e módulos de rede. Entretanto, os módulos podem implementar funções destas três classes simultaneamente, limitando-se à criatividade e ao conhecimento do programador sobre programação em espaço de *kernel*. Um módulo de *kernel* pode ser inserido e removido sem a necessidade de interromper o sistema operacional (RUBINI, 1999).

No presente trabalho, foram reunidas em um módulo de *kernel* aquelas funções básicas não disponíveis no sistema operacional na forma exigida pelo algoritmo, tais

como salvamento e recuperação de processos e interceptação de chamadas de sistema. Ambas podem ser implementadas em espaço de usuário, na forma de uma aplicação convencional e utilizando diferentes recursos para garantir sua funcionalidade. A opção por implementar em módulo de *kernel* justifica-se pela possibilidade de uso e adaptação da biblioteca CRAK, apresentada em seguida, economizando esforços de programação e, principalmente, e pela pouca interferência da interceptação de chamadas de sistema em espaço de *kernel*, se comparada com alternativas de interceptação em espaço de usuário (FONTOURA, 2002).

Nesta seção serão apresentadas as funções básicas da biblioteca CRAK e as adaptações nela realizadas para suportar as funcionalidades exigidas pelo algoritmo. As adaptações incluem a inserção da interceptação de chamadas de sistema e modificações nas funções de interface de acesso. Dessa forma, o módulo de *kernel* **CKPT** é composto pelas funções herdadas pela biblioteca CRAK mais as funções implementadas para interceptação de chamadas de sistema.

4.2.1 Biblioteca CRAK

Conforme discutido no Capítulo 2, existe uma variedade de alternativas para implementação de salvamento e recuperação de estados de processos. As alternativas vão desde a implementação completa de todos os mecanismos buscando essa funcionalidade, sem o uso de bibliotecas auxiliares, passando pela utilização de infra-estrutura especializada e ambientes simulados, chegando a alternativas de bibliotecas disponíveis para plataformas convencionais, tais como Linux, onde são utilizados os recursos básicos do sistema operacional para programar os mecanismos necessários.

A biblioteca CRAK enquadra-se nessa última abordagem, uma vez que utiliza exclusivamente mecanismos disponíveis na plataforma Linux. É uma biblioteca implementada em C/C++ e foi proposta com o propósito de prover migração de processos[3]. O objetivo principal dos autores era o salvamento do estado de uma aplicação residente em um computador para ser posteriormente restaurada no mesmo computador ou em computadores diferentes, mas não estava voltada a aplicações distribuídas.

Originalmente, a biblioteca é composta por um módulo de *kernel* e um conjunto de processos de usuário. No módulo de *kernel*, estão implementadas as funções básicas de salvamento (ilustrada pela Figura 4.2) e recuperação (ilustrada pela Figura 4.3) de um processo. A função de salvamento recebe três parâmetros: descritor de um arquivo, identificador (PID) do processo da aplicação e um conjunto de *flags*, armazenados em uma variável de tipo `int`, contendo informações sobre o modo de salvamento.

```
static int checkpoint(int fd, pid_t pid, int flags) {...}
```

Figura 4.2: Função da biblioteca para salvamento de estados dos processos.

Note-se que o primeiro argumento é um ponteiro para um arquivo em Linux e não o nome do arquivo e seu caminho. Esse arquivo deve ser criado antes da chamada da função e deve estar acessível. A biblioteca parte desse pressuposto e salva o estado do processo diretamente no arquivo apontado por esse ponteiro. Ao contrário do que se pode imaginar, essa não é uma restrição imposta e sim uma característica interessante que pode estender o salvamento de estados em diferentes alternativas. Em Linux, recursos são

[3] Foi desenvolvida na Universidade de Columbia em torno de 2001 por H. Zhong e J. Nieh.

manipulados como arquivos (dispositivos de E/S, conexões de rede, módulos de *kernel* que implementam dispositivos de bloco, etc.). Assim, pode-se, ao invés de passar como parâmetro um ponteiro para um arquivo localizado no disco local, passar um ponteiro para uma conexão de rede, possibilitando que o estado do processo seja salvo em um computador remoto via *sockets*. Esta seria uma boa alternativa para implementar um serviço alternativo de memória estável para salvamento dos *checkpoints*, conforme sugerido por Jiménez e Rajsbaum (2003).

O segundo argumento representa o identificador do processo a ser salvo. Caso a aplicação seja *multithread*, ela será composta por mais de um processo (do ponto de vista da biblioteca) e a função `checkpoint()` deverá ser acionada repetidas vezes, tantas quantas forem as *threads* que compõem a aplicação, para salvar o estado de cada uma delas. Dessa forma, cada *thread* da aplicação terá sua imagem específica e a união dessas imagens formará a imagem do estado da aplicação no nodo, ou computador, em que estiver executando. Esse é um dos pontos fracos da biblioteca devidos, provavelmente, às restrições impostas pelo sistema operacional e motivou a restrição de que a aplicação seria composta por um único processo em cada nodo.

Mas o que exatamente é salvo? Quais informações são necessárias para restaurar uma aplicação a partir de um estado intermediário? Como estas informações são salvas e como são recuperadas? A resposta para estas perguntas depende da aplicação e dos recursos alocados por ela durante a sua execução. O estado de um processo em execução inclui a imagem do processo, o conjunto dos registradores e os arquivos abertos. Além disso, dependendo da aplicação, existe uma série de outras informações, tais como diretório corrente, estado de conexões de rede, identidade do usuário (UID, GUID, etc), *signal handlers*, *System V IPC*, estado dos terminais, etc (BECK *et al.*, 1999).

A imagem do processo corresponde ao espaço de memória reservado para sua execução no sistema. A biblioteca `CRAK` copia e armazena em disco o espaço de memória do processo. O espaço de memória é composto por seções, chamadas de áreas de memória virtual (*Virtual Memory Areas - VMAs*). Cada VMA é um bloco contínuo de memória, com endereço de início e fim e atributo de proteção (`read/write/execute` e `private/share`). Uma VMA com atributo “`rwp`”, por exemplo, pode ser lida, escrita e é de uso exclusivo do processo ao qual pertence.

Quando é chamada, a biblioteca percorre todas as VMAs do processo, salvando no arquivo de imagem as suas posições, seus atributos e os seus conteúdos. Quando o processo é recuperado, essas VMAs são recuperadas através da função `mmap` (BECK *et al*, 1999).

A biblioteca não implementa salvamento incremental das VMAs, o que seria uma alternativa para reduzir a sobrecarga, em caso de realização de salvamentos consecutivos do mesmo processo (ELNOZAHY; ZWAENEPOEL, 1992a). Dessa forma, seriam salvas somente aquelas VMAs que foram alteradas a partir do último *checkpoint*. Entretanto, a biblioteca possibilita ignorar o salvamento de segmentos de código, onde é armazenado o código executável do processo, e segmentos de memória compartilhada com atributo de leitura. Nessa última área de memória estão alocadas as bibliotecas compartilhadas, comumente utilizadas pelas aplicações. Em caso de recuperação, estas áreas podem ser obtidas diretamente dos arquivos executáveis e dos arquivos das bibliotecas utilizadas pela aplicação, o que não prejudica o desempenho da recuperação.

Ao recuperar um processo, é necessário garantir que todos os registradores assumam os mesmos conteúdos do momento em que foram salvos. A biblioteca garante essa operação, salvando o conteúdo de todos os registradores e, durante a recuperação, atualizando-os na pilha corrente.

O salvamento das informações dos arquivos é relativamente simples. A dificuldade está na recuperação destas informações. Existem basicamente três tipos de arquivos em Linux: arquivos em disco, *pipes* e *sockets*. Eles possuem diferentes formas de identificação. Arquivos em disco são identificados por nome e caminho (*path*) do sistema de arquivos. *Pipes* são anônimos e diretamente identificados pelo número de seus nodos. A identificação de um *socket* depende do protocolo, sendo normalmente formado pelo endereço de rede e porta. É necessário, portanto, salvar a identidade de cada arquivo baseado em seu tipo. Em todos os casos, deve-se salvar os descritores dos arquivos de maneira a garantir que, quando o processo for recuperado, continuem a referenciar os arquivos originais. Portanto, esses arquivos devem existir no momento da recuperação. Caso contrário, não será possível retornar para um estado consistente.

Essa situação ocorre se, no momento da recuperação de um determinado arquivo em uso pelo processo no momento do salvamento de seu estado, ela não estiver mais presente, por ter sido excluído ou movido para outro diretório. Uma alternativa para resolver esse problema seria salvar também o conteúdo do arquivo e, na recuperação, restaurá-lo em seu local original antes de recuperar o seu descritor.

A biblioteca não trata essa situação e parte do pressuposto que, na recuperação, os arquivos que estavam em uso pelo processo e os diretórios em que esses arquivos estavam armazenados não tenham sofrido nenhuma alteração nem exclusão a partir do momento em que o *checkpoint* recuperado tenha sido estabelecido. Por questões de simplificação na programação do sistema de *checkpointing*, foi mantida a forma original de tratamento de arquivos abertos, implementada pela biblioteca. Entretanto, para uma solução isenta de inconsistências relacionadas aos arquivos, entende-se ser necessário salvar também o conteúdo dos arquivos em uso pela aplicação, pois os mesmos podem ter sofrido alteração, principalmente pela própria aplicação, após o estabelecimento dos *checkpoints* e, na recuperação, os seus conteúdos devem corresponder exatamente aos conteúdos dos arquivos referenciados nos respectivos *checkpoints*.

Dessa forma, a recuperação de arquivos em uso pela aplicação resume-se em usar o mesmo descritor e mover o ponteiro do arquivo para a posição original. Essa operação é realizada através da função `dup2`, a qual duplica o descritor do arquivo e ambos, o velho e novo descritor, referem-se exatamente ao mesmo arquivo. A biblioteca abre o arquivo com um novo descritor e força, através da função `dup2`, que o descritor antigo, usado pelo processo em recuperação, assuma o mesmo estado desse novo descritor. Esse mecanismo é executado para cada descritor de arquivos do processo, não havendo possibilidade de conflito de identificadores, uma vez que são únicos.

A recuperação de *pipes* é mais complexa, pois envolve um grupo de processos. A complexidade reside no fato de que a existência de *pipes* indica que há um grupo de processos comunicando-se através desse mecanismo. Se dois processos estão trocando informações via *pipe*, é necessário salvar e recuperar o estado dos dois processos simultaneamente; caso contrário o *pipe* poderá ser interrompido pelo sistema operacional após sua recuperação.

O autor da biblioteca sugere a resolução desse problema através da detecção e salvamento sincronizado do grupo de processos. Antes de salvar o estado de um processo

do grupo, envia-se uma mensagem especial `SIGSTOP`. Dessa forma, todos os processos são “congelados” e então salvos individualmente.

Na recuperação, os processos serão “descongelados” através da mensagem especial `SIGCONT`, enviada depois que todos os descritores dos *pipes* forem reabertos, da mesma forma como usada com os descritores de arquivos em disco. Quando os processos forem “descongelados”, os dois lados do *pipe* estarão prontos e não haverá quebra de *pipe* pelo sistema operacional. Essa sugestão não é implementada pelo módulo de *kernel* e exige implementação adicional em espaço de usuário.

Soluções alternativas para o salvamento e recuperação de conexões de rede também são sugeridas pelo autor da biblioteca. Diferente de *pipes*, conexões de rede envolvem não somente dois processos diferentes, mas dois computadores diferentes. Obviamente, em ambiente Linux, essa operação é bastante complexa, pois exige mecanismos de sincronização entre ambos os lados de uma conexão, tanto para o salvamento quanto para a recuperação de uma conexão de rede.

A biblioteca implementa um protótipo de migração de conexões *sockets TCP/IPv4*, com funcionalidade restrita a certas aplicações. O protótipo envolve o uso de ferramentas de execução remota, como `rsh`, salvamento e recuperação da pilha TCP das estações envolvidas na conexão de rede do processo, além de uma série de procedimentos para tentar garantir que, ao recuperar uma aplicação, ela não “perceba” que, por exemplo, seu endereço de rede não é mais o mesmo.

O próprio autor admite que esse protótipo pode não funcionar e sugere que o problema seja passado para a aplicação. Testes realizados com a biblioteca não forneceram resultados positivos na recuperação de aplicações que mantêm conexões de rede abertas durante o processamento.

A recuperação de processos utilizando a biblioteca `CRAK` é realizada através da chamada da função `restart()`, ilustrada na Figura 4.3. Essa função é implementada no módulo de *kernel* e é acionada por processos de usuário.

```
static int restart(char *, pid_t pid, int flags) {...}
```

Figura 4.3: Função da biblioteca para recuperação processos.

Os argumentos passados para essa função são: um ponteiro para o nome do arquivo que contém a imagem do processo previamente salva, um identificador de processo (`pid`) e um conjunto de *flags*, armazenados em uma variável de tipo `int`, contendo informações sobre o modo de recuperação.

O primeiro argumento é trivial: representa o arquivo que contém a imagem do processo salvo através da função `checkpoint()`. O segundo argumento representa certos processos que devem ser notificados após a recuperação. Se for 0 (zero), o processo pai é notificado. Essa notificação resume-se ao envio de mensagem especial `SIGUSR1` para o processo correspondente e representa um aviso de que o processo filho foi recuperado. É essencial em situações em que o processo faz parte de um grupo que deve ser recuperado simultaneamente.

O último argumento é composto pelos *flags* `RESTART_NOTIFY` e `RESTART_STOP`. O primeiro, se presente, causa o envio de uma notificação para o processo representado pelo identificador. O segundo força o processo a permanecer “congelado” após ser

recuperado. Ele poderá ser notificado posteriormente quando seu identificador for passado para a função `restart()` ou quando outra função o fizer.

Diferentemente do salvamento da imagem do processo, a recuperação não pode ser realizada a partir de uma conexão *socket*. Essa restrição deve-se à função `mmap` utilizada para carregar a imagem para a memória, a qual exige arquivos concretos, presentes no sistema de arquivos do próprio computador. Entretanto, como o autor da biblioteca sugere, os arquivos poderiam ser previamente copiados localmente a partir de uma máquina remota e, em seguida, utilizados na recuperação.

É importante ressaltar que a biblioteca não cria um novo processo no momento da recuperação: ela substitui o contexto do processo que chamou a função `restart()` pela imagem do processo salvo previamente. Por isso, é necessário que esse processo seja criado antes da chamada dessa função. Como sugestão do autor da biblioteca, qualquer processo de usuário pode usar a função `fork` e o processo-filho invocar a função `restart()`. É necessário implementar também os devidos ajustes nas informações sobre a hierarquia do processo recuperado, desvinculando-o do processo-pai que chamou a função `fork`.

O autor da biblioteca implementou rotinas-exemplo em processos que tratam as peculiaridades citadas. Esses processos implementam o salvamento e recuperação de aplicações *multithread*, com tratamento de arquivos abertos, *pipes*, e outras informações acessórias, como diretório corrente e estado do terminal.

Dessa forma, entende-se que a utilização das funções da biblioteca deve ser explorada a partir de processos implementados em espaço de usuário. Esses processos acessam as funções implementadas no módulo de *kernel* da biblioteca para executarem operações básicas de *checkpoint* e recuperação, além de funções acessórias para complementar as operações executadas em espaço de *kernel*.

A interface de acesso a estas funções básicas é feita através da função `IOCTL`, utilizada geralmente para manipulação de arquivos especiais em Linux. Módulos de *kernel* são considerados arquivos especiais, possuindo uma entrada correspondente no diretório `/dev`. O módulo de *kernel* da biblioteca implementa a função `IOCTL` e a disponibiliza para os processos em espaço de usuário. Os parâmetros passados para essa função indicam ao módulo de *kernel* a execução da função `checkpoint()` ou `restart()`, retornando os resultados correspondentes destas funções. Maiores informações sobre a interface de acesso às funções do módulo de *kernel* serão apresentadas na Subeção 4.2.3.

A biblioteca é disponibilizada integralmente, com o código-fonte do módulo de *kernel* e dos processos-exemplo. Também são disponibilizadas instruções de compilação e uso. A biblioteca foi projetada para ser utilizada com a versão 2.4 do *kernel* do Linux.

4.2.2 Interceptação

A interceptação de mensagens da aplicação é um dos mecanismos essenciais ao funcionamento do algoritmo. Entre as alternativas disponíveis, optou-se pela interceptação de mensagens em espaço de *kernel* – no nível de chamadas de sistema ou *system calls*. Dessa forma, a interceptação de mensagens é realizada através da interceptação das chamadas de sistema geradas pela aplicação.

A implementação da interceptação em espaço de *kernel* sugere o uso de um módulo de *kernel* o que, a partir dos recursos disponíveis na biblioteca CRAK, fornece duas alternativas de implementação: construir um módulo de *kernel* específico para interceptação de chamadas de sistema ou adaptar o módulo de *kernel* da biblioteca CRAK, adicionando essa funcionalidade. Tendo em vista que o esforço para implementação seria equivalente nas duas alternativas, optou-se pela segunda. Dessa forma, além de simplificar o conjunto de módulos do sistema, a biblioteca receberia adaptações visando torná-la mais robusta.

Conforme apresentado no Capítulo 2, a interceptação de chamadas de sistema em espaço de *kernel* é possível através do desvio das chamadas para funções que, após realizarem algum processamento, devolvem ao curso original as chamadas interceptadas. A Figura 4.4 representa esse mecanismo. Em (a) observa-se o caminho original de uma chamada de sistema gerada pela aplicação. Em (b) é apresentada a mesma chamada de sistema sendo interceptada, desviada e em seguida retomando o seu curso original.

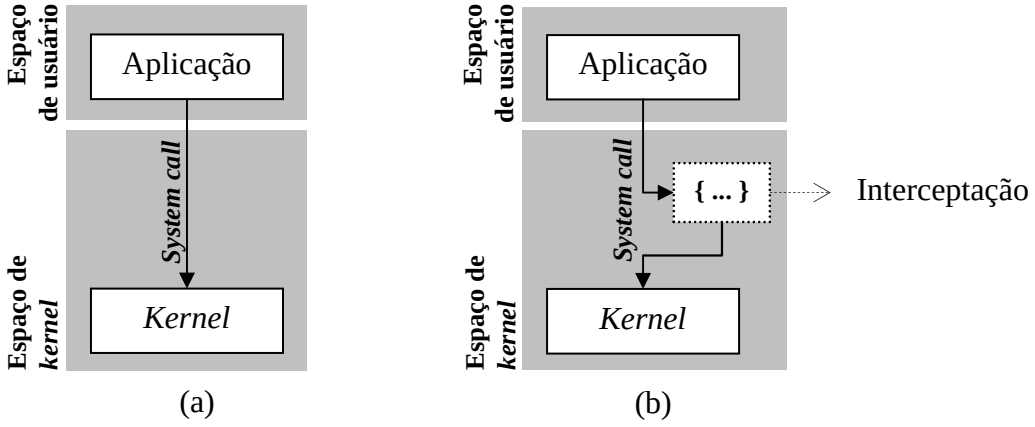


Figura 4.4: Interceptação de *system calls*.

Em Linux, todas as chamadas de sistema possuem seus endereços de *kernel* armazenados na tabela `sys_call_table`. Quando uma aplicação executa uma chamada de sistema, o *kernel* localiza nessa tabela o endereço da função e então a executa. O "gancho" explorado na interceptação é justamente alterar os endereços das chamadas de sistema que se deseja interceptar para o endereço de funções implementadas no módulo de *kernel*. Estas funções devem possuir o mesmo número e tipo de argumentos da função original. Assim, após realizar algum processamento acessório, essas novas funções devem chamar a função original ou, em alguns casos, executar toda tarefa que deveria ser executada pela função original.

A troca de endereços é realizada no momento em que o módulo é inserido no sistema. O processo inverso, por sua vez, é realizado no momento da remoção do módulo (RUBINI, 1999). Para isso, é necessário armazenar os endereços originais das chamadas de sistema a serem interceptadas para posteriormente restaurá-las.

Essa troca de ponteiros foi realizada no módulo de *kernel* da biblioteca CRAK, estendendo sua funcionalidade para suprir os mecanismos de interceptação necessários para a implementação do algoritmo.

No presente trabalho, são interceptadas somente três chamadas de sistema. A primeira corresponde à chamada `sys_execve`, necessária para detectar a carga de uma

aplicação. A segunda refere-se à chamada `sys_exit` e tem o papel de detectar o término da execução de uma aplicação. A terceira chamada de sistema interceptada é a chamada `sys_socketcall`, que corresponde às funções de troca de mensagens pela aplicação. Para cada uma dessas chamadas, foram implementadas funções correspondentes, com os mesmos argumentos das funções originais. A Figura 4.5 apresenta o código-fonte dessas trocas de ponteiros.

```
/* Armazena o endereço da system call original para posterior restauração */
original_sys_socketcall = sys_call_table[__NR_socketcall];
original_sys_execve     = sys_call_table[__NR_execve];
original_sys_exit       = sys_call_table[__NR_exit];

/* Altera os endereços da tabela causando o desvio desejado */
sys_call_table[__NR_execve]     = my_execve;
sys_call_table[__NR_exit]       = my_exit;
sys_call_table[__NR_socketcall] = my_socketcall;
```

(a) carga do módulo de *kernel*

```
/* Restaura as system calls originais na tabela */
sys_call_table[__NR_execve]     = original_sys_execve;
sys_call_table[__NR_exit]       = original_sys_exit;
sys_call_table[__NR_socketcall] = original_sys_socketcall;
```

(b) remoção do módulo de *kernel*

Figura 4.5: Alterações na tabela *sys_call_table*.

Em (a) os ponteiros originais das três chamadas de sistema são salvos e, em seguida, substituídos pelos endereços das funções implementadas no módulo de *kernel*. Em (b) é realizado o processo inverso. Essas operações são executadas, respectivamente, nas funções `init_module()` e `cleanup_module()`, que correspondem às funções de inserção e remoção do módulo de *kernel* no sistema (RUBINI, 1999).

A função `my_execve()` executa o comando disparado pelo usuário, através da execução da função de *kernel* `do_execve()`, e em seguida testa se o comando corresponde à identificação da aplicação informada pelo usuário pelo sistema de *checkpoint*. Caso afirmativo, dispara uma mensagem especial `SIGSTOP` para a aplicação, causando a sua imediata pausa antes de efetuar qualquer operação. Nesse ponto, a aplicação foi carregada para a memória mas aguarda o início da execução, que será disparado pelo sistema de *checkpointing*. Em seguida, a função armazena o identificador do processo (PID) atribuído para a aplicação e encerra, retornando ao *shell* do usuário o resultado da operação. A função `my_exit()` é acionada toda vez que uma aplicação for encerrada normalmente. Caso o identificador da aplicação (PID) corresponda àquele detectado pela função `my_execve()`, é considerado que a aplicação finalizou sua execução naquele computador.

