UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE FÍSICA DE SÃO CARLOS

**Ambiente para Minimização do Impacto de Falhas
para Aplicações Paralelas**

José Luís Zem

Orientador: Prof. Dr. Gonzalo Travieso

São Carlos
Agosto - 2005

UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE FÍSICA DE SÃO CARLOS

**Ambiente para Minimização do Impacto de Falhas
para Aplicações Paralelas**

José Luís Zem

Orientador: Prof. Dr. Gonzalo Travieso

Tese apresentada como parte dos requisitos para a obtenção do grau de Doutor em Ciências (sub-área Física Computacional) do Instituto de Física de São Carlos da Universidade de São Paulo.

São Carlos
Agosto – 2005

Aos meus filhos Marília e Daniel, à minha esposa
Elaine e aos meus pais Orivaldo e Iolanda.

Agradecimentos

Inicialmente a Deus por me dar força, conhecimento e vontade de vencer mais esta etapa de minha vida.

Ao professor Dr. Gonzalo Travieso, meu orientador, pela ajuda na elaboração deste trabalho e pela sua paciência. Sem dúvida uma excelente pessoa e um grande amigo.

À minha esposa Elaine, pela paciência, compreensão, apoio, confiança e carinho em nossos momentos mais difíceis.

Aos meus filhos Marília e Daniel, pelos momentos que não pudemos estar juntos mas que sempre desejei que estivessem do meu lado.

Às amigas que fiz no Instituto de Física de São Carlos, principalmente a Elo, a Thatyana, o André, o Francisco e o Paulino.

À todos os professores que encontrei ao longo de todos esses anos de estudo e que sempre acreditaram em mim.

À todos que, direta ou indiretamente, contribuíram ao longo deste trabalho.

Resumo

Os sistemas paralelos são importantes pois permitem concentrar recursos computacionais como processadores, memórias e dispositivos de E/S para solucionar problemas computacionais que necessitam de uma grande quantidade destes mesmos recursos e em um tempo de execução aceitável.

Tradicionalmente, o tempo, a capacidade e o custo do processamento para se resolver estes problemas computacionais utilizando-se aplicações seqüenciais podem ser proibitivos e isto acaba criando um contexto propício para se utilizar aplicações paralelas.

Em razão de ser composto por muitas partes, um sistema distribuído está sujeito a falhas em seu subsistema de comunicação, em seus processadores, em suas aplicações entre outros componentes.

Desta maneira, as aplicações paralelas, ao utilizarem os sistemas distribuídos, têm suas partes executadas em paralelo pelos recursos distribuídos. Em razão de cada um destes recursos ser um possível ponto de falha, as aplicações paralelas acabam por tornarem-se mais susceptíveis à ocorrência de falhas e, conseqüentemente, à interrupção de suas execuções.

Quando estas aplicações paralelas são interrompidas, todo o processamento realizado e o tempo gasto para tal são desperdiçados, pois as aplicações devem ser reinicializadas.

Para minimizar estes desperdícios de tempo e processamento é apresentado neste trabalho um ambiente de monitoramento e execução que fornece mecanismos para se detectar falhas da classe *fail stop* em aplicações paralelas executadas em ambientes distribuídos ou centralizados. O ambiente em questão é denominado de *AMTF (Ambiente de Monitoramento Tolerante a Falhas)*.

O ambiente AMTF utiliza as técnicas de *checkpointing/restart* para armazenar e recuperar os estados dos processos e de *heartbeat* para verificar a continuidade de execução destes mesmos processos.

Juntamente com o ambiente AMTF é disponibilizada uma biblioteca a ser utilizada pelo desenvolvedor de aplicações paralelas, sendo que a mesma oferece a liberdade de se indicar no código-fonte da aplicação o ponto e o momento que se deseja que o contexto da aplicação seja armazenado para uma possível recuperação além de sua periodicidade para os registros automáticos.

Abstract

Parallel systems enable the coordinated use of computational resources like processors, memories and I/O devices. This makes these systems important for the solution of computationally demanding problems in an acceptable time frame.

The solution of these kind of problems in a sequential system can be prohibitive due to time constraints, resource requirements or the cost of the high performance systems needed. As a consequence, the use of parallel solutions is stimulated.

A distributed system is composed of many parts, like the communication subsystem, the processors, and even the different programs that execute to keep the system running. Each of these parts is subjected to faults.

A parallel application has its many components running on the different resources of a distributed system and is thus subject to a fault in any of the distributed systems' resources. Therefore, a parallel application is more exposed to fault and interruptions in its execution than a sequential application.

As a rule, when an interruption occurs, all of the processing done by the parallel application is lost and needs to be redone, resulting in significative loss of time and computational resources.

In a effort to minimize this loss of time and computational resources, this work presents a monitoring and execution environment that enables the detection of fault of the class fail stop in parallel applications executed in distributed or centralized parallel systems. The environment is called *AMTF* (from the portuguese "*Ambiente de Monitoramento Tolerante a Falhas*").

AMTF is based on checkpointing/restart techniques to store and recover the

state of the parallel processes and in the heartbeat technique to monitor the execution of the processes.

An API for the developer of parallel applications is also included, which enables the anotation in the source code of the places where a checkpoint is requested, or the periodicity of the automatic checkpointing.

Zem, José Luís

“Ambiente para Minimização do Impacto de Falhas para Aplicações Paralelas”

José Luís Zem – São Carlos, 2005

Tese (Doutorado) – Área de Física Aplicada do Instituto de Física de São Carlos da Universidade de São Paulo

2005 - Páginas: 139

Orientador: Prof. Dr. Gonzalo Travieso.

1. Tolerância a Falhas; 2. *Cluster* de Computadores; 3. Alta Disponibilidade

I. Título

Sumário

1. Introdução	1
2. Contextualização	5
2.1. Sistemas Paralelos e a Tolerância a Falhas	7
2.2. Sistemas Distribuídos e a Tolerância a Falhas.....	9
2.3. Disponibilidade	10
2.4. Falta, Falha e Defeito	12
2.5. Tolerância a Falhas.....	14
2.5.1. Classes da Tolerância a Falhas.....	14
2.5.2. Redundância.....	16
2.5.3. Propriedades <i>Safety</i> e <i>Liveness</i>	16
2.5.4. Formas de Tolerância a Falhas	17
2.5.5. Processo de <i>Failover</i> e <i>Failback</i>	19
2.5.6. Abordagens de Projeto e Arquitetural.....	20
2.5.6.1. Abordagem Arquitetural de <i>Hardware</i>	21
2.5.6.2. Abordagem Arquitetural de <i>Software</i>	22
2.5.7. Tolerância a Falhas em <i>Software</i>	22
2.5.7.1. Técnica de Registros em <i>Log</i>	22
2.5.7.2. Técnica de <i>Checkpointing</i>	23
2.5.7.3. Técnica de Espelhamento	28
2.5.7.4. Técnica de Gravação Atômica.....	28
2.5.7.5. Técnica de Rejuvenescimento	29
2.5.8. Considerações sobre a Tolerância a Falhas em <i>software</i>	30
2.5.9. Comunicações Síncrona e Assíncrona	31
2.5.10. Sistemas de Arquivos Tolerante a Falhas.....	33
2.5.10.1. RAID - <i>Redundant Array of Independent Disks</i>	36
2.5.10.2. AFS - <i>Andrew File System</i> e <i>Coda File System</i>	38
2.5.10.3. NFS - <i>Network File System</i>	41
2.5.11. <i>Heartbeat</i>	42
2.6. Implementações da Técnica <i>Checkpointing/Restart</i>	43