A função `my_socketcall()`, por sua vez, é acionada toda vez que o processo identificado na função `my_execve()` gerar uma chamada de sistema para manipulação de *sockets*. Isso inclui todas as operações comumente utilizadas pelos processos de usuário, tais como comandos para abrir e fechar um *socket*, comandos para enviar e receber uma mensagem, etc (BECK *et al.*, 1999). Entretanto, somente a interceptação dos comandos de envio (`SYS_SENDTO`) e recebimento (`SYS_RECVFROM`) de mensagens é processada. A Figura 4.6 apresenta o código correspondente processado a cada chamada de sistema interceptada no módulo de *kernel*.

```
/* Função para interceptar a carga da aplicação */
asmlinkage int my_execve(struct pt_regs regs) {
   int error;
   // Executa comando informado pelo usuário
   error = do_execve(getname((char *) regs.ebx),
                     (char **) regs.ecx,
                     (char **) regs.edx, &regs);
   // Testa a identificação da aplicação
   if (strcmp(getname((char *) regs.ebx),proc_to_intercept)==0) {
       pid_to_intercept = current->pid;
       send_sig(SIGSTOP, current, 0);
       ckpt_devstatus = CKPT_STATUS_RUNNING;
   }
   return error;
}

/* Função para interceptar a finalização da aplicação */
asmlinkage int my_exit(int error_code) {

   if(pid_to_intercept == current->pid) {
      pid_to_intercept = 0;
      ckpt_devstatus = CKPT_STATUS_READY;
   }
   return original_sys_exit(error_code);
}

/* Função para interceptar as mensagens da aplicação */
asmlinkage int my_socketcall(int call, unsigned long *args) {
   int error;

   if ((call==SYS_SENDTO) && (current->pid==pid_to_intercept)) {
        /* Processa mensagens enviadas */
        }

   if ((call==SYS_RECVFROM) && (current->pid==pid_to_intercept)) {
        /* Processa mensagens recebidas */
        }
   error = original_sys_socketcall(call,args);
   if error<0 {
        /* Trata erro retornado pela chamada original */
        }
   return error;
}
```

Figura 4.6: Funções de interceptação

No envio de uma mensagem, a função `my_socketcall()` altera o conteúdo desta, inserindo no final do corpo da mensagem uma estrutura contendo o seu identificador (representado por uma variável inteira que é incrementada a cada mensagem) e o identificador da linha de recuperação atual (`CurrCP`) . Em seguida, é efetuada uma cópia dessa mensagem e do endereço do seu destinatário em um *buffer* (*Buffer* I da Figura 3.4) para então ser devolvida ao *kernel* para ser efetivamente enviada para o destinatário.

No recebimento da mensagem, a função `my_socketcall()` remove a estrutura previamente inserida no corpo da mensagem, insere uma mensagem de controle contendo as informações extraídas em um *buffer* (*Buffer* I da Figura 3.4) e devolve ao *kernel* para ser entregue à aplicação. O processamento das informações desse *buffer* será discutido nas seções posteriores.

Diferente da função `my_execve()`, as funções `my_exit()` e `my_socketcall()` inicialmente executam seus comandos e em seguida devolvem a chamada de sistema para o seu endereço original, armazenada nas variáveis `original_sys_exit` e `original_sys_socketcall` respectivamente. A função `my_socketcall()` implementa ainda um tratamento de possíveis erros produzidos pelas chamadas originais, o qual será apresentado posteriormente como solução alternativa para tratamento de erros relativos a manipulação de *sockets UDP* na recuperação de processos.

As três chamadas de sistema citadas são interceptadas quando acionadas por qualquer processo. Entretanto, somente é imposto um processamento adicional para aquelas cujo identificador do processo corresponda à aplicação especificada. Para as demais, as chamadas são simplesmente devolvidas ao *kernel* para serem executadas normalmente. O impacto causado por essa operação sobre o desempenho é praticamente desprezível, conforme o trabalho de Fontoura (2002).

4.2.3 Interface de acesso

Conforme abordado anteriormente, um módulo de *kernel* é considerado um tipo de arquivo especial, possuindo uma entrada no diretório `/dev` em plataformas Linux (RUBINI, 1999). Dessa forma, sob o ponto de vista de um processo de usuário, o acesso a módulos de *kernel* é realizado através das funções básicas ou operações de manipulação de arquivos, tais como `open()`, `read()`, `write()`, `close()`, `ioctrl()`, etc. Entretanto, diferente de um arquivo convencional, onde o próprio sistema operacional pode implementar estas funções, em módulos de *kernel* elas devem ser programadas no próprio módulo e disponibilizadas para as camadas superiores do sistema operacional. É através da chamada dessas funções que é possível trocar informações entre um módulo de *kernel* e um processo de usuário. Isso sempre é realizado “de cima para baixo”, ou seja, o processo de usuário invoca essas funções, as quais serão processadas pelo módulo de *kernel* e o resultado é devolvido ao processo. Aquelas funções que não forem implementadas, se acionadas pelo processo de usuário, retornarão um erro gerado pelo sistema operacional.

No presente trabalho, foram implementadas três funções para interface entre o processo **CK** e o módulo de *kernel*. A primeira é a função `open()`, herdada da biblioteca CRAK. Ela permite a abertura do módulo de *kernel* e deve ser executada antes de qualquer outra função. A função `ioctrl()`, que também já estava disponível para acionar as funções da biblioteca CRAK, foi estendida para receber novos comandos do processo **CK**. Através dessa função, o processo **CK** informa ao módulo de *kernel* o nome da aplicação que deve ser interceptada, recupera o identificador do processo (PID) da aplicação e solicita à biblioteca CRAK o salvamento ou a recuperação de processos.

A terceira função implementada é a função `read()`. É através dela que o processo **CK** tem acesso às mensagens interceptadas pelo módulo **CKPT** para serem processadas junto aos mecanismos de gerência do algoritmo.

Na Seção 3.3, foi indicado que as mensagens interceptadas são armazenadas em um *buffer* (*Buffer* I da Figura 3.4). O que a função `read()` implementa é a leitura ou remoção das mensagens desse *buffer*, entregando-as para o processo **CK**. Caso não haja nenhuma mensagem para ser lida, equivalente a um arquivo vazio, a função retorna uma indicação de erro para o processo **CK**. As demais funções relacionadas a arquivos não

foram implementadas por não haver necessidade de usá-las para os propósitos desse trabalho.

4.2.4 Recuperação de *sockets UDP*

Conforme apresentada anteriormente, a biblioteca `CRAK` não implementa a recuperação de *sockets*, apesar da tentativa de recuperação de conexões *sockets TCP* através de um protótipo. Para o presente trabalho, foi implementado o tratamento de *sockets UDP*, uma vez que o modelo de computação distribuída sugere a existência de aplicações que se comunicam exclusivamente pela troca de mensagens, através da rede de comunicação. Dessa forma, restringiu-se as aplicações àquelas que se comunicam via *sockets UDP*.

O sistema operacional Linux não disponibiliza, por padrão, nenhuma alternativa para salvamento e recuperação de *sockets*. Seria necessário efetuar adaptações no código-fonte do *kernel* para prover essa característica. Por outro lado, existem algumas alternativas de uso de bibliotecas para tratamento de quebra de conexões, como apresentado por Ekwall (EKWALL; URBÁN; SCHIPER, 2002) e Zandy (2002). Entretanto, estas bibliotecas exigem adaptações nas aplicações, tornando-as dependentes dessas bibliotecas. Em termos gerais, estas bibliotecas implementam pontos de sincronização na camada de sessão, alternativa oferecida pelo modelo de referência ISO/OSI (ISO, 1996) para re-estabelecer, de forma transparente, conexões quebradas na camada de transporte.

Testes realizados com a biblioteca `CRAK` sobre aplicações que manipulam *sockets* durante o processamento mostraram que, ao serem recuperadas, as aplicações passavam a não mais enviar nem receber mensagens. No caso de *socket UDP*, as funções `sendto()` e `recvfrom()`, utilizadas respectivamente pelas aplicações para envio e recebimento de datagramas, passavam a retornar erro. A não ser que houvesse um tratamento adequado desse erro por parte das aplicações, todo o seu processamento estaria comprometido após a recuperação a partir de um estado previamente salvo.

Constatou-se também, através da depuração dessas aplicações após a recuperação, que os erros indicam o uso de descritores inválidos, idênticos aos retornados quando as funções são invocadas sem a respectiva abertura do *socket* endereçado pelo descritor. Em termos práticos, ao recuperar uma aplicação, as variáveis com os descritores dos *sockets* indicam que estão ativas (como estavam no momento do seu salvamento). Entretanto, ao usar-se esses descritores em funções tais como `sendto()` e `recvfrom()`, é retornada uma condição de erro (o *kernel* não aceita os descritores).

Questionado por e-mail sobre essa restrição, Hua Zhong, um dos autores, informou que a biblioteca `CRAK` precisaria ser complementada para fornecer o tratamento adequado de *socket UDP* (Anexo B). Essa atividade foi iniciada por ele em um protótipo, buscando salvar e recuperar *sockets* orientado a conexão, ou TCP. *Sockets* não orientados a conexão, como *UDP*, não são tratados em nenhum momento. Nesse contato, o autor sugere que o tratamento de *sockets* não orientados a conexão é trivial, análogo ao tratamento de descritores de arquivos, conforme apresentados na Subseção 4.2.1.

A partir dessa orientação e das investigações preliminares, constatou-se que o tratamento poderia ser realizado em espaço de *kernel*, ao nível de chamadas de sistema, executando os mesmos procedimentos que seriam executados pela aplicação caso a mesma tivesse implementado o tratamento adequado dos erros. Dessa forma, após recuperar um estado previamente salvo, o erro retornado pelo *kernel* na chamada de uma

função *socket* seria interceptado, podendo ser resolvido antes de devolver o resultado da função para a aplicação. Na Figura 4.6 pode-se observar esse esquema.

Esse tratamento é realizado da seguinte forma: após interceptar, efetuar o processamento necessário e executar as chamadas de sistema de troca de mensagens, os erros retornados por estas últimas são tratados. Caso haja erro, é feita uma tentativa de reabertura do *socket* utilizando o mesmo descritor informado pela aplicação. A reabertura do *socket* dos processos clientes (aqueles que usam *sockets* para enviar mensagens) envolve a execução de chamadas `sys_close()` para fechar o *socket* supostamente apontado pelo descritor, `sys_socket()` para criar um novo *socket* e `sys_connect()` para re-conectar o descritor. No caso de processos servidores (aqueles que recebem requisições), é ainda executada a chamada `sys_listen()` para tornar o descritor apto a ser usado em funções de recebimento de mensagens. O uso conjugado dessas chamadas equivale às funções de manipulação de *sockets* invocadas pelos processos de usuário.

Após a tentativa de reabertura dos *sockets* indicados pelos respectivos descritores, as chamadas que retornaram erro são novamente invocadas e o seu resultado é devolvido para a aplicação. Essa solução mostrou-se bastante eficiente e solucionou os erros gerados na recuperação das aplicações e, principalmente, é efetuada de forma transparente sob o ponto de vista das aplicações.

Entretanto, essa solução restringe-se às chamadas de sistema envolvendo *sockets UDP*, não orientados a conexão, onde não há necessidade de sincronização dos dois lados da comunicação. Essa restrição será observada nas aplicações usadas nesse trabalho, a serem apresentadas no Capítulo 5. A recuperação de *sockets TCP*, orientados a conexão, por ser bastante complexa, não foi implementada no presente trabalho.

4.3 Processo CK

O segundo componente do sistema de *checkpointing* é o processo **CK**. Trata-se de um processo de usuário onde foram implementados os mecanismos de gerência do algoritmo. Associado às funções disponíveis no módulo **CKPT**, para salvamento e recuperação de processos e interceptação de mensagens, o processo **CK** é o módulo ativo do processamento. Nele são disparados e processados os mecanismos complementares do sistema: início do processamento, estabelecimento de linhas de recuperação da aplicação, término do processamento e mecanismos auxiliares, como rotação de coordenadores, detecção de defeitos e recuperação.

Conforme apresentado na Figura 4.1, uma instância do processo **CK** deve estar presente em todos os nodos do sistema em que a aplicação estiver executando. A união de todas estas instâncias forma o grupo de processos **CK** (também denominados membros do grupo), onde as atividades de tolerância a falhas são executadas em conjunto. Os mecanismos que serão apresentados nessa seção estão presentes em todos os processos **CK** do grupo.

A comunicação entre os processos **CK** é realizada exclusivamente através da troca de mensagens. O mecanismo de troca de mensagens foi implementado utilizando *sockets* UDP. Cada processo participante do sistema é identificado pelo endereço IP da sua estação e uma porta específica. Através desse endereço, os processos **CK** enviam e recebem mensagens pela rede. São utilizadas tanto funções *unicast* como funções

multicast nos procedimentos de troca de mensagens. Os principais mecanismos implementados envolvendo troca de mensagens serão apresentados na Seção 4.5.

A comunicação com o módulo **CKPT**, conforme apresentado anteriormente, é realizada através da chamada das funções `read()` e `ioctrl()` sobre um descritor de arquivos apontado para aquele módulo. Através do módulo **CKPT** tem-se acesso às mensagens interceptadas e à identificação do processo da aplicação.

Funcionalmente, o processo **CK** é constituído por um conjunto de *threads*, com funções específicas, que trabalham em cooperação compartilhando uma mesma área de memória. O compartilhamento consistente dessa área de memória é garantido através da utilização de semáforos (MATTHEW; STONES, 1999).

As funções implementadas no processo **CK** foram divididas em quatro blocos: 1) gerência do protocolo; 2) processamento das mensagens da aplicação; 3) *checkpoint* e recuperação; e 4) detector de defeitos. Apesar dessa divisão, há uma coesão entre os blocos, através do compartilhamento de informações e de funções comuns entre ambos.

O bloco de **gerência do protocolo** é a parte central do algoritmo. Nele são implementadas as rotinas de *commit* (Jalote, 1994) relativas ao início e término do processamento da aplicação distribuída, ao estabelecimento de linhas de recuperação, à recuperação da aplicação em caso de falha e aos demais mecanismos acessórios, como rotação de coordenadores e manutenção de parâmetros de execução. As informações recebidas do processamento de mensagens da aplicação e do detector de defeitos podem levar a ações, tais como o estabelecimento de um novo *checkpoint* (baseado em informações anexadas nas mensagens da aplicação), envio de notificações (*acks*) com confirmação de recebimento das mensagens da aplicação e interrupção do processamento em caso de falha.

Praticamente toda a comunicação de controle do sistema de *checkpointing* é centralizada no bloco de gerência do protocolo. Nele estão implementadas as funções de envio de mensagens *unicast* e *multicast* e as funções de recebimento de mensagens. Todas as mensagens recebidas são imediatamente transferidas para um *buffer*, de onde serão lidas para posterior processamento. Na medida em que as mensagens forem sendo retiradas do *buffer* e de acordo com o seu tipo, diferentes funções serão acionadas, podendo inclusive serem processadas em paralelo.

O **processamento de mensagens da aplicação** tem a função de manter atualizada a lista de mensagens potencialmente perdidas, ou seja, aquelas mensagens que foram enviadas pela aplicação mas não há confirmação de que tenham sido recebidas no seu destino. Esse controle baseia-se nas mensagens interceptadas pelo módulo de *kernel*. Mensagens interceptadas como enviadas são armazenadas em memória até que a resposta correspondente seja recebida. Mensagens interceptadas como recebidas são repassadas para a gerência do protocolo, a qual se encarrega de transmitir uma mensagem de controle (*ack*) sinalizando o recebimento da mensagem da aplicação. Essa mensagem de controle, quando recebida pelo bloco de gerência do protocolo, será repassada para o processamento de mensagens da aplicação para descartar a mensagem correspondente, registrada como enviada pela aplicação.

Caso nenhuma resposta seja recebida em um tempo pré-definido, limitado em 3 segundos neste trabalho, a mensagem registrada como enviada pela aplicação é retransmitida por um número limitado de tentativas (5 tentativas por padrão). Caso ainda não haja resposta, a mensagem será descartada, assumindo que houve perda da respectiva

mensagem pela aplicação e esta perda deverá ser tratada pela própria aplicação. Essa situação pode ocorrer em casos onde há grande perda de mensagens no canal de comunicação. Entretanto, é possível que esse tipo de deficiência seja sinalizado pelo detector de defeitos.

O bloco de processamento de mensagens da aplicação também pode sinalizar à gerência do protocolo a necessidade de estabelecimento de um novo *checkpoint*. Essa situação ocorre quando uma mensagem da aplicação, já identificada com uma nova linha de recuperação, for recebida antes da mensagem de controle específica para essa finalidade. Quando essa mensagem de controle for recebida, ela será respondida sem necessidade de estabelecer um novo *checkpoint*, pois o mesmo já foi estabelecido.

A lista de mensagens da aplicação para as quais não veio confirmação de recebimento fará parte do *checkpoint* e será salva no momento do estabelecimento deste. A função responsável por essa operação será executada pelo **bloco de *checkpoint* e recuperação**, no momento em que salvar a imagem do processo em execução através das funções da biblioteca CRAK. Esse bloco será acionado pelo bloco de gerencia do protocolo, tanto em situações de *checkpoint*, como em situações de recuperação.

O **detector de defeitos** opera em paralelo com todas as operações do algoritmo. Sua função é informar ao bloco de gerência do protocolo a ocorrência de erros (que se manifestam como defeitos do módulo correspondente) ou a recuperação do sistema após cessarem os efeitos do defeito (*recover*).

4.4 Processo MONITOR

Buscando centralizar o gerenciamento e monitorar o sistema de *checkpointing*, foi desenvolvida uma terceira ferramenta, denominada processo **MONITOR**. Esse processo implementa uma interface com o usuário para inserção de comandos e apresentação de resultados de todos os nodos participantes do sistema. Através dele, o usuário poderá iniciar, interromper, configurar e acompanhar o processamento do sistema antes, durante e após a execução de sua aplicação.

O processo **MONITOR** comunica-se exclusivamente com os processos **CK**, através de mensagens de controle enviadas diretamente para um nodo (*unicast*) ou para todos os nodos do sistema (*multicast*). Estas mensagens representam comandos específicos, e são recebidas pelos processos **CK**, processadas e respondidas com as informações solicitadas. Os comandos dividem-se em comandos para alteração de informações (comandos de configuração) e comandos para consulta de informações. A Figura 4.7 apresenta a interface de utilização dessa ferramenta durante a execução do sistema de *checkpointing* sob uma aplicação disparada em seis computadores, composta por um processo servidor (*server*) e cinco processos clientes (*client*).

Entre as informações coletadas pelo processo **MONITOR**, exibidas na Figura 4.7, estão a lista dos processos da aplicação distribuída, informações sobre esses processos tais como nome do arquivo executável, identificador do processo (PID), mensagens enviadas e recebidas, estado da execução em cada nodo e identificação do processo coordenador, indicado pela letra “M” na figura.

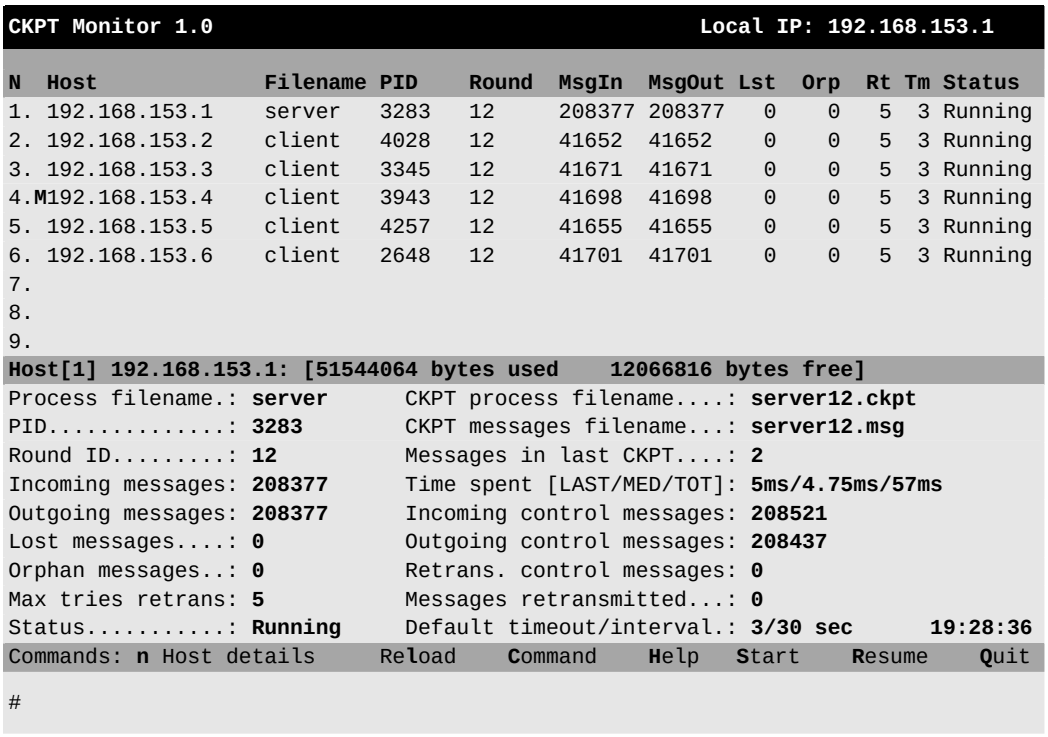


Figura 4.7: Interface do processo **MONITOR**.

Foram classificados seis diferentes estados possíveis para a execução da aplicação, os quais são:

1. `Ready`: a aplicação foi disparada mas ainda não iniciou o processamento ou a aplicação encerrou normalmente o processamento;
2. `Running`: a aplicação está em execução livre de falhas;
3. `Wait`: o processamento da aplicação foi interrompido pelo sistema de *checkpointing* para posterior liberação (essa situação é assumida quando a aplicação for recuperada após a ocorrência de uma falha);
4. `Process crash`: o processo da aplicação apresentou anormalidades percebidas pelo detector de defeitos;
5. `CKPT crash`: o próprio sistema de *checkpointing* apresentou algum defeito;
6. `Not resp`: o nodo não responde às requisições do **MONITOR**.

Além das informações sobre os processos da aplicação, são coletadas informações sobre o sistema de *checkpointing*, tais como arquivos gerados no último *checkpoint*, o identificador da linha de recuperação, estatísticas sobre os tempos gastos com o salvamento dos *checkpoints* em disco, número de mensagens de controle envolvidas e parâmetros de execução.

Não é necessária qualquer configuração adicional para executar o processo **MONITOR**. Uma vez iniciado, ele detecta automaticamente, através de mensagens específicas, a presença de todos os processos **CK** que fazem parte do sistema de *checkpointing*. Durante a monitoração, é possível inserir alguns comandos de consulta e

de configuração do sistema. Para consulta, existem dois comandos: “l” para forçar a coleta de informações de todos os processos do sistema e “n”, representando o número do processo da lista, usado para exibir informações detalhadas do nodo correspondente. Mesmo sem a inserção de comandos, o processo **MONITOR** coleta periodicamente as informações de todos os processos e as atualiza na tela.

Os **comandos de configuração** dividem-se em duas categorias: comandos para interferir na execução do sistema e comandos para ajuste de parâmetros. Para a primeira categoria foram implementados os comandos “S” para iniciar o processamento em todos os nodos do sistema e “R” para forçar uma pausa nesse processamento. Esses comandos são enviados a todos os nodos simultaneamente, através de mecanismos de *multicast*. É possível enviar os comandos anteriores, individualmente, aos processos. Para isso, usa-se o comando “C” para digitar um comando e, no *prompt* apresentado em seguida, informa-se o comando `start n` ou `stop n`, onde *n* representa o nodo do sistema. Para efeito de teste do sistema detector de defeitos, a ser apresentado posteriormente, foi ainda implementado o comando `kill [n] [all]`. Através desse é possível, remotamente, forçar a interrupção dos processos da aplicação de um determinado nodo ou de todos os nodos, respectivamente.

Na categoria de **comandos para ajuste de parâmetros** estão aqueles usados para alteração de variáveis usadas no sistema. Estas variáveis incluem o *timeout* para operações de controle do algoritmo, o número de tentativas de retransmissão de uma mensagem, o intervalo de tempo entre duas linhas de recuperação e o nome dos processos da aplicação do usuário. Todas as medidas de tempo são representadas em segundos.

Os comandos correspondentes são listados a seguir:

- `set timeout x to [n] [all]`: ajusta a variável *timeout* em *x* segundos para o nodo *n* ou para todos (*all*) os nodos;
- `set retrans x to [n] [all]`: ajusta a variável que representa o número máximo de tentativas de retransmissão de uma mensagem em *x* vezes para o nodo *n* ou para todos (*all*) os nodos;
- `set interval x to [n] [all]`: ajusta a variável que representa o intervalo de tempo entre o estabelecimento de duas linhas de recuperação consecutivas em *x* segundos, para o nodo *n* ou para todos (*all*) os nodos (essa informação é relevante somente para o processo coordenador da rodada e pode ser diferente entre os nodos);
- `set filename name to [n] [all]`: ajusta o nome e o caminho (*name*) do arquivo executável que corresponde ao processo da aplicação do usuário, para o nodo *n* ou para todos (*all*) os nodos.

Após a inserção de qualquer um dos comandos acima, é possível observar o resultado correspondente, verificando-se as informações coletadas e exibidas na tela. Se o comando for executado com sucesso, as alterações serão observadas em seguida.

A execução do sistema de *checkpointing* não está condicionada à utilização do processo **MONITOR**. As informações para ajuste de parâmetros podem ser inseridas no momento da execução do processo **CK**, conforme instruções apresentadas no momento da execução daquele processo. Da mesma forma, o processo **MONITOR** pode ser encerrado antes da finalização da execução da aplicação, uma vez que ele não interfere na execução

do sistema de *checkpointing*, exceto pela geração adicional de algumas mensagens de controle necessárias ao seu funcionamento.

O processo **MONITOR** pode ser executado em qualquer nodo ou computador conectado à mesma sub-rede em que o sistema estiver sendo executado. Isso significa que, no caso do exemplo apresentado na Figura 4.7, o processo **MONITOR** pode ser executado a partir de um sétimo computador, onde o sistema de *checkpointing* não esteja ativado ou nem mesmo instalado.

4.5 Mensagens

Após serem apresentados os três componentes que formam o sistema de *checkpointing* (módulo **CKPT**, processo **CK** e processo **MONITOR**), serão apresentadas nessa seção as mensagens envolvidas na implementação dos mecanismos do sistema. Sua implementação propriamente dita será alvo da Seção 4.6.

O algoritmo adotado nesse trabalho considera a existência de dois tipos de mensagens: mensagens da aplicação e mensagens de controle. Mensagens da aplicação correspondem às mensagens geradas pela aplicação, as quais serão interceptadas para inserção e remoção de informações de controle do algoritmo. Mensagens de controle envolvem todas as mensagens adicionais geradas pelo algoritmo, necessárias para o seu funcionamento.

As mensagens da aplicação são interceptadas em espaço de *kernel*, no módulo **CKPT**, e seu conteúdo é alterado com a inserção ou remoção de duas informações de controle: um identificador da mensagem e o índice correspondente ao último *checkpoint* estabelecido no processo transmissor. Estas informações são inseridas em mensagens enviadas pela aplicação e removidas das mensagens recebidas pela aplicação. Estas mensagens devem corresponder a datagramas UDP.

A estrutura de um datagrama, além de informações de cabeçalho, contém uma área reservada para dados que pode chegar a 65515 bytes (TANENBAUM, 1996). Em Linux, essa área é mapeada para um *buffer*, endereçado por um ponteiro do tipo caractere (`char *`). As informações de controle são inseridas, e posteriormente removidas, nesse *buffer* ao serem interceptadas. O formato adotado nesse trabalho é a estrutura "`[x;y]`", ou seja, *x* representando o identificador da mensagem e *y* representando o índice do último *checkpoint* estabelecido (armazenado na variável `PrevCP`). Por exemplo: o processo A envia uma mensagem de conteúdo "`ABCD1234`" para o processo B. Suponha-se que, no momento da interceptação, é contabilizado que essa é a 1000ª mensagem enviada pelo processo A (incrementado na variável `count_messages` listada no Apêndice B, linha 76) e que o último *checkpoint* estabelecido por ele possui o índice 20. Dessa forma, ao ser interceptada, a mensagem enviada pelo processo A passa a ter o conteúdo "`ABCD1234[1000;20]`". No nodo receptor, essa mensagem será novamente interceptada e terá a estrutura "`[1000;20]`" removida e processada antes da mensagem ser entregue para o processo B. Da estrutura removida da mensagem recebida, o sistema retira o identificador e o índice do último *checkpoint* efetuado no processo emissor, aplicando então os procedimentos do algoritmo. Esse procedimento é transparente para a aplicação, ou seja, ela não percebe que suas mensagens estão sendo usadas pelo sistema de *checkpointing*.

A cada mensagem recebida pela aplicação, o sistema gera uma mensagem de confirmação (`CKPT_ACK_APPMSG`), enviada para o nodo emissor da mensagem interceptada, através de uma mensagem de controle. Essa mensagem carrega o identificador da mensagem interceptada e também o índice do último *checkpoint* local. Ao receber essa mensagem, o nodo emissor da mensagem da aplicação obtém a confirmação de que a mensagem enviada foi entregue no nodo receptor.

O procedimento anterior é necessário para reduzir o número de mensagens a serem salvas no *checkpoint*. Os mecanismos de consistência do algoritmo sugerem que, para evitar a existência de mensagens perdidas após a recuperação, é necessário que as mensagens geradas pela aplicação sejam salvas junto com o *checkpoint*. Entretanto, para efeitos práticos, somente é necessário salvar aquelas mensagens que foram enviadas mas que ainda não se sabe se realmente foram recebidas no seu destino. Essas mensagens são armazenadas na lista `ckpt_SendApp_list`, reproduzida no Apêndice B. Essa lista é constantemente atualizada e, no momento do estabelecimento de um *checkpoint* local, ela é salva na memória estável. Em caso de recuperação do sistema após a manifestação de uma falha, essa lista é recuperada e todas as mensagens são imediatamente retransmitidas para os seus respectivos destinatários.

A retransmissão dessas mensagens, após a recuperação, exigida pelo algoritmo, é realizada pelo processo **`CK`**. Esse procedimento tem a função de inserir as mensagens registradas como não respondidas no canal de comunicação. A aplicação, no receptor, recebe essas mensagens como se tivessem sido enviadas pela própria aplicação no nodo transmissor, garantindo assim os níveis de transparência assumidos nesse trabalho.

Como mecanismo adicional, foi também implementada a retransmissão de mensagens enviadas pela aplicação cuja resposta de confirmação de recebimento não foi recebida após um determinado prazo. A mensagem somente é descartada após um número determinado de tentativas de retransmissão sem a correspondente confirmação de recebimento.

Além das mensagens necessárias para o controle das mensagens da aplicação, foi implementada uma série de outras mensagens de controle utilizadas nos demais mecanismos do algoritmo. A classificação dos tipos de mensagens de controle implementadas pode ser observada na relação de constantes listadas no Apêndice A. As mensagens de controle são representadas pela estrutura `ckpt_msg_control`, listada no Apêndice B, linhas 19 a 25. Todas as mensagens de controle carregam o índice do último *checkpoint* estabelecido pelo processo emissor (`CurrCP`).

4.6 Operações de Acordo

Durante a execução do sistema de *checkpointing* são efetuadas operações de tomada de decisões entre todos os processos participantes. Estas decisões devem ser tomadas em acordo pelo grupo de maneira a garantir a sua consistência. Exemplos destas operações são os estabelecimentos de linhas de recuperação, a rotação de coordenadores, a recuperação em caso de falhas e mecanismos de detecção de defeitos.

O estabelecimento de acordo para as operações acima foi implementado através de uma aproximação do protocolo de duas fases (*two-phase commit protocol*) (JALOTE, 1994). Considera-se uma aproximação porque, diferente do protocolo genérico, o protocolo implementado nesse trabalho não envia mensagens de negação da operação solicitada. Ou seja, nem o coordenador, nem os participantes, enviam mensagens

informando que uma determinada solicitação não foi possível de ser executada ou que uma operação iniciada deve ser cancelada. Posteriormente, durante a apresentação destas operações, essa característica será justificada.

Na execução de operações envolvendo o protocolo de duas fases, o processo iniciador é caracterizado como coordenador e os demais processos são caracterizados como participantes. Na primeira fase, o processo coordenador envia requisições (mensagens do tipo `REQ`), representando uma determinada operação, para todos os participantes. Cada participante, ao receber a requisição, efetua o processamento solicitado e, caso o processamento tenha sido bem sucedido, responde ao coordenador (com mensagens de tipo `AREQ`). Nessa fase, o coordenador aguarda por um determinado período de tempo até que todos os participantes do grupo tenham respondido. Caso esse tempo expire sem que todas as respostas tenham sido recebidas, a tentativa de operação é abortada e, dependendo do caso, repetida por um número limitado de vezes ou são executadas outras ações.

Na segunda fase, após o coordenador ter coletado todas as respostas dos participantes, ele envia uma mensagem de confirmação da operação para todos os processos participantes (mensagem tipo `CMT`). Novamente, os processos participantes respondem a essa solicitação (mensagens tipo `ACMT`), assumindo que a operação foi bem sucedida. O coordenador, após receber todas as respostas de confirmação, também assume o sucesso da operação. Caso algum processo não responda, o coordenador não considera essa operação como bem sucedida, sendo posteriormente substituída por uma nova tentativa de execução da operação.

Nas próximas seções, serão apresentadas as operações implementadas nesse trabalho, onde será possível entender em exemplos os mecanismos de acordo utilizados.

4.6.1 Estabelecimento de linhas de recuperação consistentes

O ponto chave do trabalho de implementação abordado até aqui é o estabelecimento de *checkpoints* coordenados, um em cada nodo do sistema, de forma a garantir que a união desses *checkpoints* forme um ponto global consistente de recuperação da aplicação, também chamado de linha de recuperação (CECHIN, 2002).

Conforme apresentado anteriormente, em cada nodo do sistema estão presentes, no mínimo, três componentes: o processo da aplicação em execução, o módulo de *kernel* **`CKPT`** e o processo **`CK`**. Em cada um dos nodos, são estabelecidos periodicamente pontos de recuperação, ou *checkpoints*, a partir da solicitação explícita do coordenador ou a partir de informações anexadas às mensagens recebidas. Esses *checkpoints* são estabelecidos pelo processo **`CK`** através de funções do módulo de *kernel* **`CKPT`**. Fazem parte desses *checkpoints* a imagem do processo da aplicação e a lista de mensagens que poderão se tornar perdidas.

Entretanto, para que haja consistência entre todos os *checkpoints* estabelecidos nos nodos em que a aplicação estiver em execução, são necessários mecanismos de coordenação, especificados no algoritmo empregado nesse trabalho. O estabelecimento de uma linha de recuperação consistente garante que, em caso de falha, o processamento distribuído possa retornar para um estado anterior ao da manifestação da falha, de forma consistente.

Durante o processamento livre de falhas, a cada intervalo de tempo previamente estabelecido, o processo **`CK`** coordenador inicia o estabelecimento de uma linha de

recuperação. Inicialmente, para indicar que uma nova linha de recuperação está em fase de estabelecimento, ele altera o conteúdo da variável `ME` (ilustrada no Apêndice B, linha 42) para 0, atribui à variável `CurrCP` o valor da variável `PrevCP` mais uma unidade (`CurrCP=PrevCP+1`). Em seguida, estabelece um *checkpoint* local, identificado pelo valor inteiro armazenado na variável `CurrCP` e, antes de qualquer outra mensagem ser enviada ou recebida, envia uma mensagem do tipo `CKPT_MSG_REQ`, com o conteúdo da variável `CurrCP` anexado, para todos processos participantes. Essa mensagem é enviada uma única vez, através de primitivas de *multicast*, o que faz com que o próprio processo coordenador também a receba. Entretanto, essa mensagem, quando recebida pelo próprio coordenador, é descartada. Em seguida, o processo **`CK`** do coordenador aguarda pela resposta dos demais participantes.

Os processos participantes, por sua vez, ao receberem a solicitação do coordenador, comparam o conteúdo da sua variável `CurrCP` com o índice recebido na mensagem de requisição. Se o índice recebido for igual ao seu próprio índice, respondem ao coordenador enviando uma mensagem de tipo `CKPT_MSG_AREQ`, sem estabelecerem seus *checkpoints*. Se o índice recebido for maior que o seu próprio, significa que, antes de responderem ao coordenador, eles deverão estabelecer um *checkpoint* local. Nesse caso, a resposta somente será enviada caso houver sucesso na tomada desses *checkpoints*, situação em que o conteúdo da variável `CurrCP` local será então atualizado com o índice recebido na mensagem de requisição. Caso haja insucesso por parte de algum dos membros, haverá impossibilidade de estabelecimento da linha de recuperação causada por omissão dos membros que não estabelecerem com sucesso seus *checkpoints*.

A situação em que não foi necessário estabelecer um *checkpoint* (o conteúdo da variável `CurrCP` é igual ao índice recebido na mensagem de requisição) pode ocorrer quando o processo receber, antes da solicitação específica do coordenador, uma mensagem da aplicação cujo índice anexado já tenha um valor maior do que o índice local. Esse comportamento é previsto na especificação do algoritmo e implementado no bloco de processamento de mensagens da aplicação, discutido na Seção 4.3.

Uma vez tendo recebido as respostas `CKPT_MSG_AREQ` de todos os membros participantes, o processo coordenador já tem o conhecimento de que a linha de recuperação foi estabelecida. Entretanto, é necessário informar aos outros processos essa confirmação de estabelecimento, o que possibilitará disparar algumas atividades auxiliares. Esse aviso é efetuado através do envio de uma mensagem `CKPT_MSG_CMT`. Após o envio dessa mensagem, o coordenador aguarda a confirmação de recebimento dessa mensagem por todos os participantes.

Ao receberem a confirmação de estabelecimento da linha de recuperação, os processos participantes efetuam a atualização da variável `PrevCP` com o conteúdo da variável `CurrCP`, correspondendo ao índice da linha de recuperação estabelecida. Essa operação automaticamente atribui ao último *checkpoint* obtido a condição de componente da linha de recuperação estabelecida pela rodada. Em seguida, efetuam a coleta de lixo, ou seja, a exclusão do *checkpoint* com índice inferior ao novo índice, armazenado na variável `PrevCP`, e a atribuição do valor da variável `CurrCP` para essa variável (`PrevCP = CurrCP`). Após essas duas tarefas, os processos participantes respondem ao coordenador com uma mensagem `CKPT_MSG_ACMT`. Nesse ponto, os processos participantes já efetuaram a atualização de seus pontos de recuperação.

O processo coordenador, ao receber todas as mensagens `CKPT_MSG_ACMT`, efetua a atualização do seu ponto de recuperação, atualizando a variável `PrevCP` e efetuando a coleta de lixo. Dessa forma, encerra o estabelecimento da linha de recuperação, alterando o conteúdo da variável `ME` para `1`, estando pronto para efetuar a rotação de coordenadores (vide Subseção 4.6.2).

Durante o processo de estabelecimento da linha de recuperação, haverá um intervalo de tempo em que existirão dois pontos de recuperação em cada processo: um representado pela variável `PrevCP`, que corresponde à última linha de recuperação estabelecida, e outro representado pela variável `CurrCP`, que corresponde à linha de recuperação que está sendo estabelecida. Somente quando a segunda fase do protocolo de *commit* for executada é que a linha de recuperação `PrevCP` será descartada, sendo substituída pela nova linha indexada por `CurrCP`. Dessa forma, caso ocorra alguma falha durante o estabelecimento de uma linha de recuperação, é garantido que pelo menos existirá uma linha de recuperação consistente.

Por outro lado, caso o processo coordenador não consiga confirmar o estabelecimento de uma nova linha de recuperação após um número determinado de tentativas, é assumido que existe algum problema que não foi percebido pelo detector de defeitos. Esse problema pode causar a parada do processamento da aplicação ou a execução de mecanismos de recuperação.

Nos procedimentos de recuperação, mesmo não tendo sido executada a segunda fase, os pontos de recuperação representados pelo índice `CurrCP` poderão ser utilizados para a recuperação, pois podem formar um conjunto consistente. Dessa forma, conforme abordado anteriormente, a segunda fase não é essencial para garantir a existência de uma linha de recuperação. A implementação do processo de recuperação em caso de falha será apresentada na Seção 4.6.4.

4.6.2 Rotação de coordenadores

Um mecanismo auxiliar sugerido pelo autor do algoritmo de *checkpoiting* implementado nesse trabalho é a rotação de coordenadores. Esse mecanismo tem o objetivo de evitar que o processo responsável pela função de coordenação atue como um ponto único de falhas. A cada estabelecimento de uma linha de recuperação, um novo processo será responsável pela tarefa de coordenação, sendo identificado pelos demais participantes do sistema.

Antes de explicar o mecanismo específico de rotação de coordenadores, é necessário entender como os processos formam o grupo e decidem quem será o primeiro coordenador do sistema.

Como explicado anteriormente, cabe ao bloco de gerência do protocolo, implementado no processo **`CK`**, administrar as informações e efetuar os mecanismos especificados pelo algoritmo. Assim, ao ser iniciado, são efetuadas as verificações básicas do sistema (tais como a configuração do módulo de *kernel*) e, em seguida, o processo **`CK`** inicia o procedimento de formação do grupo. Nessa etapa, todos os processos **`CK`**, ao serem carregados, repetem os mesmos procedimentos.

A primeira ação para o estabelecimento do grupo é enviar uma mensagem *multicast* de tipo `CKPT_MSG_JOIN`, a qual anuncia aos demais processos a sua entrada no grupo. Nesse ponto, o processo **`CK`** não sabe da existência de nenhum outro processo no grupo.

Assim, a lista de processos do grupo (armazenada na variável `ckpt_list_group`) possui somente a identificação do próprio processo.

A mensagem `CKPT_MSG_JOIN`, ao ser recebida pelos demais processos, fará com que o processo emissor seja inserido na lista `ckpt_list_group` do receptor, com situação igual a `CKPT_MEMBER_JOIN`, e será respondida através do envio de uma mensagem do mesmo tipo (`CKPT_MSG_JOIN`). Caso o processo emissor dessa mensagem já esteja presente na lista local do receptor, sua situação é atualizada para `CKPT_MEMBER_JOIN`, além da emissão da respectiva mensagem de resposta. Dessa forma, inicialmente, todo o processo publica-se para os demais e recebe uma resposta de cada um dos presentes, possibilitando que tomem conhecimento sobre todos os membros do grupo. Na medida em que os processos **`CK`** forem iniciados, o grupo vai sendo formado entre todos os processos **`CK`**.

Exceto em caso de recuperação, o sistema de *checkpoint* somente passará dessa etapa após ser informado pelo usuário. Ou seja, cabe ao usuário carregar o sistema de *checkpointing* e a aplicação, inserindo posteriormente um comando específico para o início do processamento da aplicação. Esse comando pode ser inserido através do processo **`MONITOR`** ou através de parâmetros na carga do processo **`CK`**, é implementado através do envio de uma mensagem *multicast* de controle, do tipo `CKPT_MSG_START`.

Ao receber a mensagem `CKPT_MSG_START`, o passo seguinte é decidir quem será o primeiro processo coordenador. Por definição, adotou-se como primeiro coordenador aquele processo que tiver o maior número IP. Dessa forma, durante a carga dos processos e formação do grupo, automaticamente já é designado como processo coordenador aquele que possuir o maior IP. Também é assumido que a rotação de coordenadores obedecerá à ordem dos números IP dos processos, do menor para o maior, de forma circular. Após o primeiro ter assumido, o sistema identifica aquele que tem o menor número IP, seguindo-se o que apresenta o segundo menor IP e assim sucessivamente até o de número IP maior que os demais, quando se retorna ao de menor IP.

Entretanto, antes de iniciar o processamento da aplicação, a decisão sobre quem será o coordenador da primeira rodada é confirmada: para isso, o mecanismo de rotação de coordenadores é disparado pelo processo adotado na inicialização como processo coordenador do sistema. Esse mecanismo é o mesmo efetuado nas rodadas posteriores e, após o início do processamento da aplicação, não serão mais aceitas mensagens do tipo `CKPT_MSG_JOIN` para adesão de novos membros ao grupo. Concomitantemente, todos os processos **`CK`** efetuam o salvamento em disco da lista de membros do grupo, armazenados na variável `ckpt_list_group`. Essa operação é efetuada uma única vez e garante que, na recuperação, possa-se saber quais eram os membros do grupo antes da manifestação da falha.

O mecanismo de rotação de coordenadores é análogo ao mecanismo de estabelecimento de linhas de recuperação anteriormente apresentado. Ele segue o protocolo de duas fases, sendo que na primeira fase o processo coordenador propõe ao grupo, através de uma mensagem de tipo `CKPT_MSG_REQ_MANAGER`, o novo coordenador, de acordo com a ordem dos endereços de cada um. Após enviar essa mensagem, o coordenador aguarda resposta dos demais membros do grupo.

Ao receberem a proposta do coordenador, os demais processos respondem ao coordenador atual concordando com o valor proposto, com mensagens de tipo `CKPT_MSG_AREQ_MANAGER`, mas ainda não adotando esse valor. Ao receber as respostas de

todos os participantes, o processo coordenador envia uma mensagem de tipo `CKPT_MSG_CMT_MANAGER` para todos os participantes e fica aguardando suas respostas.

Ao ser recebida pelos processos participantes, a mensagem `CKPT_MSG_CMT_MANAGER` faz com que esses assumam o novo coordenador proposto e respondam ao coordenador antigo com mensagens de tipo `CKPT_MSG_ACMT_MANAGER`. O coordenador antigo, ao receber todas as confirmações, abdica do seu cargo e também reconhece o novo coordenador proposto. O processo que corresponde ao novo coordenador fica agora responsável por disparar, no momento apropriado, o estabelecimento de uma nova linha de recuperação.

Mais uma vez, problemas podem resultar desse mecanismo. Por exemplo, na segunda fase, caso algum processo não responda à mensagem de confirmação, poderá resultar na existência de dois coordenadores (o novo e o antigo), cada um adotado por um grupo parcial de processos. Para resolver esse problema, duas situações foram asseguradas: 1) os processos sempre usam a mesma regra de numeração IP, propondo o mesmo processo como candidato a novo coordenador caso haja necessidade de repetição da eleição proposta (sempre obedecendo à disposição circular dos membros em relação ao seu número IP); 2) o coordenador, caso não receba resposta de todos os participantes, reinicia imediatamente os procedimentos de rotação de coordenadores. Assim, mesmo tendo sido adotado um novo coordenador, os processos sempre concordarão com o novo valor proposto, sobrepondo um possível valor previamente assumido.

Eventualmente, é possível ainda que o sistema fique num processo contínuo de escolha do novo coordenador devido a um defeito ou devido à perda de mensagens de resposta. Para evitar esse comportamento, uma alternativa seria a utilização do protocolo de três fases (SINGHAL; SHIVARATRI, 1994) para implementar a rotação de coordenadores. Mesmo em alguns casos de defeitos ou de perda de mensagens, esse protocolo possibilita efetuar o acordo sobre o novo coordenador proposto.