2.6.1. <i>Berkeley Labs Checkpoint/Restart</i> – BLCR	44
2.6.2. Outras implementações da técnica <i>Checkpointing/Restart</i>	48
2.6.3. A abordagem utilizada para o trabalho	49
3. Ambiente para Minimização do Impacto de Falhas para Aplicações Paralelas.....	51
3.1. Estrutura do Ambiente AMTF	53
3.2. Descrição dos Módulos.....	54
3.2.1. O <i>Controlador AMTF</i>	55
3.2.2. O <i>Monitor Central</i>	57
3.2.3. O <i>Monitor de Réplica</i>	62
3.2.4. O <i>Executor</i>	63
3.2.5. A <i>Aplicação Paralela</i>	64
3.2.6. O <i>Monitor do Grupo de Processos</i>	65
3.2.7. O Sistema de Arquivos Tolerante a Falhas	69
3.3. Alocação dos módulos aos nós do <i>Cluster</i> de Computadores	70
3.4. A Biblioteca AMTF	72
3.5. Ciclos de Vida	75
3.6. A Implementação dos Módulos do Ambiente AMTF	87
4. Desenvolvimento e Testes	89
4.1. Uso do Ambiente AMTF	90
4.2. Padrões de Comportamento do Ambiente AMTF	91
4.2.1. O Experimento I	92
4.2.2. O Experimento II	95
4.2.3. O Experimento III	97
4.2.4. O Experimento IV	100
4.2.5. O Experimento V	102
4.3. Interrupção e Reativação dos Módulos do Ambiente AMTF	106
4.3.1. Interrupção e Reativação do <i>Monitor Central</i>	107
4.3.2. Interrupção e Reativação do <i>Monitor de Réplica</i>	108
4.3.3. Interrupção e Reativação do <i>Monitor do Grupo de Processos</i>	110
4.3.4. Interrupção e Reativação da <i>Aplicação Paralela</i>	113
4.3.5. Interrupção e Reativação do <i>Monitor Central Virtual</i>	114
4.3.6. Interrupção e Reativação do Ambiente AMTF	116
4.3.7. Interrupção e Reativação de um Nó do <i>Cluster</i> de Computadores	117

4.4. A biblioteca do Ambiente AMTF	118
5. Conclusões e Propostas de Trabalhos Futuros	122
Referências Bibliográficas	126
Glossário	135

Lista de Figuras

Figura 2.1: Sistema logicamente paralelo	5
Figura 2.2: Sistema fisicamente paralelo (múltiplos computadores)	6
Figura 2.3: Sistema fisicamente paralelo (múltiplas CPUs).....	6
Figura 2.4: Níveis de abstração (falta, falha e defeito).....	14
Figura 2.5: Formas de Tolerância a Falhas	17
Figura 2.6: Procedimentos de <i>failover</i> e <i>failback</i>	19
Figura 2.7: O Efeito Dominó	25
Figura 2.8: Fases do sistema de gerenciamento de caches em <i>journal</i>	34
Figura 2.9: RAID de Nível 1	37
Figura 2.10: RAID de Nível 5	37
Figura 2.11: RAID de Nível 0	37
Figura 2.12: Servidores do <i>Andrew File System</i> (AFS) e do <i>CODA File System</i> ligados à estrutura de diretórios do cliente	39
Figura 2.13: <i>Heartbeat</i> com envio e recebimento	42
Figura 2.14: <i>Heartbeat</i> somente com retorno	43
Figura 3.1: Estrutura do Ambiente AMTF	53
Figura 3.2: Interação e Comunicação entre os módulos do Ambiente AMTF	55
Figura 3.3: Interação e Comunicação entre o <i>Controlador AMTF</i> e <i>Monitor Central</i>	56
Figura 3.4: Interações e Comunicações do módulo <i>Monitor Central</i>	58
Figura 3.5: Interações e Comunicações do módulo <i>Monitor de Réplica</i>	62
Figura 3.6: Interações do módulo <i>Executor</i>	63
Figura 3.7: Interações e Comunicações do módulo <i>Monitor do Grupo de Processos</i>	65
Figura 3.8: Sinalização de <i>heartbeat</i>	67
Figura 3.9: Solicitação para tomada de <i>checkpoints</i> locais e global	69
Figura 3.10: Distribuição dos módulos pelos nós de um <i>Cluster</i>	71
Figura 3.11: Módulos do AMTF com os mecanismos de comunicação em um <i>Cluster</i>	72

Figura 3.12: Ciclo de vida padrão dos módulos	76
Figura 3.13: Ciclo de vida sem a presença de falhas	77
Figura 3.14: Ciclo de vida com a presença de falhas na <i>Aplicação Paralela</i>	78
Figura 3.15: Ciclo de vida com a presença de falhas no <i>Monitor do Grupo de Processos</i>	79
Figura 3.16: Ciclo de vida com a presença de falhas no <i>Monitor Central</i>	80
Figura 3.17: Ciclo de vida com a presença de falhas no <i>Monitor de Réplicas</i>	81
Figura 3.18: Ciclo de vida da ativação do Ambiente AMTF	81
Figura 3.19: Ciclo de vida do encerramento do Ambiente AMTF	82
Figura 3.20: Ciclo de vida da reativação do Ambiente AMTF	83
Figura 3.21: Ciclo de vida do funcionamento do <i>Monitor Central Virtual</i>	84
Figura 3.22: Ciclo de vida com falha em uma cópia do <i>Monitor Central Virtual</i>	84
Figura 3.23: Ciclo de vida com reativação de nó após interrupção	86
Figura 4.1: Representação do <i>Cluster</i> de Computadores utilizados nos testes	89
Figura 4.2: Representação lógica do sistema de arquivos dos nós	90
Figura 4.3: Utilização da CPU durante o Experimento I	94
Figura 4.4: Quantidade de pacotes recebidos durante o Experimento I	94
Figura 4.5: Quantidade de pacotes transmitidos durante o Experimento I	94
Figura 4.6: Quantidade de processos no sistema durante o Experimento I	94
Figura 4.7: Utilização da CPU durante o Experimento II	96
Figura 4.8: Quantidade de pacotes recebidos durante o Experimento II	96
Figura 4.9: Quantidade de pacotes transmitidos durante o Experimento II	96
Figura 4.10: Quantidade de processos no sistema durante o Experimento II	96
Figura 4.11: Utilização da CPU durante o Experimento III	98
Figura 4.12: Quantidade de pacotes recebidos durante o Experimento III	98
Figura 4.13: Quantidade de pacotes transmitidos durante o Experimento III	98
Figura 4.14: Quantidade de processos no sistema durante o Experimento III	98
Figura 4.15: Utilização da CPU durante o Experimento IV	101
Figura 4.16: Quantidade de pacotes recebidos durante o Experimento IV	101
Figura 4.17: Quantidade de pacotes transmitidos durante o Experimento IV	101
Figura 4.18: Quantidade de processos no sistema durante o Experimento IV	101
Figura 4.19: Utilização da CPU durante o Experimento V	103
Figura 4.20: Quantidade de pacotes recebidos durante o Experimento V	103