4.6.3 Detecção de defeitos e procedimentos relacionados

A detecção de defeitos não é o objetivo central desse trabalho. Entretanto, a implementação deste módulo era indispensável para o funcionamento e ativação de alguns dos mecanismos do algoritmo. A implementação do detector de defeitos restringiu-se ao modelo de monitoração apresentado por Fetzer (FETZER; RAYNAL; TRONEL, 2001), com objetivos específicos de monitoramento de processos em Linux, buscando detectar defeitos por colapso (CRISTIAN, 1991). A monitoração implementada também permite dar início à recuperação após ter cessado o efeito do defeito detectado.

Os defeitos por colapso detectados no ambiente de execução adotado foram classificados em três tipos: defeitos nos processos da aplicação, defeitos no sistema de comunicação e defeitos no sistema de *checkpointing*.

Defeitos nos processos da aplicação correspondem às situações em que os processos param de responder às mensagens especiais (*signals*) ou são finalizados pelo sistema operacional por terem executado alguma operação ilegal, devido provavelmente a erros de programação.

Defeitos no sistema de comunicação envolvem situações em que os processos da aplicação e os processos que compõem o sistema de *checkpointing* param de se comunicar por um intervalo de tempo previamente estabelecido. É assumido, nesse caso, que há algum problema no sistema de comunicação que impede a troca de mensagens e,

conseqüentemente, a execução correta da aplicação distribuída e das operações do algoritmo. Por questões de simplificação, situações em que um membro pára de responder por ter sofrido defeito por colapso, como desligamento forçado de seu nodo, são tratadas como defeitos no sistema de comunicação.

Já os problemas que afetam o sistema de *checkpointing* envolvem situações em que os mecanismos implementados podem induzir a ocorrência de defeitos. Exemplos desses envolvem situações de estouro de *buffers* e impossibilidade de estabelecimento de linhas de recuperação devido a problemas no salvamento de *checkpoints*, como espaço insuficiente na memória estável, por exemplo.

Na ocorrência de uma das falhas anteriores, o detector de defeitos interrompe o processamento da aplicação distribuída, forçando a finalização dos processos e, após cessar o efeito do problema, inicia os procedimentos de recuperação. A interrupção da aplicação, nesses casos, foi implementada para evitar a existência de processos inconsistentes, como processos zumbis (TANENBAUM, 2001), e para garantir a liberação de recursos alocados pela aplicação. Além disso, os procedimentos de recuperação implementados exigem que, antes de desencadear a recuperação, a aplicação deve ser finalizada em todos os nodos em que ela estava sendo executada antes da detecção do defeito. No caso específico de falhas no sistema de *checkpointing*, são ainda executados procedimentos de esvaziamento dos *buffers* e liberação de memória com o objetivo de disponibilizar os recursos alocados pelo mesmo.

O funcionamento do detector de defeitos baseia-se em verificações periódicas envolvendo duas etapas: 1) verificar o estado local do sistema, consultando o estado dos *buffers*, o resultado das operações de estabelecimento de linhas de recuperação e o estado do processo da aplicação; e 2) efetuar uma verificação no estado dos demais membros do grupo.

A primeira etapa tem o objetivo de detectar localmente defeitos no sistema de *checkpointing* ou no processo da aplicação. Uma vez detectado um defeito nesse contexto, o detector de defeitos interrompe o processamento da aplicação, efetua a inicialização dos *buffers*, notifica os demais membros do grupo com uma mensagem de tipo `CKPT_MSG_NOTIFY_CRASH` e, após receber todas as respostas a estas mensagens, passa para o estado de recuperação (apresentado na Seção 3.5). Se necessário, novas mensagens de notificação são enviadas até que todos os demais membros respondam. Os demais processos, aqueles que recebem a mensagem com a notificação sobre o defeito detectado por um dos membros do grupo, respondem com mensagens de tipo `CKPT_MSG_ANOTIFY_CRASH` e, da mesma forma que o emissor, executam a finalização da aplicação, a limpeza dos *buffers* e passam para o estado de recuperação.

A segunda etapa procura identificar os membros do grupo que pararam de se comunicar, ou por problemas no sistema de comunicação, ou por problemas nos nodos do sistema, como desligamento forçado por corte de energia. Essa verificação é realizada comparando-se o conteúdo de todos os campos `timemb` (Apêndice B, linha 33), exceto dele próprio, da lista `ckpt_list_group` (Apêndice B, linha 84), com o valor capturado do relógio local do sistema. O campo `timemb` armazena o valor do relógio local do sistema no momento em que a última mensagem (mensagem da aplicação ou de controle) foi recebida do membro remoto correspondente. Caso a diferença dessa comparação ultrapasse um valor previamente estabelecido, o respectivo membro passa a ser suspeito e é submetido a uma sondagem, através do envio de mensagens de tipo `CKPT_MSG_REG_IMLIVE`. O membro suspeito, ao receber essa mensagem, responde com

uma mensagem de tipo `CKPT_MSG_AREG_IMLIVE`, contrariando a suspeita. Caso a resposta à requisição não for recebida em tempo, uma nova requisição é enviada ao processo suspeito. Não havendo resposta, a suspeita foi confirmada e passam a serem executados os mesmos procedimentos "pré-recuperação" apresentados no parágrafo anterior.

Em ambos os casos, o sistema somente passa para o estado de recuperação após ter recebido as respectivas respostas de todos os demais membros do grupo (mensagens do tipo `CKPT_MSG_ANOTIFY_CRASH`). Sendo necessário, são enviadas periodicamente novas requisições até que o objetivo seja alcançado. Essa solução foi implementada dessa forma com o objetivo de detectar o término do efeito de um determinado defeito, como o restabelecimento de um membro que sofreu colapso ou do próprio sistema de comunicação. Esse mecanismo é garantido pois um membro, mesmo em recuperação, sempre responde às notificações de defeito recebidas dos demais membros.

É possível ainda, na manifestação de um defeito, que o mesmo seja detectado por todos os demais membros os quais efetuarão o envio de mensagens de notificação simultaneamente. Da maneira como foi implementado, o detector de defeitos prevê este comportamento e, de modo geral, responde às notificações, mesmo já tendo detectado o defeito e notificado os demais membros. Ou seja, independente do estado do detector, ele sempre responde às notificações de defeito enviadas por outros membros do grupo.

Por outro lado, uma vez detectado um defeito, o detector informa ao sistema de *checkpointing* sobre a existência deste, passando a não mais responder às mensagens de coordenação, como mensagens envolvendo rotação de coordenadores e mensagens de estabelecimento de linhas de recuperação. Dessa forma, não serão mais executados os procedimentos normais (de execução) enquanto o sistema não iniciar os procedimentos de recuperação.

4.6.4 Recuperação

Recuperação em caso de falha não se constitui em objetivo central apresentado pelo autor do algoritmo. No seu trabalho, Cechin (2002) preocupou-se em garantir que sempre existisse uma linha de recuperação que pudesse ser utilizada, no caso de falha, da forma mais rápida possível. No presente trabalho, a recuperação foi implementada baseada em especificações próprias, buscando demonstrar que, a partir de uma linha de recuperação previamente estabelecida, é possível retomar o processamento da aplicação distribuída após a manifestação de falhas.

Foram implementadas duas situações de recuperação: manual e automática. A primeira corresponde a situações em que a ocorrência de uma falha impõe a participação do usuário na recuperação. A segunda possibilita que o usuário fique isento de acionar os mecanismos de recuperação. Em ambas as situações, o processamento normal é interrompido pelo detector de defeitos após a manifestação de alguma falha tolerada e, a menos que a falha tenha ocorrido no início do processamento, existe pelo menos uma linha de recuperação armazenada em disco. A diferença entre a situação normal e a automática é que na primeira será necessário disparar novamente o processo **`CK`** em cada nos nodos do sistema que falharam, informando através de parâmetros a intenção de recuperar a aplicação.

Na recuperação manual, o detector de defeitos não pode acionar a recuperação automaticamente. Dessa forma, após a manifestação da falha, ao ser acionado, o processo **`CK`** identifica os *checkpoints* locais disponíveis para recuperação (lembrando que podem existir dois *checkpoints* caso a falha tenha ocorrido durante o estabelecimento de uma

linha de recuperação). Após a identificação dos *checkpoints*, o processo recupera a lista dos membros do grupo presentes antes da detecção, salva no início do processamento, quando o usuário iniciou a execução de sua aplicação. Essa lista, identificada pela variável `ckpt_list_group`, contém a identificação de todos os membros que estavam ativos, os quais são colocados na situação `CKPT_MEMBER_NOJOIN` (ainda não fazem parte do grupo).

Na recuperação automática, cada membro do grupo conhece previamente os *checkpoints* locais disponíveis, armazenados nas variáveis `CurrCP` ou `PrevCP`. Da mesma forma, também conhece todos os membros do grupo presentes antes do defeito (através da lista `ckpt_list_group`), os quais também são colocados na situação `CKPT_MEMBER_NOJOIN`.

Até esse ponto, o processo **`CK`** conhece quais são os *checkpoints* disponíveis para recuperação e quais eram os membros que estavam em execução antes da manifestação da falha, independente da recuperação ter sido iniciada manualmente ou automaticamente. O próximo passo é executar os procedimentos de estabelecimento do grupo e eleição do coordenador. Essa etapa é trivial, análoga à apresentada na Subseção 4.6.2: primeiro todos os processos “anunciam-se”, de modo que todos passem para a situação `CKPT_MENBER_JOIN` nas respectivas listas locais. Em seguida, quando não houver mais nenhum processo na lista com situação `CKPT_MEMBER_NOJOIN`, o processo com maior IP dispara a rotação de coordenadores e o novo processo coordenador é quem vai propor a linha de recuperação a ser utilizada.

Caso existam dois *checkpoints* locais, o processo coordenador recém eleito propõe a utilização da linha de recuperação correspondente ao *checkpoint* mais recente. Se for obtido acordo, essa será utilizada para recuperação global. Caso não haja acordo, o coordenador propõe a recuperação do segundo *checkpoint*. Se não houver acordo novamente, significa que não existe uma linha de recuperação consistente e o procedimento de recuperação é abortado, situação que corresponde à ocorrência de defeitos antes mesmo do sistema de *checkpointing* ter estabelecido a primeira linha de recuperação, implicando na recuperação a partir do estado inicial, apresentado na Seção 3.3.

Para eleição da linha de recuperação é utilizado o mesmo protocolo de duas fases apresentado anteriormente, com emprego de mensagens `CKPT_MSG_REQ_RESTART` e `CKPT_MSG_AREQ_RESTART` na primeira fase e mensagens `CKPT_MSG_CMT_RESTART` e `CKPT_MSG_ACMT_RESTART` na segunda fase. Cada mensagem carrega o índice correspondente à linha de recuperação que está sendo proposta pelo coordenador e que será comparado com os índices locais recuperados pelos participantes.

Na segunda fase, o processo coordenador, antes de enviar a mensagem `CKPT_MSG_CMT_RESTART`, carrega a aplicação a partir do *checkpoint* acordado e a mantém em estado de espera. O mesmo será efetuado pelos processos participantes quando receberem a mensagem `CKPT_MSG_CMT_RESTART` do coordenador, a qual será respondida com mensagens `CKPT_MSG_ACMT_RESTART`. O coordenador, ao receber todas as mensagens `CKPT_MSG_ACMT_RESTART`, incrementa o índice correspondente à linha de recuperação, envia uma mensagem *multicast* de tipo `CKPT_MSG_START`, desbloqueia o processo da aplicação e reenvia todas as mensagens armazenadas no *checkpoint* com o novo índice associado. Os participantes, ao receberem a mensagem `CKPT_MSG_START`, também incrementam seu índice, desbloqueiam a aplicação e reenviam as mensagens da

aplicação, com o novo índice. A partir desse ponto, a aplicação voltou ao estado de execução existente antes da falha e as mensagens ainda não confirmadas (e que poderiam vir a tornarem-se perdidas) foram retransmitidas. O incremento do índice após a recuperação é necessário para garantir a limpeza dos canais, evitando assim a ocorrência de *livelocks*.

4.7 Da Descrição Formal para o Código-fonte

A fim de atingir os objetivos desta implementação, entendeu-se que seria mandatório seguir, o mais fielmente possível, a especificação do algoritmo proposto. Em seu trabalho, Cechin (2002) especificou o algoritmo utilizando TLA (*Temporal Logic of Actions*) (LAMPORT, 1994), e foram apresentadas as provas que garantem a correção e a consistência do algoritmo. Assim, a especificação do algoritmo deve ser entendida de maneira que, independentemente da linguagem de programação utilizada, leve a uma implementação correta.

Pelo caráter matemático da lógica temporal, uma especificação em TLA permite uma compreensão mais fácil do que especificações em estilo de programas, apesar de se observar tendência contrária no uso destes instrumentos. O principal argumento para essa afirmação é a expressividade da especificação. Enquanto uma especificação em TLA de um determinado problema pode reduzir-se a poucas linhas, a correspondente implementação em uma determinada linguagem de programação pode consumir milhares de linhas de código (LAMPORT, 1994). No presente trabalho, por exemplo, a especificação do algoritmo, apresentada no Anexo A, ocupa cerca de 200 linhas, enquanto a correspondente implementação em C/C++ ocupa cerca de 6000 linhas de código.

Apesar de não existir uma forma de mapeamento direto entre a especificação e a implementação, é possível estabelecer uma relação funcional entre elas. Ao contrário do que se esperava, não foram localizados na literatura subsídios suficientes para construir essa relação, exigindo a adoção de uma metodologia própria.

Diante disso, será apresentada aqui uma forma de mapeamento próprio, buscando representar a implementação do algoritmo a partir da especificação. Esse mapeamento busca apresentar a correspondência entre os elementos da especificação em TLA (variáveis, constantes, ações, etc.) e as partes de código em C/C++ que implementam esses elementos. Não serão apresentados, de ambos os lados, os detalhes de cada abordagem, pois seria equivalente a apresentar todo o código-fonte. Também não serão apresentados os elementos acessórios implementados (aqueles não especificados), pois seria necessário antes especificá-los em TLA para ter sentido a sua apresentação aqui, o que vai além dos objetivos do presente trabalho. Os elementos serão resumidos em seus identificadores e idéia geral; caso for necessário um aprofundamento maior, o leitor pode consultar o código disponibilizado em CD-ROM integrante deste volume (Apêndice J). A descrição da especificação do algoritmo em TLA é apresentada no Anexo A, enquanto as partes mais significativas do código-fonte em C/C++ são apresentadas nos Apêndices A, B, C e D.

Como ponto de partida, serão apresentadas as constantes, variáveis e premissas da especificação com sua correspondente implementação. Em seguida, as ações da especificação terão sua correspondência representada nas funções e procedimentos da implementação.

Constantes:

Especificação	Implementação	Observação
Proc	Lista *ckpt_list_group*.	Lista de todos os identificadores dos processos que formam o sistema distribuído.
Mngr	Campo *manager* da estrutura *ckpt_member*.	Identificação do processo com a função de coordenador.
Deliver	Funções *Sockets* em Linux.	Função de entrega das mensagens à aplicação.

Variáveis:

Especificação	Implementação	Observação
CurrCP	Campo *CurrCP* da estrutura *ckpt_info*.	Último *checkpoint* estabelecido.
PrevCP	Campo *PrevCP* da estrutura *ckpt_info*.	*Checkpoint* anterior ao CurrCP.
CS	Lista *ckpt_SendApp_list*	Estado dos canais ou lista de mensagens potencialmente perdidas.
ME	Campo *ME* da estrutura *ckpt_info*	Estado do coordenador.
Canal	Conexões *sockets* associadas às funções de troca de mensagens	Canais de comunicação onde mensagens são colocadas (na transmissão) e retiradas (na recepção).
H	-	Histórico de mensagens transmitidas – usado apenas para prova formal do algoritmo.

Premissas (*ASSUME*):

Especificação	Implementação	Observação
W	-	*n*-upla das variáveis da descrição – usado apenas na descrição TLA.
MsgType	Campo *msgtype* da estrutura *ckpt_msg_control* (para mensagens de controle) e mensagens interceptadas da aplicação	Conjunto de tipos de mensagens necessárias para a operação do algoritmo.
NN	-	Instante potencial de recebimento de uma mensagem pelo processo receptor – usado apenas para prova formal do algoritmo.
NoMessage	Demais mensagens usadas	Mensagens não tratadas pelo algoritmo.
Message	Mensagens da aplicação, de reconhecimento e para estabelecimento de linhas de recuperação.	Conjunto de mensagens válidas.
ProcS	Processos identificados como não coordenadores na lista *ckpt_list_group*	Conjunto de todos os identificadores de processo, exceto o coordenador.

Ações da especificação:

Especificação	Implementação	Observação
Init	*init_param()* e primeiras linhas da função *main()*	Determina a inicialização de variáveis e corresponde ao estado inicial do sistema.
StoreCS(p,m)	*ckpt_SendApp_list.insert()* em *read_messages_dev(...)*	Inserção das mensagens enviadas pela aplicação no estado do canal (*CS*)
RemoveCS(p,m)	*ckpt_SendApp_list.erase()*	Remoção das mensagens enviadas pela

	em *recv_RxAckAPP_th(...)*	aplicação do estado do canal (*CS*) após recebimento de confirmação de entrega.
Checkpoint(p,k)	*do_ckpt(...)*	Estabelecimento de um ponto de recuperação (*checkpoint*) local – envolve o salvamento da imagem do processo e do estado do canal (*CS*).
MoveCP(p)	*do_garbage_collect(...)*	Cópia do ponto de recuperação atual (*CurrCP*) para o ponto de recuperação anterior (`PrevCP`).
Verify(p,m)	Ações tomadas na função *my_socketcall(...)* e na função *Verify(...)*	Verifica a necessidade de estabelecer um *checkpoint* local antes de entregar a mensagem para a aplicação.
Send(p,q,m)	*Socket sendto()*	Transmissão de uma mensagem ou inserção da mensagem no *Canal*.
Receive(p,q,m)	*Socket recvfrom()*	Recepção de uma mensagem ou remoção da mensagem do *Canal*.
Replay(p,q,m1,m2)	Mescla de *Socket sendto()* e *recvfrom()*.	Recebimento de uma mensagem com a correspondente transmissão da resposta.
SendApp(p) e *SendAppMan*	Ações tomadas nas funções *my_socketcall(...)* e *read_messages_dev(...)*	Transmissão de uma mensagem da aplicação.
RxApp(p) e RxAppMan	Ações tomadas nas funções *my_socketcall(...)*, *read_messages_dev(...)* e *RxAckAPP(...)*	Recepção de uma mensagem da aplicação com envio de mensagem de reconhecimento.
RxAck(p) e RxAckMan	*recv_RxAckApp_th(...)*	Recepção de uma mensagem de reconhecimento.
SendReq	*initGlobalCheckpoint(...)*	Início de uma nova linha de recuperação, com o envio de uma requisição (*REQ*) pelo coordenador via *multicast*.
RxReq	*processGlobalCkpt(...)*	Recebimento e processamento da requisição de uma nova linha de recuperação pelos processos participantes, com envio de resposta ao coordenador (*AREQ*).
RxAckReq	*initGlobalCheckpoint(...)*	Coordenador recebe respostas dos processos participantes à solicitação de uma nova linha de recuperação, envia confirmação da operação (*CMT*) e efetua a coleta de lixo.
RxCmt	*processGlobalCkpt(...)*	Processos participantes recebem confirmação da operação de estabelecimento de linha de recuperação, respondem a essa confirmação (*ACMT*), e efetuam a coleta de lixo.
RxAck	*initGlobalCheckpoint(...)*	Coordenador recebe respostas (*ACMT*) de todos os processos participantes e encerra a operação de estabelecimento de linha de recuperação.

Pela sua boa expressividade, a especificação do algoritmo é sem dúvida um subsídio essencial para garantir a sua correta implementação. Houve diversos momentos antes e durante a implementação em que a especificação foi consultada buscando esclarecer eventuais dúvidas sobre mecanismos do algoritmo e, principalmente, sobre aspectos de

consistência envolvidos. Sugere-se aos leitores interessados em aprofundarem-se no contexto do presente trabalho que, como ponto de partida, façam uma revisão sobre os conceitos básicos de TLA e, a partir de então, entendam o algoritmo a partir da especificação. O trabalho de Cechin (2002) é uma boa fonte de referência para essa abordagem.

4.8 Limitações e Aspectos Pendentes

A implementação do algoritmo e de seus mecanismos auxiliares resultou na elaboração de um sistema ou uma ferramenta de tolerância a falhas, passível de ser utilizada em conjunto com a execução de aplicações distribuídas na plataforma Linux. As premissas de execução em um sistema nativo, sem o uso de infra-estrutura especializada, foram obedecidas. Da mesma forma, foram obedecidas as premissas de transparência para a aplicação. Por último, a ser apresentado no próximo capítulo, o impacto do sistema é mínimo.

Entretanto, a viabilidade da implementação, respeitando as premissas expostas acima, impôs restrições que impediram uma implementação mais genérica do sistema de *checkpointing*. As restrições começam pela plataforma, onde foram utilizadas funções básicas do sistema operacional Linux, como salvamento do estado dos processos e interceptação de chamadas de sistema, impedindo que a mesma implementação possa ser compilada e executada em outras plataformas.

Por outro lado, as restrições impostas às características das aplicações impedem que determinadas aplicações possam funcionar adequadamente. Por exemplo, o sistema não salva o estado das janelas das aplicações que trabalham com interface gráfica, devido a restrições na biblioteca de *checkpointing* utilizada. Da mesma forma, aplicações que utilizam *sockets* orientados a conexão, apesar de serem salvos no momento do estabelecimento de *checkpoints*, não serão restauradas adequadamente, em uma eventual recuperação.

Ainda, sob o ponto de vista das aplicações distribuídas, a implementação exige que sejam compostas por um único processo em cada nodo do sistema. Nesse caso, não se trata de uma restrição da biblioteca mas uma decisão de projeto que visa à simplificação da implementação. Para contornar esta restrição, seria necessário estender a implementação no sentido de passar a manipular grupos de processos que compõem a aplicação num determinado nodo ao invés de manipular um único processo por nodo. Conseqüentemente, para garantir a consistência na recuperação, seria necessário efetuar o salvamento sincronizado de cada processo, uma vez que, além da comunicação por troca de mensagens, poderia existir a comunicação entre processos da aplicação no mesmo nodo por outros mecanismos, como memória compartilhada ou *pipes*.

Mesmo nos mecanismos suportados pela implementação existem restrições pontuais que podem gerar erros na execução. Uma destas restrições está presente no mecanismo de interceptação de mensagens. No momento do envio de uma mensagem pela aplicação, o sistema a intercepta anexando nela informações de controle. Estas informações resumem-se a poucos bytes. Entretanto, há situações em que não é possível a inserção de mais nenhum *byte* devido ao limite no tamanho destas mensagens.

Em Linux, a constante `UIO_MAXIOV`, presente no código-fonte do *kernel*, representa o tamanho máximo da área de dados de um datagrama. Na versão 2.4 do *kernel*, utilizado no presente trabalho, essa constante possui o valor 1024. Isso significa que, caso uma

aplicação envie uma mensagem com tamanho total de 1024 bytes, não será possível anexar as informações de controle necessárias para a garantia de consistência do algoritmo. A solução para esse problema seria implementar um mecanismo adicional de fragmentação de mensagens da aplicação, no mesmo nível da interceptação. Esse mecanismo deveria dividir a mensagem em duas no momento da sua emissão e reconstruí-la, a partir dos fragmentos recebidos, na sua recepção. Esse mecanismo também não foi implementado devido à sua complexidade.

Outro ponto sensível da implementação é a forma de uso dos *buffers* de mensagens. No ambiente de teste do sistema, a ser apresentado no Capítulo 5, não foram detectados problemas devido ao uso desses *buffers*. Entretanto, em condições adversas, como as situações onde a vazão da camada de comunicação não suporta o número de mensagens geradas pela aplicação, esses *buffers* podem crescer a ponto de consumirem toda a memória do sistema, inviabilizando tanto o processamento da aplicação quanto o processamento dos mecanismos do próprio algoritmo.

5 ANÁLISE DE RESULTADOS

Até o momento, foram apresentados os detalhes envolvendo os esforços para a implementação do algoritmo. Construiu-se um sistema de *checkpointing* que pode inserido no sistema operacional para a execução de aplicações distribuídas. A partir do sistema construído, foi efetuada uma análise da sua funcionalidade buscando comprovar os objetivos propostos. O objetivo principal dessa análise foi avaliar a sobrecarga imposta pelo sistema durante execução livre de falhas, de forma a criticar sua usabilidade e seu desempenho.

O presente capítulo tem o objetivo de apresentar essa análise, que consistiu na coleta e comparação de resultados de uma série de execuções de aplicações com e sem o uso do sistema de *checkpointing*. Serão relatadas neste capítulo: as métricas utilizadas, o comportamento das aplicações, os critérios de coleta de informações e a avaliação das mesmas, de acordo com os conceitos apresentados por Jain (1991). Por fim, serão apresentadas conclusões a partir das avaliações efetuadas. Os resultados preliminares da avaliação foram publicados em Buligon; Cechin; Jansch-Pôrto (2005).

5.1 Métricas e Variáveis

Em condições normais de processamento, sem a presença de qualquer serviço de tolerância a falhas, uma aplicação distribuída gasta um determinado tempo desde o início até o término do seu processamento. Esse tempo é usado como referência para avaliar o impacto causado pelo sistema de *checkpointing* implementado nesse trabalho. Em seguida, a mesma aplicação tem anexado o sistema de *checkpointing* e volta então a ser executada. O tempo gasto por ela é medido e comparado com a referência. O resultado dessa comparação indica o impacto causado pelo sistema.

Entretanto, para se obter conclusões desses resultados, é necessário analisá-lo a partir de métricas de desempenho. Das métricas apresentadas por Jain (1991), foram usadas somente aquelas relativas à contenção de recursos: consumo de CPU e quantidade de mensagens geradas na rede. O motivo de usar estas métricas é justificado pelo fato do algoritmo disputar estes recursos (CPU e rede) com a aplicação. O desempenho da aplicação pode ser reduzido devido justamente a esta interferência. Detalhes da utilização desses recursos serão apresentados na Seção 5.2, onde também serão apresentadas as aplicações.

Além da seleção das métricas, faz-se necessário escolher algumas variáveis que são usadas para caracterizar a aplicação e o sistema de *checkpointing*. Estas variáveis correspondem ao tempo médio de execução da aplicação para resolver um determinado problema, ao tempo gasto para o salvamento em disco do estado de um determinado

processo da aplicação, ao intervalo entre cada salvamento e ao número de repetições de cada execução da aplicação para resolver o mesmo problema.

O tempo médio de execução da aplicação é usado como referência para avaliar a sobrecarga de tempo imposto pelo sistema de *checkpointing*. Uma aplicação que executa por um período muito curto pode inviabilizar qualquer avaliação e, conseqüentemente, o uso do sistema. O tempo de execução da aplicação deve ser suficiente para que o sistema de *checkpointing* possa executar todos os seus mecanismos.

O tempo gasto para o salvamento dos *checkpoints* é uma variável que pode determinar, por si só, o impacto no sistema. Devido à implementação não incremental do salvamento de *checkpoints* em disco, e devido ao fato de que durante esse salvamento a aplicação fica bloqueada, é de se esperar que, em situações onde a aplicação utilize grande quantidade de memória, o tempo total gasto com o salvamento dos *checkpoints* represente uma parcela significativa do tempo adicional imposto por esta operação. Da mesma forma, a freqüência com que os *checkpoints* são estabelecidos também tem influência na determinação desse tempo.

Como foram consideradas as médias de tempo gasto em cada situação, foi necessário também determinar o número adequado de repetições, ou seja, quantas vezes é preciso repetir a resolução do mesmo problema pela aplicação para, a partir da média dos tempos coletados, efetuar as comparações e avaliar os resultados.

5.2 Aplicações

Para avaliar a sobrecarga do sistema implementado, identificou-se o interesse na utilização de aplicações que executam por um longo período de tempo, consumindo recursos de CPU, de comunicação e de memória. Além disso, a possibilidade de variação dessas três métricas também possibilitou avaliar o sistema em situações extremas, ou seja, aquelas que envolvem os limites computacionais impostos pelo ambiente de execução.

Visando atender a esses objetivos, foi desenvolvida uma aplicação sintética que utiliza os três recursos citados anteriormente. Essa aplicação pode ser configurada de modo a criar um os perfis ou comportamentos desejados de aplicação. Na prática, os perfis possíveis modelados não resolvem nenhum problema computacional, mas geram uma sobrecarga bem definida no ambiente de execução.

A aplicação é composta basicamente por duas partes: inicialização e execução. Na inicialização, a aplicação aloca recursos de memória de acordo com os parâmetros informados. Durante o processamento, a aplicação permanece em um laço (*loop*) por um número definido de repetições, modelado pelo número de mensagens enviado pela aplicação (“faça até que *n* mensagens forem enviadas”). Nesse laço, a aplicação pode efetuar três operações, na seguinte ordem, conforme Figura 5.1:

- efetuar processamento durante um determinado período de tempo (linha 04 a 06);
- enviar uma mensagem e, dependendo da configuração, aguardar ou não pela resposta (linha 07 a 12);
- esperar por um tempo especificado (linha 14).

```
01  pBuffer = (char *)malloc(nBytesMemory);
02
03  while(msgSend<nMessages) {
04       while(nTimeCPU){
05          do_something();
06       }
07       message = (char *)malloc(ramdom(nLengthMsg));
08       send_message(message,netDestination);
09       free(message);
10       if(waitForAck) {
11          message = receive_ack(netDestination);
12       }
13       wait(nTimeWait);
14       msgSend++
15  }
16  free(pBuffer)
17  exit(0);
```

Figura 5.1: Operações da aplicação sintética.

Com a escolha de valores adequados para configurar as três operações citadas anteriormente, foi possível determinar, através da observação por ferramentas de monitoramento do sistema operacional, tais como Top e Ntop, a sobrecarga imposta aos recursos de CPU e de rede no ambiente de execução. Assim, a conjugação dos três fatores determinou valores médios que caracterizaram cada perfil de aplicações em termos de consumo de CPU e consumo de banda de rede para troca de mensagens. O término da aplicação sintética foi determinado a partir da freqüência de mensagens geradas. Esse parâmetro corresponde ao número total de mensagens geradas por processo da aplicação, equivalendo a um tempo de execução da aplicação suficiente para as observações necessárias. O tempo especificado foi de 10 minutos e o número total de mensagens equivalente a esse tempo foi determinado pela taxa média de mensagens.

A aplicação sintética desenvolvida é formada por um processo servidor, responsável pelo recebimento das mensagens geradas e, se necessário, pelo envio de mensagens de confirmação. Os demais processos são processos clientes e possuem a estrutura representada na Figura 5.1. A aplicação inicia o processamento quando o processo servidor entra em execução e encerra quando todos os processos cliente tiverem gerado o número especificado de mensagens (equivalente ao tempo de execução desejado).

Tabela 5.1: Perfil das aplicações

Aplicação	Consumo de CPU (Top v2.0.7)	Consumo de rede (Ntop v2.1.0)	Mensagens geradas (condição para término)
NET	0.0% a 2.1%	0.8 Mbps a 0.9 Mbps	60000 (com *ack*)
BAL	4.5% a 11.8%	0.5 Mbps a 0.6 Mbps	40000 (com *ack*)
CPU	78.4% a 98.3%	0.1 Mbps a 0.2 Mbps	10000 (com *ack*)
LOW	0.0%	0.3 Mbps a 0.4 Mbps	30000 (com *ack*)
HIGH	0.0% a 0.8%	7.5 Mbps a 8.0 Mbps	600000 (sem *ack*)

Foram configurados cinco perfis de aplicações, identificadas pelos nomes NET, BAL, CPU, LOW e HIGH. O comportamento dessas aplicações, no que diz respeito à CPU e rede, é apresentado na Tabela 5.1.

A aplicação NET faz pouco uso do recurso CPU, mas apresenta uso intenso da rede. No laço de processamento, grande parte do tempo de execução é direcionada para o envio de mensagens e recebimento de suas respostas. A aplicação BAL representa o equilíbrio entre o consumo de rede e CPU, dividindo igualmente o tempo de execução entre o processamento e a troca de mensagens. A aplicação CPU faz uso intenso de CPU e pouco da rede, reservando a maior parte do tempo para o processamento. A aplicação LOW não efetua qualquer processamento e o tempo de processamento é usado na troca de mensagens seguido de um período de espera (linha 13 da Figura 5.1).

A aplicação HIGH possui um comportamento atípico para aplicações distribuídas convencionais e foi concebida com propósitos puramente de testes. O objetivo foi o de estudar situações extremas de uso de rede, uma vez que ela faz uso intenso desse recurso enviando, repetidamente, mensagens sem aguardar pelo recebimento de resposta. Não é efetuado processamento e o tempo disponível é usado para o envio de mensagens. Esse perfil foi idealizado para avaliar o comportamento do algoritmo em situações de extrema geração de mensagens pela aplicação.

Em todos os perfis, as mensagens geradas possuem tamanho variável, buscando modelar o tráfego real. O tamanho médio de cada mensagem foi estabelecido em 100 bytes, variando entre 1 e 200 bytes. Esse parâmetro também pode ser ajustado de acordo com o perfil desejado.

As medições de consumo de rede, coletadas com a ferramenta Ntop, estão listadas no Apêndice E. Nesses gráficos estão representados os picos de consumo de rede de cada aplicação durante uma execução com e sem o uso do sistema de *checkpointing*. Os picos identificados pelo símbolo “+” correspondem à execução das aplicações com a utilização o sistema de *checkpointing*. A avaliação destes resultados será apresentada no decorrer do presente capítulo.

A quantidade de memória alocada não influencia o comportamento das aplicações pois essa área de memória não é lida ou escrita pela aplicação durante a execução. Logo, não tem relação com o consumo de CPU. Ela tem o propósito exclusivo de determinar o tamanho do *checkpoint* a ser estabelecido em disco e, como não foram empregados mecanismos de salvamento incremental, o tempo de salvamento corresponde sempre ao pior caso, ou seja, aquele em que toda a memória alocada pelo processo precisa ser salva. Para as avaliações do presente trabalho, foram estabelecidas três situações de alocação de memória pelas aplicações: 1) nenhuma memória adicional é alocada, sendo que o tamanho do *checkpoint* vai corresponder basicamente ao código executável do processo da aplicação; 2) alocação de 5 Mbytes de memória e 3) alocação de 15 Mbytes de memória. Nessas duas últimas situações, o tamanho do *checkpoint* vai corresponder ao tamanho de memória alocado mais o tamanho do código executável e de eventuais variáveis alocadas pelo processo.

5.3 Definição das Execuções e da Coleta de Dados

A partir da definição do perfil das aplicações usadas para avaliar o algoritmo, foi necessário determinar a forma como seriam realizadas as execuções e como seria efetuada a coleta de informações a serem avaliadas.

O ambiente de execução das simulações foi composto por um conjunto de 6 computadores ligados em rede. A configuração básica das máquinas foi composta por processadores Pentium IV, com velocidade de 2.4Ghz, 512 Mbytes de memória RAM, disco IDE Ata 133 de 40Gbytes e placa de rede de 100 Mbits. A interligação em rede foi realizada com um equipamento `hub` com velocidade de 100 Mbits, isolando-a de qualquer outra sub-rede para evitar interferência externa nos recursos de comunicação. Da mesma forma, somente os serviços básicos do sistema operacional foram habilitados, evitando também qualquer interferência nos recursos de CPU por eventuais serviços desnecessários. Todos os equipamentos foram instalados com sistema operacional Linux, na versão 2.4 do *kernel*. O sistema de arquivos utilizado foi o `ext2`.

A aplicação distribuída foi composta por processos que executaram cada um em uma máquina, tendo um deles o perfil de servidor e os outros o perfil de clientes. Esses processos executaram um determinado processamento e comunicaram-se através de troca de mensagens, de acordo com os perfis estabelecidos. Em cada máquina foi executado, opcionalmente, o sistema de *checkpointing*.

As medidas de consumo de CPU foram coletadas a partir da execução de cada perfil de aplicação, em todas as máquinas, com o posterior estabelecimento da média. Os valores de sobrecarga de CPU foram coletados a partir da ferramenta Top versão 2.0.7.

Para as medições de sobrecarga no serviço de comunicação, foi interligada na mesma rede uma sétima máquina onde foi executada a aplicação Ntop versão 2.1.0. Essa aplicação possui características de um *sniffer* e gera gráficos de utilização da rede. Esse computador também foi utilizado para a execução do processo **`MONITOR`**, responsável pela configuração, execução e monitoração do sistema de *checkpointing*. Dessa forma, as seis máquinas que rodaram o sistema não sofreram interferência do mecanismo monitor.

As aplicações distribuídas foram executadas em três diferentes situações: 1) sem sistema de *checkpointing*; 2) com o sistema de *checkpointing* instalado mas sem ser utilizado; 3) com a utilização do sistema de *checkpointing*.

A primeira situação teve o objetivo de coletar os tempos gastos para a execução das aplicações sem nenhum serviço adicional instalado e correspondeu à execução normal da aplicação. A segunda teve o objetivo de avaliar a interferência passiva da presença do sistema de *checkpoint* no ambiente computacional. Nessa situação, somente o módulo de *kernel* estava habilitado e, conseqüentemente, todas as mensagens da aplicação foram interceptadas, apesar de não terem sido submetidas a nenhum processamento. Na terceira situação o sistema de *checkpointing* foi ativado, e as informações coletadas serviram para sua avaliação.

Para obter-se valores estatisticamente corretos, foi necessário definir o número de repetições de execuções necessária em cada perfil da aplicação. Como os tempos de cada execução nunca foram iguais, decidiu-se trabalhar com médias e, portanto, foi necessário estabelecer um número adequado de repetições de cada execução, para formar essa média.

Para estabelecer o número de repetições, foi utilizado o conceito de intervalo de confiança (JAIN, 1991). Através desse método, estabeleceu-se um percentual de confiança sobre a média, representando quanto a média da amostra estaria próxima da média da população, uma margem de erro e um número inicial de amostragens. A partir dessas informações, calculou-se quantas repetições seriam necessárias para garantir o intervalo de confiança estabelecido. No Apêndice F, estão relacionados os números de

repetições necessárias para cada perfil de aplicação, a partir de um intervalo de confiança de 95%, com margem de erro de 2,5% e de uma amostragem de 4 repetições para cada perfil de aplicação. Nesse mesmo apêndice estão definidos os diferentes cenários de execução de cada aplicação.

A última informação refere-se ao intervalo entre estabelecimentos de linhas de recuperação pelo sistema de *checkpointing*. Tendo em vista que o tempo médio de execução de cada aplicação idealizada foi em torno de 10 minutos, definiu-se que o tempo correspondente ao intervalo de estabelecimento das linhas de recuperação seria de 30 segundos. A exemplo de Yang *et al.* (2004), a escolha do intervalo foi arbitrária. O valor deveria ser suficiente para que, durante a execução da aplicação, o sistema de *checkpointing* pudesse exercitar todos os mecanismos implementados, o que foi comprovado a partir de mecanismos de depuração implementados através de mensagens de sinalização registradas em arquivos de *logs*.

5.4 Avaliação dos Resultados

Nessa seção, os resultados apresentados serão avaliados, buscando estabelecer conclusões sobre o comportamento do algoritmo e da implementação realizada no presente trabalho. O detalhamento dos resultados obtidos nas execuções está reunido nos Apêndices G, H e I. No Apêndice G, estão relacionados os tempos médios gastos por cada aplicação, nos diferences cenários de execução. No Apêndice H, estão resumidos os valores médios de mensagens salvas por *checkpoint* e por execução da aplicação. No Apêndice I, estão relacionadas quantidades médias de mensagens interceptadas geradas pela aplicação, e de mensagens de controle geradas pelo sistema de *checkpointing*.

5.4.1 Fator de interceptação

O primeiro fator a ser avaliado diz respeito à interceptação de mensagens em espaço de *kernel*. Fontoura (2002) apresentou em seu trabalho um estudo sobre o impacto do uso dessa técnica e concluiu que, no pior caso, a sobrecarga imposta foi de 6,89%, o que foi considerado aceitável para a configuração dos equipamentos utilizados.

A avaliação dos resultados das execuções das aplicações, com sobrecarga passiva do módulo de *kernel* (mecanismo de *checkpointing* não ativado), equivalente ao experimento de Fontoura, permitiu determinar o impacto do uso dessa técnica sobre o ambiente de execução utilizado. A Tabela 5.2 apresenta a síntese dos resultados obtidos.

Tabela 5.2: Sobrecarga imposta pela interceptação

Aplicação	Tempo médio sem interceptação	Tempo médio com interceptação	Sobrecarga
NET	621543	621747	0,033 %
BAL	653874	654555	0,104 %
CPU	614535	615087	0,090 %
LOW	613803	614298	0,081 %
HIGH	613287	614943	0,270 %

Confirmando as conclusões de Fontoura, o uso de interceptação impôs um impacto desprezível ao sistema. O pior caso, equivalente à execução da aplicação HIGH, cujo

perfil corresponde à intensa geração de mensagens, apresentou uma sobrecarga de 0,27%. Este valor baixo incentivou o uso dessa técnica.

A diminuição da sobrecarga, se comparada com os resultados de Fontoura, deveu-se principalmente à configuração dos equipamentos. A infra-estrutura de rede, por exemplo, passou de uma vazão de 10Mbits para 100Mbits. Já a velocidade do processador passou de 200MHz para 2.4GHz (2400MHz). Isto representa um aumento de 10 vezes na velocidade da rede e de 12 vezes e na velocidade de CPU. Outro fator a ser considerado foi o conjunto de perfis das aplicações usadas, que não foram iguais nos dois trabalhos.

5.4.2 Fator de *checkpoint*

Uma das principais informações obtidas das execuções foi a média de tempo gasta para o estabelecimento de *checkpoints*. Conforme apresentados anteriormente, os *checkpoints* foram salvos em disco e a aplicação foi bloqueada durante o salvamento. Portanto, os tempos gastos nessa operação tiveram relação direta na sobrecarga imposta.

Foram estabelecidos três tamanhos diferentes de *checkpoint* através da quantidade de memória alocada pelas aplicações: 0.1 Mbytes, 5 Mbytes e 15 Mbyes. Na Tabela 5.3, é apresentada a média dos tempos de estabelecimento de cada tamanho de *checkpoint*. Na Figura 5.2, é apresentado um gráfico comparativo dos valores listados.

Tabela 5.3: Tempo gasto no estabelecimento dos *checkpoints*

Aplicação	0.1 MBytes	5 Mbytes	15 Mbytes
NET	0,4 ms	60,7 ms	182,1 ms
BAL	0,7 ms	57,2 ms	202,5 ms
CPU	1,5 ms	59,3 ms	182,8 ms
LOW	0,6 ms	50,8 ms	168,2 ms
HIGH	0,8 ms	55,8 ms	181,0 ms

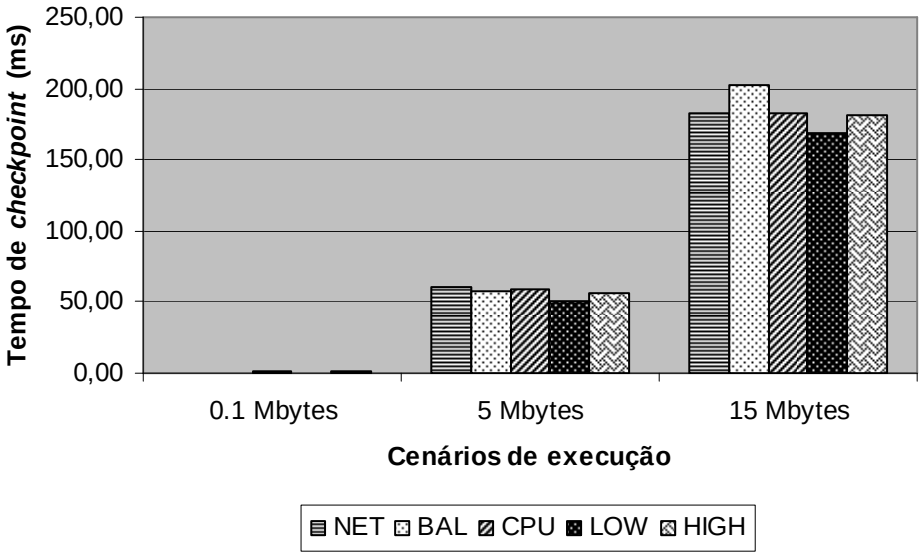


Figura 5.2: Comparação no tamanho dos *checkpoints*.

O algoritmo prevê que os *checkpoints* devem conter, além do estado dos processos, as mensagens que poderão vir a tornarem-se perdidas, ou seja, aquelas enviadas pela aplicação mas que o sistema de *checkpointing* ainda não recebeu confirmação de

recebimento. Dessa forma, o tempo gasto para salvar essas mensagens já está contabilizado nos valores apresentados na Tabela 5.3. Os experimentos mostraram que o tempo gasto exclusivamente para o salvamento dessas mensagens é muito pequeno, se comparado com o tempo gasto para o salvamento do estado do processo. A Tabela 5.4 apresenta o número médio de mensagens geradas pela aplicação em cada um dos perfis de aplicações. Associado a essa média é ilustrado o valor percentual que ela representa em relação ao volume de dados dos *checkpoints* estabelecidos em cada um dos perfis das aplicações. Por exemplo, do *checkpoint* salvo da aplicação NET, na situação em que ela aloca 0.1Mbytes de memória, 6,8% equivalem às mensagens e 93,2% (o restante) equivale à imagem básica do processo. Para efeito desses cálculos, considerou-se que, cada para mensagem salva, são gravados 1056 bytes em disco (tamanho dos registros).

O número ilustrado entre parêntesis associado a cada perfil das aplicações representa a taxa média de mensagens geradas pela aplicação no canal de comunicação (rede) durante sua execução. Essa taxa é obtida dividindo-se o número total de mensagens geradas (Tabela 5.1) pelo seu tempo médio de execução com o sistema de *checkpointing* habilitado e a aplicação gerando *checkpoints* pequenos (Cenário 3 – Apêndice G). Por exemplo: a taxa de 957 msg/s da aplicação NET equivale à soma das mensagens da aplicação (60000 mensagens + 60000 *acks* = 120000), vezes o número de processos clientes (5 processos), dividido pelo tempo médio de execução da aplicação. Ou seja: (120000 x 5) ÷ 627,087s = **957** msg/s.