Figura 4.21: Quantidade de pacotes transmitidos durante o Experimento V	103
Figura 4.22: Quantidade de processos no sistema durante o Experimento V	103
Figura 4.23: Comparativo dos tempos de utilização da CPU nos Experimentos III, IV e V	105

Listagens

Listagem 3.1: Utilização das chamadas disponibilizadas pela biblioteca AMTF	73
Listagem 4.1: <i>Log do Monitor de Réplica (interrupção do Monitor Central)</i>	107
Listagem 4.2: <i>Log do Monitor Central (interrupção do Monitor Central)</i>	108
Listagem 4.3: <i>Log do Monitor Central (interrupção do Monitor de Réplica)</i>	108
Listagem 4.4: <i>Log do Monitor de Réplica (interrupção do Monitor de Réplica)</i>	110
Listagem 4.5: <i>Log do Monitor Central (interrupção do Monitor do Grupo de Processos)</i>	111
Listagem 4.6: <i>Log do Monitor do Grupo de Processos (interrupção do Monitor do Grupo de Processos)</i>	112
Listagem 4.7: <i>Log do Monitor do Grupo de Processos (interrupção na Aplicação Paralela)</i>	113
Listagem 4.8: <i>Log do Monitor Central do Nó 1 monitorando e reativando o Monitor Central do Nó 0</i>	114
Listagem 4.9: <i>Log do Monitor Central do Nó 0 após ser reativado pelo Monitor Central do Nó 1</i>	116
Listagem 4.10: <i>Log do Monitor Central demonstrando o retorno sem a existência de aplicação paralela interrompida</i>	117
Listagem 4.11: <i>Log do Monitor Central demonstrando o retorno com a existência de aplicação paralela interrompida</i>	117
Listagem 4.12: Código-fonte de aplicação paralela utilizando as chamadas do ambiente AMTF	119
Listagem 4.13: Código-fonte de aplicação seqüencial utilizando as chamadas do ambiente AMTF	121

1. INTRODUÇÃO

Sistemas paralelos são importantes porque permitem concentrar recursos computacionais como os processadores, as memórias e os dispositivos de entrada e saída para solucionar problemas (representados por programas computacionais como modelagem climática, realidade virtual, simulações, etc) que necessitam de uma quantidade de recursos computacionais relativamente grande em um tempo de execução aceitável.

O aumento no desempenho fornecido por estes sistemas resulta da capacidade de dividir o trabalho em pequenas partes e encaminhar cada uma delas para ser processada paralelamente em nós de computação diferentes.

Um sistema distribuído está sujeito a diversos tipos de falhas, dentre os quais destacam-se aqueles provenientes de problemas no subsistema de comunicação, nos processadores, nas aplicações, nos sistemas operacionais e demais componentes que constituem um sistema de computação.

Tradicionalmente, o tempo, a capacidade e o custo do processamento para se resolver problemas computacionais de grandes proporções utilizando-se aplicações seqüenciais podem ser proibitivos e é neste contexto que torna-se necessário utilizar aplicações paralelas para solucionar estes mesmos problemas.

Desta forma, as aplicações paralelas, quando colocadas em execução, utilizam as funcionalidades disponibilizadas pelos sistemas distribuídos, tais como a distribuição das tarefas a serem executadas utilizando-se os vários processadores, a utilização freqüente do subsistema de comunicação para a comunicação e o acesso,

também, constante ao sistema de arquivos disponibilizado.

Assim, justamente por utilizar constantemente estes recursos distribuídos e muitas vezes não preparados para tratar as falhas, as aplicações paralelas estão mais susceptíveis a serem interrompidas em decorrência de uma falha em algum destes componentes distribuídos do que as aplicações seqüências.

Quando estas aplicações paralelas são interrompidas, seja em razão da falha de algum dos componentes distribuídos ou muitas vezes por causa da própria aplicação paralela, todo o processamento realizado até aquele momento e o tempo gasto para tal são desperdiçados exigindo que, na maioria das vezes, as aplicações sejam reiniciadas.

Para minimizar estes desperdícios de tempo e de processamento realizado torna-se importante a pesquisa por mecanismos que venham a tratar tais problemas, estejam eles nos componentes distribuídos ou nas aplicações paralelas.

O presente trabalho procura amenizar tais desperdícios disponibilizando um ambiente de monitoramento e execução que forneça mecanismos para se detectar falhas da classe *fail stop* em aplicações paralelas, permitindo que as mesmas possam ser reiniciadas automaticamente a partir dos últimos estados armazenados.

O ambiente desenvolvido e apresentado neste trabalho, aqui chamado de AMTF (*Ambiente de Monitoramento Tolerante a Falhas*), utiliza a técnica de *checkpoint/restart* para armazenar e recuperar os estados dos processos, sendo que tais estados são armazenados ou recuperados de arquivos localizados em um sistema de arquivos confiável e estável.

Para viabilizar a tomada e a restauração dos *checkpoints* o ambiente AMTF utiliza o *Berkeley Labs Checkpoint/Restart (BLCR)*, este consiste em um conjunto de aplicativos e bibliotecas que permitem a tomada e a gravação do contexto dos processos em arquivos, porém de maneira manual.

O ambiente AMTF emprega também a técnica de *heartbeat* para realizar o monitoramento das aplicações paralelas e demais componentes do próprio ambiente.

O trabalho apresentado aqui também disponibiliza uma biblioteca para uso por parte do desenvolvedor de aplicações paralelas para que o mesmo possa ter a liberdade de indicar em seu código-fonte onde e quando deseja que os contextos dos processos sejam tomados, bem como sua periodicidade.

Assim, a importância e a principal contribuição deste trabalho reside na possibilidade de viabilizar que arquiteturas paralelas de baixo custo, no caso os *clusters* de computadores, possam contar com mecanismos que, mesmo com o aparecimento de falhas durante a execução das aplicações paralelas, preserve o processamento e o tempo gasto pelas mesmas, reativando-as automaticamente e de maneira transparente ao usuário.

O texto apresenta um levantamento da literatura onde procura classificar os sistemas de processadores paralelos e também conhecer como eles necessitam da tolerância a falhas.

Também descreve questões relativas à disponibilidade, como ela pode ser classificada, além de definições de faltas, falhas e defeitos, o local onde ocorrem e uma visão de níveis de abstrações.

Sobre a tolerância a falhas propriamente dita, o texto apresenta conceitos e definições de termos empregados, questões relativas a tolerância em *hardware* e *software*, bem como as propriedades que se buscam atingir para manter o sistema computacional em segurança e operacional.

O texto também aborda o conceito de formação de grupos de componentes com a intenção de atingir um alto poder de computação e suas implicações quanto ao número de componentes necessários e os modelos de comunicação.

Apresenta o BLCR (*Berkeley Lab's Linux Checkpoint/Restart*), que atua como responsável pela tomada e recuperação de *checkpoints* demonstrando como ele é utilizado e como pode ser empregado conjuntamente como o LAM/MPI.

Apresenta e explica detalhadamente o ambiente AMTF, sua estrutura e módulos componentes, bem como suas respectivas funções e comportamentos esperados, além de uma descrição dos experimentos e testes realizados bem como de seus resultados.

Finaliza indicando algumas possibilidades de trabalhos futuros que podem ser conduzidos a partir do ambiente proposto e dar continuidade à pesquisa.

2. CONTEXTUALIZAÇÃO

Os sistemas de processamento paralelo passaram a ser desenvolvidos em razão da limitação da capacidade de processamento (desempenho) apresentada por sistemas seqüenciais de computação e também devido à necessidade de se obter sistemas que apresentassem melhores resultados referentes à disponibilidade e a confiabilidade.

Surgiram, nessa linha, dois tipos de sistemas de computação que permitiam executar tarefas paralelas e cooperativas, sendo chamados de sistemas logicamente paralelos e sistemas fisicamente paralelos [STALLINGS 1999].

Figura 2.1 - Sistema logicamente paralelo.

Enquanto que os sistemas logicamente paralelos se caracterizavam por intercalar o processamento de várias tarefas em uma única CPU (figura 2.1), os sistemas fisicamente paralelos executavam suas tarefas em múltiplos computadores (figura 2.2) ou em múltiplas CPUs (figura 2.3), compartilhando os demais recursos computacionais.

Na classe de sistemas fisicamente paralelos baseados em múltiplos computadores utilizam-se computadores independentes e que são responsáveis pelas etapas individuais do processamento. Nesta classe surgiram também sistemas compostos por computadores interligados fracamente através de subsistemas de comunicação.

Figura 2.2 - Sistema fisicamente paralelo (múltiplos computadores).

Figura 2.3 - Sistema fisicamente paralelo (múltiplas CPUs).

Na classe de sistemas fisicamente paralelos baseados em múltiplas CPUs, surgiram os sistemas multiprocessados que, além de utilizarem o compartilhamento de memória e de unidades de entrada e saída, apresentavam um único sistema operacional integrado.

A evolução dos sistemas baseados em múltiplos computadores levou ao aparecimento das redes de computadores, dos sistemas distribuídos e da tecnologia de computação em *cluster* e em *grid* [BERSTIS 2005]. Já os sistemas baseados em multiprocessadores levaram ao aparecimento dos ambientes multiprocessados do tipo UMA (*uniform memory access* - acesso uniforme a memória) e do tipo NUMA (*nonuniform access memory* – acesso não uniforme a memória) [TANENBAUM 2003].

2.1. Sistemas Paralelos e a Tolerância a Falhas

A tolerância a falhas pode ser definida como sendo a habilidade de um sistema de comportar-se de maneira bem definida durante a ocorrência de falhas, procurando minimizar o efeito das mesmas através de seu tratamento. Uma abordagem mais detalhada sobre tolerância a falhas é apresentada na seção 2.5.

Tradicionalmente, a comunidade de computação de alto desempenho não tem dado muita atenção para o problema da tolerância a falhas em plataformas paralelas devido ao fato de que as máquinas paralelas tradicionais não são afetadas freqüentemente pelas falhas ou faltas, sejam elas de *hardware* ou de *software* [PINGALI 2003].

Na questão do *hardware*, as máquinas paralelas normalmente custam muito caro (cerca de milhões de dólares) e os projetistas das mesmas preocupam-se em projetar e construir componentes confiáveis, dedicados e integrá-los de maneira que sejam produzidas plataformas de computação extremamente robustas e, normalmente, para se atingir a disponibilidade, utilizam-se também a replicação destes mesmos recursos de *hardware*.

Já na questão *software*, os mesmos (*softwares*), muitas vezes, também são

replicados, ou seja, caso a execução de uma aplicação venha a falhar existem outras execuções, idênticas a ela, em andamento em outros componentes de *hardware*, garantindo assim a continuidade do processamento.

Porém, alguns questionamentos têm sido apresentados no mundo destas plataformas paralelas e estão fazendo com que a preocupação com a tolerância a falhas, ora vista como secundária, venha a ser tratada com maior relevância [PINGALI 2003].

Como exemplos destes questionamentos podem ser citados:

- a) o aumento considerável na quantidade de processadores utilizados por estas plataformas,
- b) o deslocamento dos sistemas de *hardware* caros para sistemas de custos mais baixos e,
- c) o fato de que algumas aplicações necessitam ser executadas por longos períodos de tempo.

A preocupação com estes questionamentos tem aumentado em virtude de que, para atender a estas necessidades por capacidade de processamento e tempo de execução mais longo têm-se agregado cada vez mais processadores às plataformas paralelas, fazendo com que cada um destes novos processadores torne-se um ponto de falha em potencial (como gargalos e queda no processamento) e eventualmente possa comprometer o funcionamento do todo em caso de uma interrupção local.

Também é importante considerar o avanço dos sistemas de computação de baixo custo, visto que, utilizando-se de componentes de computação e comunicação comuns ao mercado, os mesmos podem surgir como uma alternativa economicamente viável para atender a estas demandas, porém, tais componentes

não garantem a confiabilidade disponibilizada, normalmente, pelos componentes especialmente projetados, necessitando assim de maneiras alternativas de mantê-los estáveis e operacionais.