Tabela 5.4: Taxa de mensagens salvas em cada *checkpoint*

Taxas	NET (957 msg/s)	BAL (603 msg/s)	CPU (160 msg/s)	LOW (483 msg/s)	HIGH (4892 msg/s)
0.1Mbytes	6.9 msg 6.8%	12.8 msg 11.9 %	2.1 msg 2.1 %	2.3 msg 2.4 %	4.1 msg 4.1 %
5Mbytes	10.4 msg 0.2 %	12.8 msg 0.3 %	2.9 msg 0.06 %	2.9 msg 0.06 %	2.3 msg 0.05 %
15Mbytes	11.4 msg 0.08 %	13.6 msg 0.1 %	14.2 msg 0.1 %	2.1 msg 0.02 %	3.1 msg 0.02 %

De acordo com os valores apresentados na Tabela 5.4, conclui-se que o volume de dados equivalente às mensagens é muito pequeno, correspondendo, no máximo, a 11.9% da quantidade total de informações serem salvas em memória estável.

Levando-se em consideração que o período entre *checkpoints* é de 30 segundos e que cada cenário da aplicação executa por um tempo médio de 10 minutos, pode-se concluir que em cada execução serão estabelecidos, em média, 20 *checkpoints*. Dessa forma, é possível calcular o tempo gasto exclusivamente com o salvamento dos *checkpoints* durante a execução das aplicações, bastando multiplicar os tempos da Tabela 5.3 por 20.

5.4.3 Fator de mensagens geradas

O funcionamento dos mecanismos do algoritmo impõe o uso de mensagens adicionais, referenciadas nesse trabalho como mensagens de controle. Estas mensagens compartilham os mesmos recursos de rede disponíveis para a aplicação. É adequado, portanto, avaliar a sobrecarga imposta por estas mensagens no ambiente de execução.

A Tabela 5.5 resume a média de mensagens geradas pelas aplicações e a média de mensagens de controle necessárias para o funcionamento do algoritmo. Os resultados completos a partir dos quais essas médias foram extraídas estão no Apêndice H.

Tabela 5.5: Número de mensagens geradas

Aplicação	Controle [4]	Sobrecarga
293477	792	0,27 %

Basicamente, o sistema de *checkpointing* impõe a geração de uma mensagem de controle para cada mensagem gerada pela aplicação, devido à necessidade de confirmar o recebimento das mensagens da aplicação. Além das mensagens de confirmação, o algoritmo gera mensagens de controle para execução dos demais mecanismos, tais como estabelecimento de linhas de recuperação, rotação de coordenadores e detecção de defeitos. Essas mensagens correspondem a 0,27% do total de mensagens contabilizadas.

As mensagens de sistema geradas, apesar de representarem aproximadamente o mesmo número de mensagens da aplicação, não geram sobrecarga equivalente nos recursos de rede. Esse comportamento pode ser observado na Tabela 5.6, onde é apresentado o consumo de recursos de rede em cada cenário.

Tabela 5.6: Sobrecarga das mensagens de controle

Aplicação	Sistema original	Com *checkpointing*	Sobrecarga
NET	0,85 Mbps	1,35 Mbps	58,82 %
BAL	0,55 Mbps	0,85 Mbps	54,54 %
CPU	0,15 Mbps	0,25 Mbps	66,67 %
LOW	0,35 Mbps	0,55 Mbps	57,14 %
HIGH	7,75 Mbps	8,75 Mbps	12,90 %

A explicação para essa relação deve-se ao fato de que as mensagens de controle são pequenas, compostas por poucos bytes (36 bytes), e possuem tamanho fixo. Dessa forma, para cada mensagem da aplicação, independentemente do seu tamanho, é gerada uma pequena mensagem de controle. Mensagens grandes, ao trafegarem na rede, consomem mais recursos comparativamente ao tráfego de mensagens pequenas. A prova disso aparece nos resultados observados nas execuções. Para aplicações que trabalham com mensagens de confirmação (NET, BAL, CPU e LOW), a sobrecarga foi superior a 50%, uma vez que a aplicação também gera um grande número de mensagens pequenas para implementar mensagens de resposta – *acks*. Já na aplicação HIGH, onde a aplicação não gera mensagens de confirmação, resumindo-se a gerar mensagens proporcionalmente grandes, a sobrecarga imposta pelas mensagens de controle não passou de 13%, apesar de duplicar o número de mensagens trafegando na rede.

[4] Foram omitidas as mensagens de confirmação de recebimento das mensagens da aplicação, geradas pelo algoritmo. Como para cada mensagem da aplicação é gerada uma mensagem de confirmação, seria equivalente dizer que o número de mensagens geradas pelo algoritmo faz, no mínimo, dobrar o número de mensagens geradas no canal de comunicação durante a execução da aplicação.

5.4.4 Sobrecarga do sistema de *checkpointing*

Após terem sido mostrados separadamente os fatores de sobrecarga, serão apresentados os resultados observados das execuções das aplicações com o sistema de *checkpointing* em funcionamento. A Tabela 5.7 resume todos os fatores de sobrecarga, nos diferentes cenários e sem a ocorrência de falhas. Essas informações estão listadas no Apêndice G.

Tabela 5.7: Sobrecarga do sistema de *checkpointing*

Tamanho do *Checkpoint*	Aplicação	Tempo de execução em *ms*		Sobrecarga
		Sistema original	Com *Checkpoint*	
0.1 Mbytes	NET	621543	627087	0,89%
	BAL	653874	663012	1,40%
	CPU	614535	625162	1,73%
	LOW	613803	619852	0,98%
	HIGH	613287	617249	0,65%
5 Mbytes	NET	621543	643374	3,51%
	BAL	653874	683664	4,56%
	CPU	614535	652570	6,19%
	LOW	613803	628940	2,47%
	HIGH	613287	627913	2,38%
15 Mbytes	NET	621543	680022	9,41%
	BAL	653874	718721	9,92%
	CPU	614535	685990	11,63%
	LOW	613803	671948	9,47%
	HIGH	613287	669678	9,19%

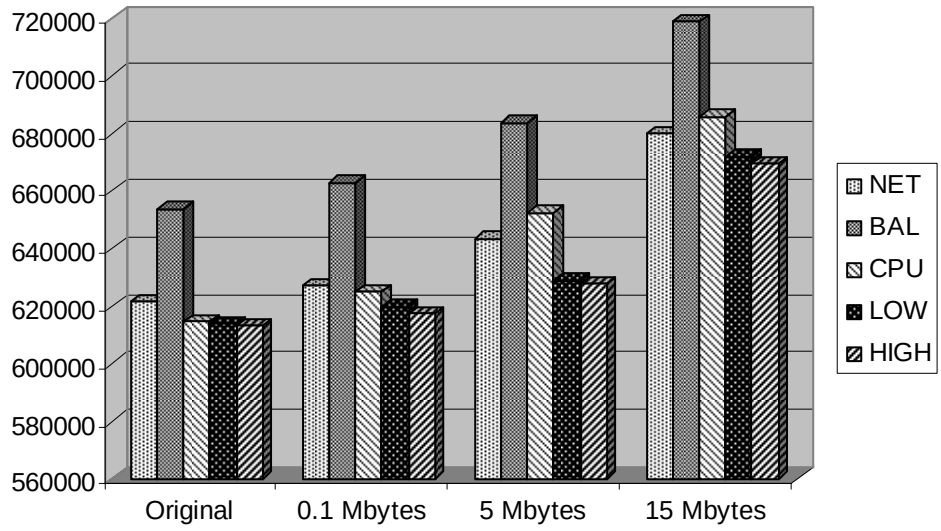


Figura 5.3: Sobrecarga do sistema de *checkpointing*

Observa-se, na Figura 5.3, que a sobrecarga imposta pelo sistema de *checkpointing* cresce na medida em que aumenta o tamanho de memória alocada pelo sistema. Essa

proporção indica que o principal fator que determina a sobrecarga é o estabelecimento dos *checkpoints*, envolvendo o salvamento de informações em disco.

Entretanto, observou-se também que o tempo contabilizado exclusivamente para o salvamento em disco dos *checkpoints* não corresponde necessariamente à maioria da sobrecarga imposta. Multiplicando-se os tempos apresentados na Tabela 5.3 por 20, que é a média de *checkpoints* estabelecidos, obtêm-se valores pequenos de tempo em relação à diferença entre os tempos gastos pela execução das aplicações com e sem *checkpointing*. Essa aparente discrepância é explicada pelo modo como o sistema de arquivos do Linux manipula as operações sobre o sistema de arquivos. É característica do Linux manter os dados manipulados em *cache* para liberar a aplicação de eventuais bloqueios durante operações envolvendo arquivos (MITCHEL; OLDHAM; SAMUEL, 2001). Os dados correspondentes somente serão materializados (*flush*) em disco posteriormente, em *background* pelo sistema de arquivos, e essa materialização consome recursos do sistema operacional, tornando-o mais lento, refletindo em retardos na execução das aplicações e serviços do sistema operacional.

Por outro lado, pode-se concluir que os demais mecanismos do algoritmo causam pouca sobrecarga, não passando de 2% no caso do cenário em que a aplicação não aloca memória adicional (0.1 Mbytes). Essa conclusão indica a baixa interferência do algoritmo de *checkpointing*, quando em execuções livres de falhas, comprovando a teoria defendida na proposta do algoritmo.

5.5 Tempo de recuperação

A recuperação das aplicações foi testada a partir da simulação de falhas como o cancelamento forçado dos processos da aplicação (*kill*) e isolamento momentâneo de um dos equipamentos através da desconexão do cabo de rede. Essas falhas acionaram os mecanismos automáticos de recuperação, e os tempos coletados a partir de amostras determinaram que, nos casos de recuperação com sucesso, os tempos foram relativamente pequenos, chegando no máximo a 20 ms nos cenários em que a aplicação manipula grande quantidade de memória (15 Mbytes). Parte do tempo é gasto com a leitura do estado do processo para a memória e parte com o sincronismo entre processos, necessário à recuperação. Detalhes dos procedimentos envolvidos na recuperação das aplicações foram apresentados na Seção 4.6.4.

A detecção da linha de recuperação foi extremamente rápida, uma vez que bastou a cada processo efetuar uma leitura em disco para descobrir quais os *checkpoints* que estavam presentes, seguido da coordenação para decidir qual o *checkpoint* a ser utilizado. Sob o ponto de vista da aplicação, pode-se afirmar que o tempo de computação perdido devido ao retorno da computação foi, no máximo, o tempo correspondente ao intervalo entre o estabelecimento das linhas de recuperação. No caso do presente trabalho, 30 segundos. Essa afirmação é pertinente em situações onde o próprio sistema de *checkpointing* efetuou automaticamente o retorno da computação, sem necessidade de intervenção do usuário.

Em caso de necessidade de intervenção, o tempo de computação perdido pela aplicação corresponde ao tempo necessário para o usuário disparar manualmente os procedimentos de recuperação, após a manifestação da falha, mais o tempo gasto pelos mecanismos de recuperação. Portanto, nessa segunda situação, não é possível precisar quanto tempo de computação seria perdido pela aplicação.

Assim, o melhor caso equivale àquele em que houve manifestação temporária de uma falha durante ou exatamente após o estabelecimento de uma linha de recuperação, e que o próprio sistema de *checkpointing* recuperou automaticamente a aplicação. Experimentos mostraram que, nesses casos, a aparência percebida pelo usuário é como se a aplicação não tivesse parado de executar, dando a impressão que ela simplesmente “pensou” por um pequeno instante.

No que diz respeito à consistência pós-recuperação das mensagens, observou-se que as aplicações não “perceberam” que as mensagens perdidas foram retransmitidas pelo sistema de *checkpointing* e não pela própria aplicação. Da mesma forma, o comportamento esperado pela aplicação no que diz respeito à manipulação de mensagens foi confirmado nos casos de recuperação, ou seja, foi consistente com a operação sem falhas/recuperação.

Foram feitos experimentos buscando comprovar a possibilidade de recuperação consistente da aplicação sintética, utilizada no presente trabalho. Esses experimentos concentraram-se em situações hipotéticas, as quais obteve-se sucesso na recuperação. Entretanto, não foram feitos testes exaustivos, sobre condições específicas, e com aplicações mais complexas para comprovar completamente a exatidão dos mecanismos implementados. Sugere-se, como continuação do presente trabalho, seguir essa direção.

6 CONCLUSÃO E TRABALHOS FUTUROS

Foram descritos, neste trabalho, os passos para a implementação transparente de recuperação por retorno baseada em *checkpointing* em sistemas distribuídos assíncronos, sem recorrer ao uso de infra-estrutura especializada, e os resultados obtidos com este protótipo.

Este trabalho diferencia-se por usar um algoritmo provado formalmente e pelo objetivo de fornecer *checkpointing*/recuperação de forma transparente aos programadores das aplicações. Essa transparência refere-se a não exigir a alteração do código-fonte das aplicações nem do sistema operacional. Adicionalmente, a forma como o sistema de *checkpointing* foi implementado (não exigindo nenhuma infra-estrutura especializada para garantir a execução dos serviços) facilita sua inserção no sistema operacional e seu emprego pelos usuários do sistema, além de comprovar o baixo custo da implementação.

Com relação à especificação formal, observou-se ser uma ferramenta poderosa, pois auxilia significativamente a tarefa de implementação, uma vez que todos os detalhes relacionados estão descritos, bastando serem implementados. Além disso, com um algoritmo que tenha sido provado formalmente, pode-se utilizar, com segurança, qualquer tipo de implementação para os mecanismos acessórios (aqueles que não fazem parte da especificação), como a detecção de defeitos, a rotação de coordenadores ou a recuperação propriamente dita da aplicação distribuída.

Apesar dos autores de bibliotecas de *checkpointing* de uso geral disponíveis afirmarem a simplicidade do uso de seus códigos, a realidade foi outra. A reutilização não foi trivial e demandou um estudo aprofundado de cada biblioteca de maneira que fosse possível identificar as características e limitações de cada uma. Percebeu-se que o salvamento de estado não é uma tarefa trivial. Além do estado básico de cada processo, que está em memória e nos registradores, é necessário salvar e re-estabelecer o estado das conexões de comunicação entre os processos da aplicação e as mensagens que poderão vir a se tornar perdidas. Esta tarefa não foi implementada em nenhuma das bibliotecas estudadas, devido à complexidade exigida. Foi, portanto, necessário delimitar o problema, restringindo a implementação à manipulação de comunicação não orientada a conexão (UDP), que é menos complexa. Isso resultou em limitações na disponibilidade de aplicações, tendo sido desenvolvidas aplicações sintéticas específicas como alternativa para exercitar os possíveis gargalos identificados em diferentes cenários de execução.

No caso específico da biblioteca CRAK utilizada, acredita-se que seja possível estendê-la com otimizações e com a implementação de novos recursos, com o objetivo de aumentar a compatibilidade com as aplicações e com novas versões do *kernel* do Linux. Como exemplos de recursos adicionais, pode-se relacionar a recuperação de conexões *sockets* orientados à conexão (TCP), a recuperação de estado das janelas para aplicações gráficas e, através da incorporação de técnicas de recuperação baseada em *logs*, a

recuperação de eventos gerados por aplicações que interagem com o mundo externo. Dessa forma, um número maior de aplicações poderá ser beneficiado com os recursos de tolerância a falhas deste trabalho.

No caso específico do Linux, descobriu-se que a interceptação de chamadas de sistema é uma tarefa simples e de baixo impacto no desempenho. Este mecanismo foi fundamental para se obter uma implementação do algoritmo de recuperação em que a aplicação não necessitasse ser alterada. A contrapartida da utilização de técnicas de interceptação foi a necessidade de implementação de parte do sistema de *checkpointing* em espaço de sistema (*kernel*), o que compromete a portabilidade.

A partir do trabalho de implementação aqui realizado e das características de desempenho observadas na execução dos experimentos, foi possível obter diversas conclusões sobre recuperação por retorno e sobre o próprio algoritmo.

Os resultados mostraram que o principal fator que interferiu no tempo de execução das aplicações que fizeram uso do sistema de *checkpointing* foi o tamanho do *checkpoint* estabelecido, o que corresponde ao tamanho de memória alocada pela aplicação e pelas mensagens salvas. A sobrecarga aumenta linearmente, na mesma proporção em que aumenta o tamanho dos *checkpoints*. Por outro lado, a sobrecarga imposta no tempo de execução pelos mecanismos de coordenação entre os processos foi mínima. Para reduzir a sobrecarga causada pelo tamanho dos *checkpoints,* técnicas de otimização podem ser investigadas e implementadas como trabalhos futuros, uma vez que neste trabalho não houve essa preocupação. Nesse sentido, é conveniente que técnicas de otimização não acarretem problemas cuja complexidade de solução cause um impacto demasiado na etapa de recuperação dos *checkpoints*.

Também foi identificada uma interferência muito pequena causada pela interceptação das mensagens da aplicação. No pior caso, em cenários de alta taxa de troca de mensagens pela aplicação, a sobrecarga foi de 0,27%, o que incentiva o uso dessa técnica. No tocante à sobrecarga imposta pelo algoritmo, apesar de não ter sido aplicada nenhuma otimização nos gargalos, a sobrecarga imposta foi bastante aceitável, não passando de 2%. Este valor é bastante inferior aos resultados observados em trabalhos encontrados na literatura, onde mesmo com otimizações nas operações, os resultados investigados mostraram uma sobrecarga, em condições ideais e *checkpoints* pequenos, entre 3% e 5,8%. Acredita-se que a sobrecarga poderia ser ainda menor se tivessem sido implementadas extensões visando reduzir o tempo gasto no salvamento dos *checkpoints* em memória estável, como salvamento incremental por exemplo.

A partir da observação do consumo dos recursos do sistema, utilizando as ferramentas TOP e NTOP, foi possível obter informações que indicaram separadamente a sobrecarga imposta no processamento e no meio de comunicação. Foi possível concluir que o algoritmo é sensível à utilização de CPU pelas aplicações, mas sua influência é menos significativa quando a taxa de transmissão de mensagens aumenta.

Todos os experimentos foram conduzidos com aplicações executando em média durante 10 minutos e com um intervalo de estabelecimento de linhas de recuperação de 30 segundos. Obviamente esses valores podem não corresponder a valores encontrados em situações utilizadas na prática, mas foram adequados o suficiente para estimar a sobrecarga imposta pelo sistema de *checkpointing* em execuções livres de falhas, possibilitando, assim, a avaliação do algoritmo e da sua implementação.

Como continuação deste trabalho, além das extensões sugeridas anteriormente para serem implementadas na biblioteca CRAK e das otimizações visando diminuir os gargalos no salvamento dos *checkpoints* em memória estável, sugere-se a avaliação do comportamento do sistema de *checkpointing* quando executado em ambientes de alto desempenho, como aqueles providos de multiprocessamento e com taxa de transmissão de mensagens pela rede na ordem de Gbits/s. Ainda, visando direcionar a avaliação exclusivamente para o tempo gasto com o salvamento dos *checkpoints*, sugere-se a utilização de meios de armazenamento ultra rápidos, como os recentes SSDs (*Solid State Disks*), cujas velocidades de armazenamento ultrapassam a taxa de 4GB/s, contra os atuais 100MB/s dos discos IDE utilizados neste trabalho.

REFERÊNCIAS

ALVAREZ, G. A.; CRISTIAN, F. CESIUM: Testing Hard Real-Time Dependability Properties of Distributed Protocols. In: IEEE INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED REAL-TIME DEPENDABLE SYSTEMS, 3., 1997. **Proceedings…** [S.l.:s.n.], 1997.

BAR, M. **Linux Internals**. New York: McGraw-Hill, 2000.

BASNEY, J.; LIVNY, M.; TANNENBAUM, T. High Throughput Computing with Condor. **HPCU News**, [S.l.], v.1, n.2, June 1997.

BECK, M. *et al*. **Linux Kernel Internals**. Harlow: Addison Wesley, 1999.

BHARGAVA, B. K.; LIAN S. R.; LEU P. L. Experimental Evaluation of Concurrency Checkpointing and Rollback-Recovery Algorithms Source. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 6., 1990, Los Angeles. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 1990.

BULIGON, C.; CECHIN, S.; JANSCH-PÔRTO, I. Implementing Rollback-Recovery Coordinated Checkpoints. In: INTERNATIONAL SCHOOL & SYMPOSIUM ON ADVANCED DISTRIBUTED SYSTEMS, ISSADS, 2005. **Proceedings…** Berlin: Springer-Verlag, 2005. p.246-257. (Lecture Notes in Computer Science, 3563)

CECHIN, S. L. **Protocolo de Recuperação por Retorno, Coordenado, não Determinístico**. 2002. 844 f. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

CRISTIAN, F. Understanding Fault-Tolerant Distributed Systems. **Communications of the ACM**, New York, v.34, n.2, p.56-78, 1991.

EKWALL, R.; URBÁN P.; SCHIPER, A. Robust TCP Connections for Fault Tolerant Computing. In: IEEE INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED SYSTEMS, 9., 2002. **Proceedings…** Los Alamitos: IEEE Computer Society Press, 2002. p.501-509.

ELNOZAHY, E. N. *et al*. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. **ACM Computing Surveys,** New York, v.34, n.3, p.375-408, 2002.

ELNOZAHY, E. N.; ZWAENEPOEL, W. Manetho: Transparent Rollback-recovery with Low Overhead, Limited Rollback and Fast Output Commit. **IEEE Transactions on Computers, Special Issue on Fault-Tolerant Computing**, Los Alamitos, v.41, n.5, p.526-531, 1992.

ELNOZAHY, E. N.; ZWAENEPOEL, W. The Performance of Consistent Checkpoint. In: IEEE SYMPOSIUM ON RELIABLE SYSTEMS, 11., 1992. **Proceedings…** [S.l.:s.n.], 1992a. p.39-47.

FETZER, C.; RAYNAL, M.; TRONEL, F. An adaptive failure detection protocol. In: IEEE PACIFIC RIM INTERNATIONAL SYMPOSIUM ON DEPENDABLE COMPUTING, 8., 2001. **Proceedings…** [S.l.:s.n.], 2001. p.146-153.

FONTOURA, A. B. **Avaliação de Abordagens para Captura de Informações da Aplicação**. 2002. 129 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

INTERNATIONAL ORGANIZATION FOR STANDARDS. **ISO/IEC 8327-1:** Information Technology-Open Systems Interconnection-Connection-Oriented Session Protocol: Protocol Specification. [S.l.], 1996.

JAIN, R. **The art of computer systems performance analysis:** techniques for experimental design, measurement, simulation, and modeling. New York: John Wiley, 1991.

JALOTE, P. **Fault Tolerance in Distributed Systems**. New Jersey: Prentice-Hall, 1994.

JIMÉNEZ, R. M.; RAJSBAUM, S. Cyclic Strategies for Balanced and Fault-Tolerant Distributed Storage. In: LATIN AMERICAN SYMPOSIUM ON DEPENDABLE COMPUTING, 1., 2003. **Proceedings…** Berlin: Springer-Verlag, 2003.

KAASHOEK, M. F. *et al*. Transparent Fault-Tolerance in Parallel Orca Systems. In: SYMPOSIUM ON EXPERIENCES WITH DISTRIBUTED AND MULTIPROCESSOR SYSTEMS, 3., 1992. **Proceedings…** [S.l.:s.n.], 1992. p.297-312.