2.2. Sistemas Distribuídos e Tolerância a Falhas

Uma possível definição para sistemas distribuídos seria a de uma coleção de computadores autônomos, conectados através de uma rede e um controle de distribuição que permita coordenar as atividades e compartilhar os recursos do sistema, de maneira que o usuário utilize o sistema como se o mesmo fosse composto por um único computador [EMMERICH 2005].

Os sistemas distribuídos possuem algumas vantagens quando comparados aos sistemas centralizados destacando-se uma melhor relação custo/desempenho, um aumento do poder computacional, a utilização de aplicações projetadas para serem naturalmente distribuídas, a disponibilidade constante dos serviços (de processamento e de arquivos, por exemplo) fornecidos, a distribuição da carga de trabalho pelos equipamentos e também a capacidade de crescimento conforme as necessidades forem surgindo (escalabilidade).

Além dessas vantagens, podem ser citadas algumas características adicionais que os sistemas distribuídos apresentam tais como a multiplicidade de componentes, de pontos de controle e de pontos de falhas sendo que estas considerações acabam por exigir a presença de algumas características como o compartilhamento de recursos, a escalabilidade, a transparência na forma de se interagir com os serviços, a tolerância a falhas e a replicação entre outras.

Pelo fato dos sistemas distribuídos possuírem múltiplas partes de *hardware* e de *software* funcionando conjuntamente, as chances de alguma dessas partes falhar

é bem maior do que ocorreria em um sistema simples ou único.

Dentre os possíveis pontos de falhas que podem surgir nos sistemas distribuídos podem ser destacadas as falhas na rede de comunicação, no *hardware* e no *software*. Desta forma, o sistema deve ser capaz de manter a disponibilidade mesmo com os baixos níveis de confiabilidade destes pontos de falha.

Esta tolerância a falhas pode ser obtida através de procedimentos de recuperação ou então através da redundância. Estes procedimentos são discutidos ao longo do trabalho.

2.3. Disponibilidade

A disponibilidade de um sistema computacional consiste na probabilidade de que o mesmo esteja funcionando, ou disponível, e pronto para uso em um determinado instante de tempo, sendo que o nível de disponibilidade permite uma classificação em disponibilidade básica, alta disponibilidade e disponibilidade contínua [RESNICK 2003].

A *disponibilidade básica* consiste em uma classe de sistemas que são projetados e implementados com componentes (*hardware*, *software* e procedimentos) comuns, procurando satisfazer aos requisitos mínimos de funcionamento e nada mais do que isso. Geralmente pode ser encontrada em computadores convencionais, sem nenhum mecanismo especial de detecção ou replicação, seja em *software* ou *hardware*, que vise de alguma forma mascarar as eventuais falhas destas máquinas [RESNICK 2003].

Na *alta disponibilidade*, os sistemas são projetados e implementados utilizando-se componentes com características suficientes para satisfazer os

requisitos de funcionamento, porém possuindo também redundância nos mesmos buscando mascarar certos tipos de faltas e/ou falhas. Dessa maneira, estes sistemas obtêm a habilidade de restaurar a operação correta, permitindo a continuidade no fornecimento dos serviços durante o período em que um componente tenha falhado [RESNICK 2003] [BIRMAN 1995].

Já a *disponibilidade contínua* aproxima-se ainda mais da disponibilidade ininterrupta, diminuindo assim o tempo de inoperância do sistema de maneira que este venha a ser desprezível ou mesmo inexistente [BIRMAN 1995]. Ela utiliza os mesmos mecanismos da alta disponibilidade e procura manter-se disponível em praticamente todas as paradas, sejam elas planejadas ou não; porém torna-se imperativa a replicação de componentes [RESNICK 2003].

Como se pode observar a partir da definição sobre disponibilidade, o objetivo principal da mesma está em buscar uma maneira de manter os serviços prestados por um sistema ou recursos computacionais disponíveis para outros elementos mesmo que o sistema em questão venha a modificar-se ou alterar-se internamente em razão de uma falha ou falta.

Dentro deste objetivo da disponibilidade está implícito o conceito de mascaramento de falhas, obtido através da redundância ou replicação de recursos. Para tanto, um serviço ou recurso que se deseje altamente (ou continuamente) disponível deve ser colocado por trás de uma camada de abstração, que permita ajustes em seus mecanismos internos mantendo intacta a interação com os elementos externos.

Assim, no centro da disponibilidade contínua ou alta encontra-se a área de tolerância a falhas, que procura manter a continuidade dos serviços prestados por um sistema computacional e seus respectivos recursos, através de redundância de *hardware* e reconfiguração de *software*, mesmo na presença de falhas.

Esta redundância pode ser conseguida quando se têm vários componentes

replicados interligados, agindo como se fossem apenas um (transmitindo a imagem de um sistema único), porém, cada um monitorando os demais e assumindo os serviços e tarefas de seus parceiros caso perceba que algum deles falhou.

Uma característica importante da disponibilidade seja ela alta ou contínua, é a de manter o sistema com componentes simples, encontrados em lojas do ramo, também conhecidos como componentes de prateleira (*off-the-shelf*). Isso faz com que os mesmos sejam facilmente encontrados e estejam acessíveis a um preço aceitável.

2.4. Falta, Falha e Defeito

Torna-se importante definir alguns termos utilizados no contexto da disponibilidade e tolerância a falhas. Inicia-se aqui pelos termos falta (*fault*), falha (*failure*) e defeito.

Estes termos, que parecem tão próximos e são muitas vezes interpretados de maneiras variadas, designam a ocorrência de algo anormal em três universos diferentes de um sistema computacional [GÄRTNER 1999].

A *falta* acontece no chamado nível mais baixo de abstração, ou seja, no universo denominado físico, ou ainda no *hardware* do sistema de computação.

Como exemplos de faltas podem ser citados uma flutuação na fonte de alimentação, uma interferência eletromagnética ou um problema em uma célula de memória. Estes são eventos indesejáveis que acontecem no universo físico e afetam o funcionamento de um computador ou partes dele.

A falta pode ocorrer quando o componente comporta-se de maneira

inesperada ou então é interrompido e pode ser classificada como falta *por omissão*, *por tempo* ou *por resposta* [CRISTIAN 1991].

A falta por omissão ocorre quando um componente omite-se a atender uma entrada ou uma solicitação. Já a falta por tempo ocorre quando a resposta do componente para uma determinada entrada ou requisição está funcionalmente correta, porém fora do tempo esperado, este tipo de falta pode ser sub-dividido em *falta por antecipação*, onde a resposta chega antes do tempo esperado ou *falta por atraso*, onde a resposta chega após o tempo previsto. E finalmente a falta por resposta, que ocorre quando o componente responde incorretamente, ou seja, a resposta ou alguma operação realizada pode estar incorreta.

A ocorrência de uma falta pode acarretar uma *falha* [GÄRTNER 1999] sendo a falha caracterizada como a representação da falta no universo informacional (sistema operacional e aplicação) e acontece no denominado nível intermediário de abstração, ou seja, no nível da informação.

Como exemplo de uma falha tem-se que a alteração de um *bit* de um determinado *byte* pode fazer com que uma instrução seja interpretada de maneira errônea, provocando um erro na representação de alguma informação.

Esta informação errônea, se não for percebida ou tratada, poderá gerar o que se conhece por *defeito*. Neste caso, o sistema pode simplesmente travar, ou exibir alguma mensagem de erro, ou ainda, perder os dados do usuário sem maiores avisos. Isto é percebido no nível mais alto de abstração, ou seja, no universo do usuário.

Desta forma, tem-se que uma falta ocorre no universo físico e pode causar uma falha no universo informacional, que por sua vez pode causar um defeito percebido no universo do usuário, conforme demonstra a figura 2.4.

A tolerância a falhas procura atuar nos níveis baixo e intermediário de

abstração, mascarando o aparecimento das faltas ou então recuperando as informações quando da ocorrência das falhas.

No restante deste trabalho os termos falta e falha são utilizados como sinônimos para sinalizar tanto faltas reais como falhas decorrentes destas mesmas faltas, pois deseja-se tratar as faltas quando elas já se tornarem falhas.

Figura 2.4 - Níveis de abstração (falta, falha e defeito).

2.5. Tolerância a Falhas

A tolerância a falhas, que pode ser definida como sendo a habilidade de um sistema comportar-se de maneira bem definida durante a ocorrência de falhas, visa minimizar o aparecimento das mesmas ou então tratá-las quando ocorrerem.

2.5.1. Classes da Tolerância a Falhas

Um componente pode ser considerado como falho quando apresentar um

comportamento diferente daquele definido em sua especificação. Esse comportamento pode ser enquadrado nas classes conhecidas por *crash failure*, *fail stop* e *bizantino* [SCHNEIDER 1990] [CRISTIAN 1991].

Na classe *crash failure* o componente interrompe sua execução em algum ponto do processamento por um determinado período de tempo, mas retorna a sua execução (como se ocorresse uma pausa no processamento). Esta classe pode ser subdividida em *amnesia crash failure*, *partial-amnesia crash failure* e *pause-crash*.

- A *amnesia crash failure* ocorre quando o componente reinicia em um estado inicial pré-definido que não depende das entradas recebidas antes da queda.
- *Partial-amnesia crash failure* ocorre quando, durante o reinício, parte do estado é a mesma que antes da queda, enquanto que o resto do estado é inicializado para um estado inicial pré-definido.
- *Pause-crash* ocorre quando o componente reinicia no estado que possuía antes da queda.

Já na classe *fail stop* [SCHNEIDER 1983], o componente apresenta uma falha e seu funcionamento é prontamente interrompido, não voltando mais a funcionar, sendo que isto é detectado pelos demais componentes.

E na última classe, o *bizantino* [LAMPORT 1982], o funcionamento do componente torna-se instável, imprevisível, comportando-se de maneira errônea ou não, ou então maliciosa.

Quando se projetam sistemas tolerantes a falhas, um primeiro pré-requisito está em especificar qual ou quais as classes (*crash failure*, *fail stop* ou *bizantino*) devem ser toleradas e em seguida dotar o sistema com componentes e mecanismos que forneçam a proteção contra as falhas das classes escolhidas.

2.5.2. Redundância

Em sistemas tolerantes a falhas, os efeitos das falhas e interferências indevidas podem ser superados através da redundância temporal e/ou física.

A redundância temporal corresponde à determinação de um resultado através de execuções repetidas da mesma operação pelo mesmo elemento, usando, eventualmente, métodos diferentes.

Já a redundância física consiste em ter-se elementos repetidos capazes de executar a mesma operação. Esses elementos referem-se tanto ao *hardware* como ao *software* do sistema.

2.5.3. Propriedades *Safety* e *Liveness*

Tomando-se como base o funcionamento e a disponibilidade de sistemas distribuídos e *clusters*, além de sua alternância entre os estados funcionando e em reparo, alguns autores definem duas grandes propriedades para os respectivos sistemas. Estas propriedades referem-se ao conjunto de execuções que os sistemas realizam e são denominadas *safety* e *liveness* [GÄRTNER 1999].

A propriedade *safety* pode ser caracterizada formalmente especificando-se quais execuções são seguras ou não, garantindo que o resultado emitido por elas esteja correto através da execução de algum procedimento de detecção e ativação de procedimentos de correção, como mecanismos de *checksum*, por exemplo.

Já a propriedade *liveness* indica que o processamento parcial de um sistema estará funcionando (*live*), para um conjunto de processamento maior se e somente se ele puder ser estendido, ou seus resultados puderem ser aproveitados pelo restante do processamento do sistema ou componente, em outras palavras, preocupa-se em manter o sistema ou componente apto a executar, sem preocupar-

se com os resultados emitidos pelo mesmo, mas obrigando-o a emitir um resultado.

Em resumo, a propriedade *liveness* procura garantir que o sistema ou componente mantenha-se sempre funcionando, operacional e disponível e a propriedade *safety* procura garantir que o resultado da execução feita pelo componente ou sistema esteja sempre correta.

2.5.4. Formas de Tolerância a Falhas

A maneira como um componente se comporta em uma situação de falha e o estado que ele venha ou não garantir no decorrer da mesma determina quais propriedades (*safety* e/ou *liveness*) da tolerância a falhas e suas respectivas formas são atingidas, conforme indicado na figura 2.5 [GÄRTNER 1999].

Figura 2.5: Formas de Tolerância a Falhas [GÄRTNER 1999].

Em um dos extremos, como no *mascaramento* das falhas, estas não interrompem o funcionamento e não corrompem a integridade do sistema, assim as propriedades *safety* e *liveness* são atingidas em sua plenitude.

Já no outro extremo, a forma *nenhuma*, a falha pode corromper o

funcionamento e a integridade do processamento, fazendo com que as propriedades *safety* e *liveness* não sejam atingidas.

Já nos casos intermediários, o *não mascaramento* deixa o sistema funcionando (*liveness*), mas possivelmente não seguro quanto às respostas emitidas e o *fail safe* deixa o sistema seguro no que se refere ao resultado do processamento (*safety*), mas possivelmente não operacional.