KRISHNAN, S; GANNON, D. Checkpoint and Restart for Distributed Components In. IEEE/ACM INTERNATIONAL WORKSHOP ON GRID COMPUTING, GRID, 2004. **Proceedings…** [S.l.:s.n.], 2004.

LAMPORT, L. The Temporal Logic of Actions. **ACM Transactions on Programming Languages and Systems**, New York, v.16, n.3, p.872–923, 1994.

LAMPORT, L. Time, Clocks, and the Ordering of Events in a Distributed Systems. **Communications of the ACM**, New York, v.21, n.7, p.558–565, July 1978.

MATTHEW, N.; STONES, R. **Beginning linux programming**. Birmingham: Wrox Press, 1999.

MITCHEL, M.; OLDHAM, J. SAMUEL, A. **Advanced Linux Programming**. New York: New Riders Publishing, 2001.

NEBBET, G. **Windows NT/2000 Native API Reference**. Indianapolis: Macmillan Techinical Publishing, 1999.

NS - NETWORK Simulator. Disponível em: < http://www.isi.edu/nsnam/ns >. Acesso em: maio 2005.

ÖZSU, M. T.; VALDUREZ, P. **Princípios de Sistemas de Banco de Dados Distribuídos**. Rio de Janeiro: Campus, 2001.

PINHEIRO, E. **Truly-Transparent Checkpointing of Parallel Applications**. Rio de Janeiro: Universidade Federal do Rio de Janeiro – UFRJ/COPPE, 1999. Technical report.

PLANK, J. S.; BECK, M.; KINGSLEY, G. et al. Libckpt: Transparent Checkpointing under Unix. In: USENIX TECHNICAL CONFERENCE, 1995. **Proceedings…** New Orleans, Louisiana: [s.n.], 1995. p.213-223.

PRESSMAN, R. **Engenharia de *Software***. São Paulo: McGraw-Hill, 1995.

RANDELL, B. System Structure for fault-tolerance. **IEEE Transactions on Software Engineering**, [S.l.], v.SE-1, p.220-232, June 1975.

RAYNAL, M. **Distributed algorithms and protocols**. Chichesser: John Wiley, 1992.

RODRIGUES, L. A. **Extensão do Modelo de Defeitos do Framework para Simulação de Algoritmos Distribuídos no Neko**. 2005. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

RUBINI, A. **Linux Device Drivers**. EUA: O'Reilly, 1998.

SANKARAN, S. *et al*. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In: LACSI SYMPOSIUM, 2003. **Proceedings…** [S.l.:s.n.], 2003.

SCHLICHTING, R. D.; SCHNEIDER, F. B. Fail-stop processors: an approach to designing fault-tolerant computing systems. **ACM Transactions on Computer Systems**, New York, v.1, n.3, p.222–238, Aug. 1983.

SILVA, F. A. **Recuperação com Base em *Checkpointing*: uma Abordagem Orientada a Objetos**. 2002. 157 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

SINGHAL, M.; SHIVARATRI, N. G. **Advanced Concepts in Operating Systems:** Distributed, database, and multiprocessor operating systems. EUA: McGraw-Hill, 1994.

STELLNER, G. CoCheck: Checkpointing and Process Migration for MPI. In: INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM, 10., 1996. **Proceedings…** [S.l.:s.n.], 1996. p.526-531.

SUDAKOV, O. O.; MESHCHERYAKOV. E. S. **CHPOX - Checkpointing For Linux.** Disponível em: <http://www.cluster.kiev.ua/tasks/chpx_eng.html>. Acesso em: abr. 2003.

TANENBAUM, A. S. **Computer Networks**. New Jersey: Prentice-Hall, 1996.

TANENBAUM, A. S. **Distributed Systems Principles and Paradigms**. New Jersey: Prentice-Hall, 2002.

TANENBAUM, A. S. **Modern Operating Systems**. 2nd ed. New Jersey: Prentice-Hall, 2001.

CHECKPOINTING.ORG: The Home of Checkpointing Packages. Disponível em: <http://www.checkpointing.org>. Acesso em: mar. 2003.

TRINDADE, R. M. **Uso do Network Simulator – NS para Simulação de Sistemas Distribuídos em Cenários com Defeitos**. 2003. 132 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

URBÁN, P.; DÉFAGO, X.; SCHIPER, A. Neko: A single environment to Simulate and prototype distributed algorithms. In: IEEE INT'L CONFERENCE ON INFORMATION NETWORKING, 15., 2001. **Proceedings…** [S.l.:s.n.], 2001. p. 503-511.

ZANDY, V.C. **ckpt:** A process checkpoint library. Disponível em: <http://www.cs.wisc.edu/~zandy/ckpt>. Acesso em: mar. 2003.

ZANDY, V.C.; MILLER, B.P. Reliable network connections. In: ANNUAL ACM/IEEE INTERNATIONAL CONFERENCE ON MOBILE COMPUTING AND NETWORKING, 8., 2002. **Proceedings…** [S.l.:s.n.], 2002. p.95-106.

ZHONG, H. NIEH, J. **CRAK:** Linux checkpointing/restart as a kernel module. Columbia: Department of Computer Science, Columbia University, 2001. (Technical Report CUCS-014-01)

YANG, J. M.; ZHANG, D. F.; QIN, Z. WINDAR: A Multithreaded Rollback-Recovery Toolkit on Windows. In: IEEE PACIFIC INTERNETIONAL SYSMPOSIUM ON DEPENDABLE COMPUTING, 10., 2004. **Proceedings…** [S.l.:s.n.], 2004. p.395-400.

ANEXO A DESCRIÇÃO DA ESPECIFICAÇÃO DO ALGORITMO

Este anexo apresenta a especificação do algoritmo escrita em TLA e a descrição de seus elementos, extraído do trabalho de Cechin (2002).

```
                          MODULE Recovery
1   EXTENDS Naturals
2
3   CONSTANTS
4     Proc,  Conjunto dos identificadores dos processos do sistema
5     Mngr,  Identificador do manager (Coordenador)
6     Deliver  Deliver(m) entrega uma mensagem para a aplicação
7
8   VARIABLES
9     CurrCP  Ultimo CP estabelecido
10    PrevCP  Penúltimo CP estabelecido
11    CS  Estado do canal -- memória volátil
12    ME  Estado do Manager
13    CanalCanais de comunicação entre os processos
14    H  Message HISTORY
15
16  ASSUME
17    w ≜ <CurrCP, PrevCP, CS, ME, Canal, H>
18    MsgType ≜ { "APP", "ACK", "REQ", "AREQ", "CMT", "ACMT" }
19    NN ≜ CHOOSE x: (∀n ∈ Nat: x > n)
20    NoMsg ≜ CHOOSE x: (x.d ∉ MsgType)
21    Message ≜ [tx: Proc, rx: Proc, itx: Nat, irx: Nat ∪
23    { NN }, d: MsgType, msg: Message ∪ { NoMsg }]
24    ProcS ≜ Proc \ {Mngr}
-------------------------------------------------------------------
25  TypeInvariant ≜
26    ∧ CurrCP = [p ∈ Proc → Nat x SUBSET(Message)]
27    ∧ PrevCP = [p ∈ Proc → Nat x SUBSET(Message)]
28    ∧ CS = [p ∈ Proc → SUBSET(Message)]
29    ∧ ME ∈ {0,1}
30    ∧ Canal = [p ∈ Proc → [q ∈ Proc → SUBSET(Message)]]
31    ∧ H = SUBSET(Message)
32
33  Init ≜
34    ∧ CurrCP = [p ∈ Proc → <0,{}>]
35    ∧ PrevCP = [p ∈ Proc → <0,{}>]
36    ∧ CS = [p ∈ Proc → {}]
37    ∧ ME = 1
```

```
38    ∧ Canal = [p ∈ Proc → [q ∈ Proc → {}]]
39    ∧ H = {}
40
41  StoreCS(p,m) ≜ CS' = [CS EXCEPT ![p] = @ ∪ {m}]
42
43  RemoveCS(p,m) ≜ ∧ m ≠ NoMsg
44    ∧ CS' = [CS EXCEPT ![p] = @ \ {m}]
45
46  Checkpoint(p,k) ≜ CurrCP' = [CurrCP EXCEPT ![p] = <k,CS[p]>]
47
48  MoveCP(p) ≜ PrevCP' = [PrevCP EXCEPT ![p] = CurrCP[p] ]
49
50  Verify(p,m) ≜ ∧ m.itx > CurrCP[p][1] ⇒ Checkpoint(p,m.itx)
51                ∧ m.itx ≤ CurrCP[p][1] ⇒ CurrCP' = CurrCP
52
53  Send(p,q,m) ≜ ∧ m.tx  = p ∧ m.rx = q ∧ p ≠ q
54                ∧ m.itx = CurrCP'[p][1]
55                ∧ m.itx = NN
56                ∧ Canal' = [Canal EXCEPT ![q][p] = @ ∪ {m} ]
57                ∧ H' = H ∪ {m}
58
59  Receive(p,q,m) ≜ ∧ m.tx  = q ∧ m.rx = p ∧ p ≠ q
60                ∧ m ∈ Canal[p][q]
61                ∧ m.irx = CurrCP'[p][1]
62                ∧ Canal' = [Canal EXCEPT ![p][q] = @ \ {m} ]
63
64  Replay(p,q,m1,m2) ≜ ∧ m1.tx  = q ∧ m1.rx = p
65                ∧ m2.tx  = p ∧ m2.rx = q ∧ p ≠ q
66                ∧ m1 ∈ Canal[p][q]
67                ∧ m1.irx = CurrCP'[p][1]
68                ∧ m2.itx = CurrCP'[p][1]
69                ∧ m2.irx = NN
70                ∧ Canal' = [Canal EXCEPT
71                ![p][q] = @ \ {m},
72                ![q][p] = @ ∪ {m} ]
73                ∧ H' = H ∪ {m2}
```

```
74  SendApp(p) ≜ ∧ ∃ m1 ∈ Message: ∃ q ∈ Proc:
75                ∧ m1.d = "APP"
76                ∧ m1.msg = NoMsg
77                ∧ Send(p,q,m1)
78                ∧ StoreCS(p,m1)
79     ∧ UNCHANGED <CurrCP, PrevCP, ME>
80
81  SendAppMan ≜ ∧ ∃ m1 ∈ Message: ∃ q ∈ Proc:
82                ∧ m1.d = "APP"
83                ∧ m1.msg = NoMsg
84                ∧ Send(Mngr,q,m1)
85                ∧ StoreCS(Mngr,m1)
86     ∧ UNCHANGED <CurrCP, PrevCP, ME>
87
88  RxApp(p) ≜ ∧ ∃ m1 ∈ Message: ∃ q ∈ Proc:
89                ∧ Verify(p, m1)
89                ∧ m1.d = "APP"
90                ∧ Deliver(p,m1)
91                ∧ ∃ m2 ∈ Message:
```

```
92                ∧ m2.d = “ACK”
93                ∧ m2.msg = m1
94                ∧ Replay(p,q,m1,m2)
95    ∧ UNCHANGED <PrevCP, CS, ME>
96
97  RxAppMan ≜ ∧ ∃ m1 ∈ Message: ∃ q ∈ Proc:
98                ∧ m1.itx ≤ CurrCP[Mngr][1]
99                ∧ m1.d = “APP”
100   ∧ Deliver(p,m1)
101               ∧ ∃ m2 ∈ Message:
102               ∧ m2.d = “ACK”
103               ∧ m2.msg = m1
104               ∧ Replay(Mngr,q,m1,m2)
105   ∧ UNCHANGED <CurrCP, PrevCP, CS, ME>
106
107 RxAck(p) ≜ ∧ ∃ m1 ∈ Message: ∃ q ∈ Proc:
108               ∧ Receive (p, q, m1)
109               ∧ Verify(p, m1)
110   ∧ m1.d = “ACK”
111   ∧ m1.msg.tx = p ∧ m1.msg.rx = q
112   ∧ RemoveCS(p,m1,msg)
113   ∧ UNCHANGED <PrevCP, ME, H>
114
115 RxAckMan ≜∧ ∃ m1 ∈ Message: ∃ q ∈ Proc:
116               ∧ Receive (Mngr, q, m1)
117               ∧ m1.itx ≤ CurrCP[Mngr][1]
118   ∧ m1.d = “ACK”
119   ∧ m1.msg.tx = Mngr ∧ m1.msg.rx = q
120   ∧ RemoveCS(Mngr,m1,msg)
121   ∧ UNCHANGED <CurrCP, PrevCP, ME, H>
122
123 RxReq(p) ≜∧ ∃ m1 ∈ Message:
124               ∧ Verify(p,m1)
125   ∧ m1.d = “REQ”
126   ∧ ∃ m2 ∈ Message:
127   ∧ m2.d = “AREQ”
128   ∧ m2.msg = NoMsg
129   ∧ Replay(p, Mngr, m1, m2)
130   ∧ UNCHANGED <PrevCP, CS, ME>
131
132 SendReq ≜∧ StartNewCP ∧ ME=1
133           ∧ CurrCP[Mngr][1] = PrevCP[Mngr][1]
134   ∧ m1.d = “REQ”
135   ∧ ∀ q ∈ ProcS: ∀ m ∈ Message:
136                   m ∈ Canal[q][Mngr] ⇒  m.d = “APP” ∨
137                                         m.d = “ACK”
138   ∧ Checkpoint(Mngr, CurrCP[Mngr][1]+1)
139   ∧ ∀ q ∈ ProcS: ∃ m1 ∈ Message:
140   ∧ m1.d = “REQ”
141   ∧ m1.msg = NoMsg
142   ∧ Send (Mngr, q, m1)
143   ∧ ME’ = 0
144   ∧ UNCHANGED <PrevCP, CS>
145
146 RxAckReq ≜∧ ME = 0
147           ∧ CurrCP[Mngr][1] > PrevCP[Mngr][1]
148   ∧ ∀ q ∈ ProcS: ∃ m1 ∈ Message:
```

```
150    ∧ m1.itx = CurrCP[Mngr][1]
151    ∧ m1.d = “AREQ”
152    ∧ ∃ m2 ∈ Message:
153     ∧ m2.d = “CMT”
154     ∧ m2.msg = NoMsg
155     ∧ Replay(Mngr, q, m1, m2)
156             ∧ MoveCP(Mngr)
157    ∧ UNCHANGED <CurrCP, CS, ME>
158
159 RxAckCmt ≜∧ ME = 0
160             ∧ CurrCP[Mngr] = PrevCP[Mngr]
161    ∧ ∀ q ∈ ProcS: ∃ m1 ∈ Message:
162    ∧ m1.itx = CurrCP[Mngr][1]
163    ∧ m1.d = “ACMT”
164    ∧ Receive(Mngr, q, m1)
165             ∧ ME’ = 1
166    ∧ UNCHANGED <CurrCP, PrevCP, CS, H>
167
168 RxCmt(p) ≜∧ ∃ m1 ∈ Message:
169    ∧ m1.itx = CurrCP[p][1]
170    ∧ m1.d = “CMT”
171    ∧ ∃ m2 ∈ Message:
172     ∧ m2.d = “ACMT”
173     ∧ m2.msg = NoMsg
174     ∧ Replay(p, Mngr, m1, m2)
175             ∧ CurrCP[p][1] > PrevCP[p][1]
176             ∧ MoveCP(p)
177    ∧ UNCHANGED <CurrCP, CS, ME>
178
179 Manager ≜∨ SendAppMan ∨ RxAppMan ∨ RxAckMan
180           ∨ SendReq ∨ RxAckReq ∨ RxAckCmt
181
182 Slave(p) ≜∨ SendApp(p) ∨ RxApp(p) ∨ RxAck(p)
183           ∨ RxReq(p) ∨ RxCmt(p)
184
185 Next ≜ Manager ∨ ∃ p ∈ ProcS: Slave(p)
186
187 Recovery ≜ Init ∧ □ [Next]w
```

```
188 LOCAL
189
190 E(S) ≜ ∀ P,q ∈ Proc: S[p][1] = S[q][1]
191
192 MB(x,y,S) =  ∧ x.itx < S[x.tx][1]
193              ∧ x.irx < S[x.rx][1]
194              ∧ y.itx < S[x.rx][1]
195              ∧ y.irx < S[x.tx][1]
196
197 MA(x,y,S) =  ∧ x.itx ≥ S[x.tx][1]
198              ∧ x.irx ≥ S[x.rx][1]
199              ∧ y.itx ≥ S[x.rx][1]
200              ∧ y.irx ≥ S[x.tx][1]
201
202 MK(x,y,S) =  ∧ x.itx < S[x.tx][1]
203              ∧ y.itx ≥ S[x.rx][1]
204              ∧ y.irx ≥ S[x.tx][1]
```

```
205                ∧ x ∈ S[x.tx][2]
206
207 MP(x,y,S) =   ∧ x.itx < S[x.tx][1]
208                ∧ x.irx < S[x.rx][1]
209                ∧ y.irx ≥ S[x.tx][1]
210                ∧ x ∈ S[x.tx][2]
211
212 M(x,y) =    ∧ (x.d = “APP”)
213             ∧ (y.d = “ACK”)
214             ∧ (x.tx = y.rx)
215             ∧ (x.rx = y.tx)
216             ∧ (y.msg = x)
217
218 CC(x,y,S) = MB(x,y,S) ∨ MA(x,y,S) ∨ MK(x,y,S) ∨ MP(x,y,S)
219
220 MH(H,S) ≜ ∀ x,y ∈ Message:
221   x ∈ H ∧ y ∈ H ∧ M(x,y) ⇒ CC(x,y,S)
222
223 Com(H,S) ≜ E(S) ∧ MH(H,S)
224
225 Consistent ≜ Com(H,CurrCP) ∨ Com(H,PrevCP)
```

```
226 THEOREM
227   Recovery ⇒ □ Consistent
```

ANEXO B CONSIDERAÇÕES DO AUTOR DA BIBLIOTECA CRAK

-----Mensagem original-----

De: Hua Zhong [mailto:huaz@cs.columbia.edu]

Enviada em: quinta-feira, 18 de dezembro de 2003 17:05

Para: Clairton Buligon

Assunto: RE: CRAK

Clairton,

UDP is not supported. Not because it's hard (it's stateless compared to TCP), but it's so simple that it's not interesting.

You can try to implement it yourself. I guess the most you need to do is to record the port number when you check point, and bind to the same port when you restart, and it _should_ just work. (Maybe you don't even need a bind) Of course you need to socket/dup to the original fd.

It's just the theory, you'd have to hack into it yourself. If you get an error, check the error number and look into kernel source to see what's happening.

Hua

APÊNDICE A CONSTANTES

```
1    ... /* includes e demais constantes acessórias */
2
3    // Mensagens de controle para estabelecimento de linhas de recuperação
4    #define CKPT_MSG_REQ                // Solicitação de  novo checkpoint
5    #define CKPT_MSG_AREQ               // Resposta à solicitação
6    #define CKPT_MSG_CMT                // Solicitação de  confirmação
7    #define CKPT_MSG_ACMT               // Resposta à solicitação
8
9    // Mensagens de controle para rotação de coordenadores
10   #define CKPT_MSG_REQ_MANAGER        // Propõe novo manager
11   #define CKPT_MSG_AREQ_MANAGER       // Aceita manager proposto
12   #define CKPT_MSG_CMT_MANAGER        // Confirma manager proposto
13   #define CKPT_MSG_ACMT_MANAGER       // Aceita confirmação de novo manager
14
15   // Mensagens de controle para execução da aplicação
16   #define CKPT_MSG_START              // Inicia ou continua execução da aplicação
17   #define CKPT_MSG_STOP               // Pausa na execução da aplicação
18   #define CKPT_MSG_KILL_PROC          // Força encerramento da aplicação
19
20   // Mensagens de controle do detector de defeitos
21   #define CKPT_MSG_REQ_IMLIVE         // Solicitação de verificação de estado
22   #define CKPT_MSG_AREQ_IMLIVE        // Responde a solicitação de verificação
23   #define CKPT_MSG_NOTIFY_CRASH       // Notifica sobre defeito na aplicação
24   #define CKPT_MSG_ANOTIFY_CRASH      // Aceita notificação sobre defeito
25
26   // Mensagens de recuperação após eliminação da falha
27   #define CKPT_MSG_REQ_RESTART        // Propõe recuperação da aplicação
28   #define CKPT_MSG_AREQ_RESTART       // Concorda com recuperação da aplicação
29   #define CKPT_MSG_CMT_RESTART        // Confirma recuperação da aplicação
30   #define CKPT_MSG_ACMT_RESTART       // Responde à solicitação de confirmação
31
32   // Status da aplicação e do sistema
33   #define CKPT_STATUS_READY           // Sistema guardando carga da aplicação
34   #define CKPT_STATUS_RUNNING         // Sistema e aplicação em execução normal
35   #define CKPT_STATUS_WAIT            // Aplicação bloqueada pelo sistema
36   #define CKPT_STATUS_PCRASH          // Crash no processo da aplicação
37   #define CKPT_STATUS_CCRASH          // Crash no sistema de checkpoint
38   #define CKPT_STATUS_NORESP          // Sem resposta – provável partição na rede
39
40   // Serviço de membership
41   #define CKPT_MEMBER_NOJOIN          // Membro NÃO faz parte do grupo
42   #define CKPT_MEMBER_JOINED          // Membro faz parte do grupo
43   #define CKPT_MEMBER_CRASHSUSPECT    // Membro suspeito de estar com defeito
44   #define CKPT_MEMBER_CRASH           // Membro do grupo em estado de Crash
```

```
45
46  // Status do membro no grupo
47  #define CKPT_PROCESS_MANAGER      // Processo CK é o coordenador
48  #define CKPT_PROCESS_NOMANAGER    // Processo CK NÃO é o coordenador
49  #define CKPT_PROCESS_CANDIDATE    // Processo CK é candidato à ser coordenador
50
51  // Ack para mensagens da aplicação
52  #define CKPT_ACK_APPMSG           // Mensagem de confirmação de recebimento
```