Ainda sobre o mascaramento, ele pode ser classificado em *mascaramento hierárquico de falhas* ou *mascaramento de falhas em grupo*.

No mascaramento hierárquico de falhas, um comportamento falho pode ser classificado somente com respeito a uma certa especificação de comportamento que se espera dele. Se um componente depende de um nível mais baixo para fornecer um determinado serviço, então uma falha neste nível mais baixo acarretará em uma falha no oferecimento do respectivo serviço pelo nível mais alto, sendo que, se esta falha for então mascarada, não será vista pelo nível mais alto e o serviço em questão não será afetado [CRISTIAN 1991].

Já no mascaramento de falhas em grupo, para se tentar garantir que um serviço continue disponível para os usuários, independentemente das falhas de seus componentes, pode-se implementar o serviço através de um grupo de componentes redundantes e fisicamente independentes, isso garantirá que, no caso de um deles falhar, o restante continue fornecendo o serviço.

Um pré-requisito para a implantação de um serviço de *software* por um grupo capaz de mascarar falhas está na existência de múltiplos processadores *hosts* (*hosts processors*) com acessos aos mesmos recursos físicos utilizados pelo serviço em questão [CRISTIAN 1991]. A replicação destes recursos faz-se necessária para tornar o serviço disponível independente das falhas dos recursos individuais.

2.5.5. Processo de *Failover* e *Failback*

O processo no qual um componente assume os serviços de outro, quando este último apresentar falhas é chamado de *failover* [BIRMAN 1995]. Este procedimento pode ser feito automaticamente ou manualmente, sendo a maneira automática a que normalmente se espera de uma solução com disponibilidade alta ou contínua.

Além do tempo entre a ocorrência da falha e a sua detecção, existe também o tempo entre a detecção e o reestabelecimento do serviço. Dependendo da natureza do serviço, executar o *failover* pode significar a interrupção das transações em andamento. As transações podem ser perdidas e neste caso, sendo as mesmas necessárias, devem ser retomadas ou reiniciadas após o *failover*.

Ao ser percebida a falha de um componente, além do *failover*, é necessário que se faça sua manutenção. Ao ser reparado, o componente deverá ser reintegrado ao serviço, e então realizar o processo inverso ao *failover*, denominado *failback* [BIRMAN 1995]. Assim, o *failback* é o processo de retorno de um determinado serviço ou componente ao seu estado original e também pode ser automático ou manual. Todos estes procedimentos de *failover* e *failback* estão representados na figura 2.6.

Figura 2.6: Procedimentos de *failover* e *failback*.

Desta forma, em um sistema tolerante a falhas, caso um componente

apresente uma falha isto deve ser detectado por outro componente e ele (componente falho) deve ser isolado e então reparado ou substituído por um novo componente. Caso este também venha a apresentar uma falha, será substituído por outro e assim sucessivamente.

Ainda com base na figura 2.6, durante a sua vida útil, um componente pode ser considerado como estando em um destes dois estados: *funcionando* ou *em reparo*. O estado *funcionando* indica que o componente está operacional e o estado *em reparo* significa que o mesmo falhou e ainda não foi substituído por um novo.

Em caso de problemas, o componente vai do estado *funcionando* para o estado *em reparo*, e quando a substituição é feita, volta para o estado funcionando. Sendo assim, pode-se dizer que o componente apresenta ao longo de sua vida um tempo médio até apresentar falha (MTTF - *Mean Time To Failure*) e um tempo médio para reparo (MTTR - *Mean Time To Repair*) [RESNICK 2003] [GARTNER 1999].

O tempo de vida de um componente é uma seqüência de MTTFs e MTTRs à medida que apresenta falhas e é recuperado. A disponibilidade do componente pode ser obtida pela relação entre o tempo de vida útil do componente e seu tempo total de vida. Isto pode ser representado por:

$$Disponibilidade = \frac{MTTF}{(MTTF + MTTR)}$$

2.5.6. Abordagens de Projeto e Arquitetural

Um sistema pode apresentar maior confiabilidade e disponibilidade se for desenvolvido com o uso de técnicas adequadas para isso [KIRNER 1988].

Quanto à abordagem de projeto, podem ser encontradas duas maneiras de buscar o aumento do grau de confiabilidade, de disponibilidade e de tolerância a

falhas de um sistema, além de tentar atingir as propriedades *safety* e *liveness*. A primeira maneira atua no sentido de impedir o aparecimento das falhas e a segunda no sentido de, caso não seja possível impedi-las, então tolerá-las.

Com respeito à abordagem arquitetural, ela pode ser observada sob dois pontos de vista, o de *hardware* e o de *software*, detalhados a seguir.

2.5.6.1. Abordagem Arquitetural de *Hardware*

Na abordagem arquitetural de *hardware* algumas arquiteturas "concentram" recursos básicos como CPU, memória, controladores de E/S e de comunicação em componentes denominados *unidades reposicionáveis simples* (*single replaceable units*) e outras em unidades reposicionáveis de recursos individuais, somente para discos, CPUs e memórias, individualmente [CRISTIAN 1991].

Uma unidade reposicionável de *hardware* é uma unidade física passível de sofrer uma falha (também conhecida como um possível ponto de falha) e que pode ser substituída por outra unidade.

Além de ser escalável, ela também é considerada uma unidade independente das demais e pode ser removida do conjunto sem afetar as demais unidades e pode também ser adicionada ao sistema para aumentar seu desempenho, capacidade de atendimento e disponibilidade.

Estas unidades reposicionáveis podem ser classificadas em *unidades reposicionáveis de campo*, onde um técnico é responsável pela troca das mesmas, e em *unidades reposicionáveis do consumidor*, onde a substituição pode ser feita pelo próprio usuário; assim, sua substituição pode ser feita rapidamente.

2.5.6.2. Abordagem Arquitetural de Software

Na abordagem arquitetural de *software*, os serviços de *software* têm o funcionamento análogo às unidades reposicionáveis de *hardware*, ou seja, existem unidades básicas de falhas, substituíveis e escaláveis de *software*.

O objetivo principal destas unidades escaláveis é serem removidas do sistema quando apresentarem falhas, sem interromper os demais serviços e atividades dos usuários.

2.5.7. Tolerância a Falhas em Software

Procura-se obter a tolerância a falhas em *software* utilizando-se algumas técnicas como registros em *logs* (*logging*) [JOHNSON 1989], pontos de verificação (*checkpointing*) [PARK 2000] [KIM 2002] [JOHNSON 1989] [VAIDYA 1999] [LUMPP 1998] [MAIER 1993] [DOW 2002] [SILVA 1998] [PINGALI 2003], espelhamento (*mirroring*) [CRISTIAN 1991], gravações atômicas (*atomic commit*) [RAYNAL 2001] e rejuvenescimento [DOSS 2001].