APÊNDICE B ESTRUTURAS DE DADOS E VARIÁVEIS

```
1   // Estrutura para mensagens da aplicação
2   struct ckpt_message {
3       unsigned long message_id;       // Identificador da mensagem da aplicação
4       uint32_t ip_addr;               // Endereço IP do destinatário
5       uint16_t port;                  // Número da porta do destinatário
6       char UDP_msg[UIO_MAXIOV];       // Conteúdo original da mensagem
7       int retransmits;                // Número de retransmissões desta mensagem
8       time_t time_sec;                // Momento em que a mensagem foi interceptada
9       int CurrCP;                     // Identificador da linha de recuperação
10  };
11
12  // Estrutura para fila de mensagens em espaço de kernel – Buffer I
13  struct ckpt_message_buffer {
14      struct ckpt_message message;  // Estrutura da mensagem
15      struct ckpt_message_buffer * Next;// Ponteiro para próximo elemento
16  };
17
18  // Estrutura para mensagens de controle - para unicast e multicast
19  struct ckpt_msg_control {
20      int msgtype;                    // Tipo de mensagem de controle
21      uint32_t ip_dest;               // Endereço IP do destinatário em caso de unicast
22      time_t recvtime;                // Momento em que a mensagem foi recebida
23      char data[20];                  // Dados extras, se necessário
24      int CurrCP;                     // Identificador da linha de recuperação
25  };
26
27  // Estrutura para membros do grupo
28  struct ckpt_member {
29      uint32_t ip_addr;               // Endereço IP do membro do grupo
30      int CurrCP;                     // Linha de recuperação atual
31      int PrevCP;                     // Última linha de recuperação estabelecida
32      int manager;                    // Função do membro no grupo
33      time_t timemb;                 // Último instante em que o membro se manifestou
34      int status;                     // Estado do membro em relação ao grupo
35  };
36
37  // Estrutura para informações locais mantidas por cada membro do grupo
38  struct ckpt_info {
39
40      // Informações de controle do algoritmo
41      pid_t pid;                      // Identificador do processo sendo monitorado
42      int ME;                         // Indica estado do Coordenador {0,1}
43      int CurrCP;                     // Linha de recuperação atual
44      int PrevCP;                     // Última linha de recuperação estabelecida
45      char snapshot_filename[64];// Nome do arquivo com o checkpoint do processo
46      char messages_filename[64];// Nome do arquivo com as mensagens da aplicação
```

```
47      int ckptinterval;               // Intervalo entre checkpoints – em segundos
48      int retransmits;                // Máximo de tentativas de retransmissão
49      int timer_sec;                  // Timeout para abortar uma operação
50      int manager;                    // Função do membro no grupo
51      int status;                     // Estado local do sistema
52
53      // Informações adicionais
54      uint32_t ip_addr;               // Endereço IP local
55      char processname[32];           // Nome do processo da aplicação
56      char filename[64];              // Comando executado para carga da aplicação
57
58      // Informações estatísticas
59      unsigned long nrmsg_in;// Número de mensagens recebidas pela aplicação
60      unsigned long nrmsg_out;        // Número de mensagens enviadas pela aplicação
61      unsigned long nrmsg_lost;       // Número de mensagens perdidas
62      unsigned long nrmsg_resend;     // Número de mensagens re-enviadas
63      int nrmsg_lastckpt;             // Nº de mensagens salvas no último checkpoint
64      time_t starttime;               // Instante em que o último checkpoint iniciou
65      unsigned long timetockpt;// Tempo gasto para estabelecer o último checkpoint
66      unsigned long totaltimetockpt;// Tempo total gasto com checkpoint
67      unsigned long ctrl_nrmsg_in;  // Mensagens de controle recebidas
68      unsigned long ctrl_nrmsg_out; // Mensagens de controle enviadas
69      unsigned long ctrl_nrmsg_ret; // Total de mensagens retransmitidas
70      int memorytotal;                // Total de memória do sistema operacional
71      int memoryused                  // Memória utilizada pelo sistema operacional
72  };
73
74  / ***** VARIÁVEIS DO MÓDULO DE KERNEL *****/
75
76  unsigned long count_messages;       // Identificador incremental
77  struct ckpt_message_buffer *send_messages;// Buffer I
78
79  / ***** VARIÁVEIS DO PROCESSO CK *****/
80
81  struct ckpt_info ckpt_local_info;// Informações locais
82  list<ckpt_message> ckpt_SendApp_list;  // Buffer II
83  list<ckpt_msg_control> ckpt_list_msgcontrol;// Buffer III
84  list<ckpt_member> ckpt_list_group;// Membros
```

APÊNDICE C FUNÇÕES DO MÓDULO CKPT

```
1   /* Interceptação de carga de aplicações */
2   asmlinkage int my_execve(struct pt_regs regs) {...}
3
4   /* Interceptação de encerramento da execução de aplicações */
5   asmlinkage int my_exit(int error_code) {...}
6
7   /* Interceptação de mensagens da aplicação */
8   asmlinkage int my_socketcall(int call, unsigned long * args) {...}
9
10  /* Interface de leitura das mensagens da aplicação interceptadas e armazenadas no Buffer I  */
11  static ssize_t ckpt_read(struct file * file,
12      char * buf,
13      size_t count,
14      loff_t * offset) {...}
15
16  /* Interface de acesso à biblioteca CRAK e demais funções do módulo de kernel */
17  int ckpt_ioctl(struct inode * inode_i,
18    struct file * file,
19    unsigned int cmd,
20    unsigned long arg) {...}
```

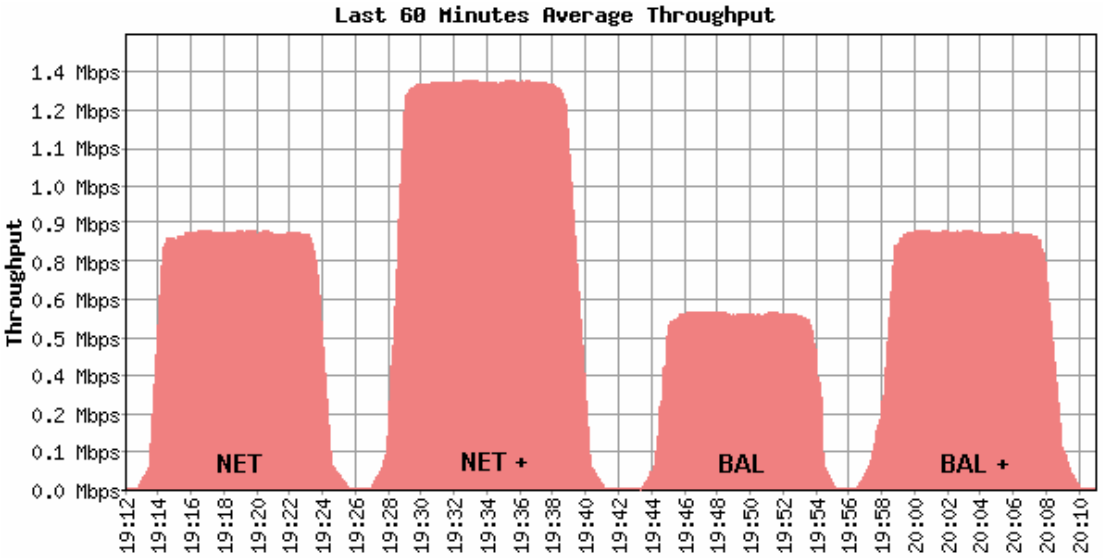
APÊNDICE D FUNÇÕES DO PROCESSO CK

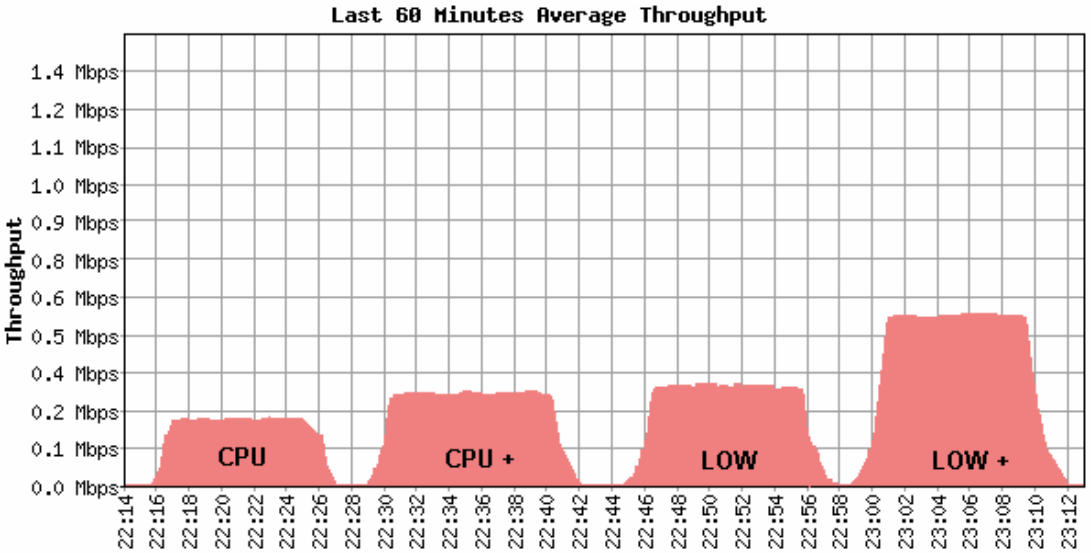
```
1   /* Inicialização dos processos */
2   bool init_param(struct ckpt_info &ckpt_local_info,
3         list<ckpt_member> &ckpt_list_group) {...}
4
5   /* Anuncia intenção de join no grupo via multicast */
6   void joinGroup(struct ckpt_msg_control command_message) {...}
7
8   /* Processa mensagem de novo membro no grupo */
9   int UpdateMemberGroup(struct ckpt_msg_control update_member) {...}
10
11  /* Propõe novo coordenador */
12  int rotateMngr() {...}
13
14  /* Processa proposta novo coordenador */
15  int acceptManager(struct ckpt_msg_control msg_rotate_mngr) {...}
16
17  /* Inicia nova linha de recuperação */
18  int initGlobalCheckpoint() {...}
19
20  /* Coleta de Lixo */
21  void do_garbage_collect(int old_version) {..}
22
23  /* Detector de defeitos */
24  int fault_detector(int ft_mode) {..}
25
26  /* Recuperação */
27  int start_recover(int nfault) {..}
28
29  /* Salvamento de checkpoints */
30  int do_ckpt(int ckpt_msgbuff_verify) {..}
31
32  /* Função Verify(p,m) */
33  int verify(int TxCurrCP) {..}
34
35  /* Processa requisição de nova linha de recuperação */
36  int processGlobalCkpt(struct ckpt_msg_control msg_global_ckpt)
37     {..}
38
39  /* Leitura das mensagens do módulo de kernel – Lê do buffer I */
40  int read_messages_dev(unsigned long ack_message_id) {..}
41
42  /* Processa mensagens da aplicação – Atualiza Buffer II */
43  int process_Appmessages() {..}
44
45  /* Envia mensagem de confirmação – ACK – para mensagens recebidas pela aplicação */
```

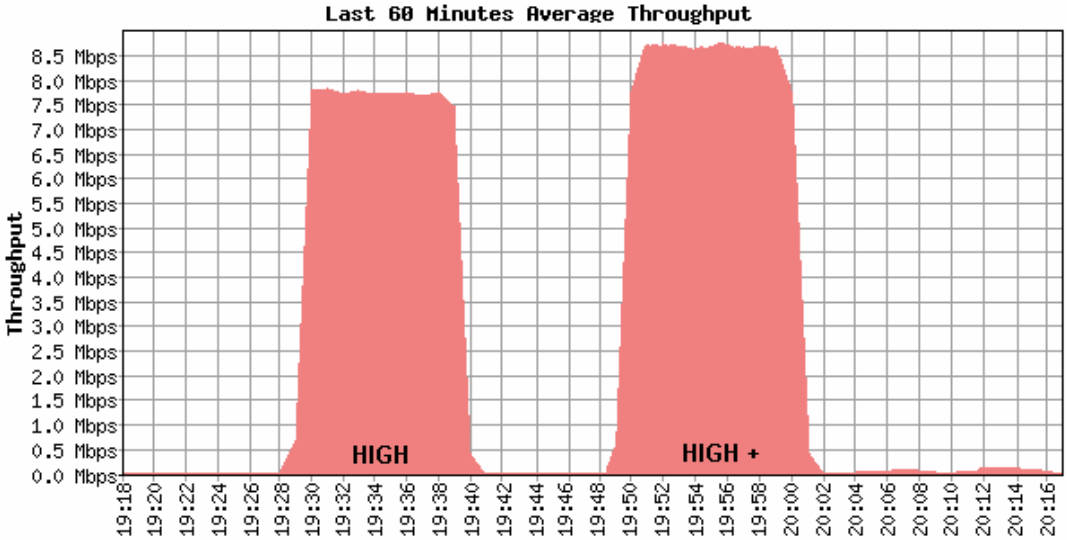
```
46   int RxAckAPP(struct ckpt_msg_control RxAckMsg) {..}
47
48   /* Thread para receber mensagens de confirmação – ACK – Atualiza Buffer II */
49   int *recv_RxAckAPP_th(void *arg) {..}
50
51   /* Thread para receber mensagens de controle do algoritmo – escrita no Buffer III */
52   int *recv_messages_th(void *arg) {..}
53
54   /* Thread para processar as mensagens de controle – leitura do Buffer III */
55   int *process_messages_th(void *arg) {..}
```

APÊNDICE E CONSUMO DE RECURSOS DE REDE

Esse apêndice lista os diagramas representando o consumo de banda de rede capturado a partir da ferramenta Ntop, nos diferentes cenários de execução das aplicações, sem e com (indicados pelo símbolo “+”) a utilização do sistema de *checkpointing.*






Last 60 Minutes Average Throughput
Throughput
8.5 Mbps
8.0 Mbps
7.5 Mbps
7.0 Mbps
6.5 Mbps
6.0 Mbps
5.5 Mbps
5.0 Mbps
4.5 Mbps
4.0 Mbps
3.5 Mbps
3.0 Mbps
2.5 Mbps
2.0 Mbps
1.5 Mbps
1.0 Mbps
0.5 Mbps
0.0 Mbps
HIGH
HIGH +
19:18
19:20
19:22
19:24
19:26
19:28
19:30
19:32
19:34
19:36
19:38
19:40
19:42
19:44
19:46
19:48
19:50
19:52
19:54
19:56
19:58
20:00
20:02
20:04
20:06
20:08
20:10
20:12
20:14
20:16

APÊNDICE F CÁLCULO DO Nº DE REPETIÇÕES NECESSÁRIAS

Intervalo de confiança: **95%**.

Margem de erro: **2,5%**.

Tabela de *quantis* da unidade normal de distribuição: (JAIN,1991, p. 629).

Cenários da Aplicação	Medidas realizadas (em *ms*)				Repetições necessárias
	1	2	3	4	
NET-none	629341	610574	611547	630953	**6**
NET-CKPTd	630563	611687	625875	618862	**4**
NET-CKPT0	634928	613384	637371	632523	**6**
NET-CKPT5	638339	629360	649949	650920	**5**
NET-CKPT15	684689	659579	686339	695999	**10**
BAL-none	655618	671478	650679	642679	**7**
BAL-CKPTd	653056	669081	641082	660748	**7**
BAL-CKPT0	671839	668349	649470	669209	**5**
BAL-CKPT5	669339	689240	695109	676920	**6**
BAL-CKPT15	729429	701849	738129	728149	**9**
CPU-none	601291	622411	630524	605241	**10**
CPU-CKPTd	611141	628785	599420	621037	**8**
CPU-CKPT0	619699	628730	618150	639890	**5**
CPU-CKPT5	664199	641210	668570	640639	**10**
CPU-CKPT15	680359	699340	669780	703839	**10**
LOW-none	607531	623814	608696	615169	**3**
LOW-CKPTd	610202	616210	608028	622752	**3**
LOW-CKPT0	618477	614760	626618	619552	**2**
LOW-CKPT5	634386	622401	624436	634535	**2**
LOW-CKPT15	680750	668663	673289	665091	**2**
HIGH-none	618495	619718	603943	610993	**3**
HIGH-CKPTd	623151	613526	605288	617805	**3**
HIGH-CKPT0	616335	622643	621812	608204	**3**
HIGH-CKPT5	628682	631229	632548	619191	**2**
HIGH-CKPT15	679544	669347	658272	671547	**4**

Definição dos cenários

none	Sistema de *checkpointing* ausente.
CKPTd	Sistema de *checkpointing* presente mas desabilitado. Somente interceptação ativada.
CKPT0	Execução da aplicação com sistema de *checkpointing* sem alocação de memória adicional pela aplicação. Tamanho do *checkpoint* local(*): entre 71176 bytes a 77228 bytes.
CKPT5	Execução da aplicação com sistema de *checkpointing* e alocação de 5 Mbytes de memória adicional pela aplicação. Tamanho do *checkpoint* local(*): entre 5322248 bytes a 5326396 bytes.
CKPT15	Execução da aplicação com sistema de *checkpointing* e alocação de 15 Mbytes de memória adicional pela aplicação. Tamanho do *checkpoint* local(*): entre 15816200 bytes a 15816252 bytes.

(*) Mais 1056 bytes para cada mensagem salva no *checkpoint*.

APÊNDICE G RESULTADOS DAS SIMULAÇÕES REALIZADAS

Aplicação	Medições - tempo em milisegundos											Overhead %
	1	2	3	4	5	6	7	8	9	10	Média	
Cenário 1 - Ausência do sistema de *checkpointing*												
NET	629341	610574	611547	630953	629193	617648					**621543**	--
BAL	655618	671478	650679	642679	652989	653953	649724				**653874**	--
CPU	601291	622411	630524	605241	618126	621064	628405	610431	603206	604653	**614535**	--
LOW	607531	623814	608696	615169							**613803**	--
HIGH	618495	619718	603943	610993							**613287**	--
Cenário 2 - *Checkpointing* instalado no sistema mas não sendo usado pela aplicação - interceptação												
NET	630563	611687	625875	618862							**621747**	**0,033**
BAL	653056	669081	641082	660748	658341	649158	650421				**654555**	**0,104**
CPU	611141	628785	599420	621037	628688	610518	602184	618921			**615087**	**0,090**
LOW	610202	616210	608028	622752							**614298**	**0,081**
HIGH	623151	613526	605288	617805							**614943**	**0,270**
Cenário 3 - *Checkpointing* habilitado sem alocação de memória adicional pela aplicação												
NET	634928	613384	637371	632523	624018	620297					**627087**	**0,892**
BAL	671839	668349	649470	669209	656195						**663012**	**1,398**
CPU	619699	628730	618150	639890	619340						**625162**	**1,729**
LOW	618477	614760	626618	619552							**619852**	**0,985**
HIGH	616335	622643	621812	608204							**617249**	**0,646**
Cenário 4 – *Checkpointing* habilitado e alocação de 5 MBytes de memória em cada processo da aplicação												
NET	638339	629360	649949	650920	648303						**643374**	**3,512**
BAL	669339	689240	695109	676920	683856	687520					**683664**	**4,556**
CPU	664199	641210	668570	640639	641008	650405	661268	644780	652533	661092	**652570**	**6,189**
LOW	634386	622401	624436	634535							**628940**	**2,466**
HIGH	628682	631229	632548	619191							**627913**	**2,385**
Cenário 5 - *Checkpointing* habilitado e alocação de 15 Mbytes de memória em cada processo da aplicação												
NET	684689	659579	686339	695999	671187	672875	688661	689817	690170	660906	**680022**	**9,409**
BAL	729429	701849	738129	728149	698743	708738	728131	721946	713374		**718721**	**9,917**
CPU	680359	699340	669780	703839	670521	688329	679068	698921	690166	679578	**685990**	**11,627**
LOW	680750	668663	673289	665091							**671948**	**9,473**
HIGH	679544	669347	658272	671547							**669678**	**9,195**

APÊNDICE H MENSAGENS SALVAS E TEMPO POR *CHECKPOINT*

		Mensagens salvas		Tempo por *checkpoint* em *ms*		
Aplicação	Cenário	p/*Checkpoint*	Total	Médio	Máximo	Unitário
NET	CKPT-0	6,9	783,3	8,0	13,0	**0,4**
NET	CKPT-5	10,4	1182,0	1152,7	1856,8	**60,7**
NET	CKPT-15	11,4	943,9	2513,1	4383,0	**182,1**
BAL	CKPT-0	12,8	1464,0	12,7	31,2	**0,7**
BAL	CKPT-5	12,8	1456,3	1087,4	1696,3	**57,2**
BAL	CKPT-15	13,6	1142,2	2834,7	4870,0	**202,5**
CPU	CKPT-0	2,1	244,2	29,1	31,2	**1,5**
CPU	CKPT-5	2,9	333,6	1126,1	1825,7	**59,3**
CPU	CKPT-15	14,2	1182,6	2541,4	4441,8	**182,8**
LOW	CKPT-0	2,3	239,3	10,7	14,5	**0,6**
LOW	CKPT-5	2,9	260,5	993,0	1502,0	**50,8**
LOW	CKPT-15	2,1	180,8	2283,8	3891,3	**168,2**
HIGH	CKPT-0	4,1	451,8	15,0	18,8	**0,8**
HIGH	CKPT-5	2,3	426,8	1103,5	1779,5	**55,8**
HIGH	CKPT-15	3,1	305,8	2096,2	4090,8	**181,0**

APÊNDICE I MENSAGENS INTERCEPTADAS

		Msgs da aplicação		Msgs de controle			
Aplicação	Terminal	Recebidas	Enviadas	Retrans	Perdidas	Recebidas	Enviadas
NET	T1	60000	60025	0	0	60213	60066
	T2	300127	300127	5081	127	300230	300262
	T3	60000	60023	0	0	60238	60082
	T4	60000	60030	0	0	60208	60077
	T5	60000	60028	0	0	60197	60069
	T6	60000	60022	0	0	60252	60071
BAL	T1	40000	40189	0	0	40284	40067
	T2	200995	200995	2026	81	200224	200113
	T3	40000	40194	0	0	40288	40065
	T4	40000	40192	0	0	40287	40062
	T5	40000	40181	0	0	40297	40068
	T6	40000	40193	0	0	40289	40067
CPU	T1	10000	10052	0	0	10259	10063
	T2	50252	50252	2351	284	50207	50333
	T3	10000	10050	0	0	10264	10064
	T4	10000	10051	0	0	10258	10062
	T5	10000	10053	0	0	10261	10059
	T6	10000	10047	0	0	10261	10063
LOW	T1	30000	30010	2	0	30208	30063
	T2	150060	150060	819	54	150207	150111
	T3	30000	30015	1	0	30210	30070
	T4	30000	30009	2	0	30209	30070
	T5	30000	30007	2	0	30210	30067
	T6	30000	30012	1	0	30208	30067
HIGH	T1	2	600004	1559	0	600204	70
	T2	3000025	10	0	0	212	3000090
	T3	2	600006	1524	0	600205	71
	T4	2	600004	1524	0	600205	71
	T5	2	600006	1507	0	600204	71
	T6	2	600006	1544	0	600204	69

T*n* = identificador da estação.
Em T2 foi executado o processo servidor e nos demais os processos clientes.

APÊNDICE J CONTEÚDO DO CD-ROM

O CD-ROM disponibilizado com este trabalho contém o código-fonte completo do sistema de *checkpointing* e das aplicações sintéticas, além de instruções de compilação, instalação e utilização do mesmo.

Também foram reunidas as bibliotecas de *checkpointing* estudas neste trabalho, além da biblioteca `CRAK` na sua forma original.

Por fim, para uma eventual necessidade de acesso ao ambiente de desenvolvimento trabalhado pelo autor, foi inserido neste CD-ROM uma imagem completa do sistema operacional (arquivos binários e código-fonte), instalado em um dos nodos da rede em que os foram conduzidas execuções das aplicações distribuídas e do sistema de *checkpointing* para coleta dos resultados utilizados na avaliação. Essa imagem foi obtida a partir da ferramenta Norton Ghost® versão 7.0, e contém o conteúdo instalado na partição 3 (partição primária) do disco rígido. Sugere-se, como alternativa para acesso aos arquivos dessa imagem sem a necessidade de instalá-la, a utilização da ferramenta Ghost Explorer®. Por questões de licenciamento, essas duas ferramentas não foram disponibilizadas neste CD-ROM.

O conteúdo do CD-ROM está organizado como segue:

- **Diretório `CKPT`**: contém os arquivos que formam o sistema de *checkpointing*.
- **Diretório `APPS`**: contém os arquivos das aplicações sintéticas.
- **Diretório `LIBS`**: contém as bibliotecas de *checkpointing* estudadas.
- **Diretório `IMAGEM`**: contém a imagem do ambiente de implementação.