2.5.7.1. Técnica de Registros em Log

A técnica de registros em *log* faz uso de um sistema de armazenamento estável onde são registradas as operações realizadas pelos processos (do processamento ao envio e recebimento de mensagens); este armazenamento é feito em um arquivo (*log*) para que seja possível uma recuperação (*roll back*) futura no caso do aparecimento de alguma falha.

Nesta técnica, quando um processo apresenta uma falha, apenas ele deve ser reiniciado e executar o procedimento de *roll back* utilizando o arquivo de *log*; os

demais processos sobreviventes continuam sua execução como se nada tivesse acontecido, porém podem auxiliar ao processo reiniciado manipulando (enviando ou recebendo) as mensagens utilizadas antes do surgimento da falha.

2.5.7.2. Técnica de *Checkpointing*

A técnica de *checkpointing* permite que programas armazenem seus estados em intervalos regulares para que possam ser reiniciados após uma interrupção, sem perder o trabalho já realizado.

Esta é uma técnica utilizada para tolerar falhas transientes e para se evitar a perda total de um processamento já realizado. Alguns tipos de *checkpoints* apresentados na literatura [SILVA 1998] [DOW 2002] [PINGALI 2003] são descritos ao longo desta seção.

Cada processo pode armazenar apenas o seu estado individual em *checkpoints* chamados locais, sendo uma coleção destes *checkpoints* locais denominada de *checkpoint* global [WANG 1995] [JOHNSON 1989] [LUMPP 1998].

Podem ser encontradas quatro abordagens principais para tolerar falhas que utilizam a técnica de *checkpoint* [SILVA 1998]:

- i) deixar que o programador se preocupe em tratar as falhas através do próprio programa utilizando chamadas que realizem o tratamento de exceções;
- ii) fornecer suporte via compilador;
- iii) o sistema operacional deve efetuar automaticamente o *checkpointing* e a recuperação, liberando o programador, sendo que este deve apenas especificar o intervalo entre os *checkpoints*;

- iv) o suporte a *checkpoints* é fornecido por uma biblioteca. Esta abordagem requer que o programador adicione os pontos onde os *checkpoints* serão realizados diretamente no código-fonte.

A abordagem (i) pode transformar-se em uma tarefa complexa e, deixar as rotinas de tratamento das falhas a cargo do programador pode resultar em um aumento significativo no volume de código-fonte a ser produzido [SILVA 1998].

A abordagem (ii) torna-se interessante pois delega-se a tarefa de adicionar os pontos de tomada de *checkpoints* para o compilador, sendo que os mesmos serão posicionados no melhor lugar possível, buscando reduzir o tamanho do arquivo de *checkpoints* e o momento de sua tomada. A dificuldade nesta abordagem reside na portabilidade do compilador e na utilização do mesmo em ambientes distribuídos e/ou paralelos diferenciados por conta de eventuais particularidades que os mesmos possuam.

A abordagem (iii) apresenta a idéia de que o registro e a recuperação das aplicações devem ser feitos automaticamente sem nenhum esforço por parte do programador, ou seja, o próprio sistema operacional está encarregado de implementar e recuperar os *checkpoints*, criando uma situação transparente para o desenvolvedor. A questão negativa deste modelo reside no fato da aplicação ser vista como uma caixa-preta e o *checkpoint* não irá considerar eventuais características específicas da aplicação.

A abordagem (iv) fornece suporte de *checkpoints* através de uma biblioteca. Ela não é transparente para o usuário uma vez que o mesmo deverá explicitamente indicar no código-fonte os locais onde os *checkpoints* devem ser tomados. A desvantagem desta abordagem reside em sua não transparência e a vantagem está no controle daquilo que efetivamente deve ser registrado ou não da aplicação.

As duas últimas abordagens (*checkpoint* transparente no nível do sistema ou definido pelo usuário) têm suas vantagens e podem ser encontradas muitas

discussões sobre a melhor maneira de tratar as falhas, transparentemente pelo sistema operacional ou através de aplicações que utilizam este sistema operacional.

A técnica de *checkpointing*, seja definida pelo sistema ou pelo programador, é muito utilizada em aplicações paralelas e distribuídas e pode ser classificada em duas grandes categorias, a coordenada e a não coordenada [PINGALI 2003] [DOW 2002].

A categoria coordenada apresenta uma maneira de evitar estados inconsistentes após uma recuperação, visto que todos os processos devem estar sincronizados e as mensagens que já tenham sido enviadas deverão atingir seus destinos antes que os *checkpoints* sejam realizados.

Já a categoria não coordenada permite que cada processo possa decidir, de maneira independente, quando tomará seu *checkpoint*. Esta autonomia deve ser vista com cautela, pois ela pode fazer com que uma aplicação necessite reiniciar-se completamente por conta de recuperações realizadas em cascata; estas recuperações sucessivas são chamadas de Efeito Dominó [JOHNSON 2002] sendo que o mesmo está representado na figura 2.7.

Figura 2.7: O Efeito Dominó [JOHNSON 2002].

A figura 2.7 mostra a execução de um sistema composto por dois processos que se comunicam com a progressão do tempo da esquerda para a direita. O estado de cada processo será armazenado toda vez que se encontrar um quadrado, porém

as mensagens não serão armazenadas.

Caso o Processo 1 venha a falhar no ponto indicado pelo círculo, seu estado poderá ser recuperado a partir do último *checkpoint* (quadrado). Uma vez que isso obriga o Processo 1 a retornar ao ponto anterior ao envio da mensagem para o Processo 2, o Processo 2 também será obrigado a voltar ao ponto anterior ao recebimento desta mensagem. O *checkpoint* do Processo 2 obriga o Processo 1 a voltar mais um *checkpoint*, o qual também obriga o Processo 2 a voltar mais um *checkpoint* e assim por diante até que os dois processos retornem ao primeiro *checkpoint*.

A categoria coordenada não é susceptível ao Efeito Dominó devido aos processos sempre reiniciarem de um *checkpoint* global mais recente. A recuperação é simplificada e o dispositivo de armazenamento sofre uma menor carga de trabalho do que a provocada pela categoria não coordenada [DOW 2002].

Ainda sobre as categorias dos *checkpoints*, a categoria coordenada pode ser subdividida em bloqueante e não bloqueante.

A forma bloqueante faz com que todos os processos interrompam suas execuções antes de um *checkpoint* global ser realizado. Técnicas de bloqueio por *software* utilizam a técnica de barreira, ou seja, quando os processos atingem uma barreira global, cada um armazena seu estado local e então é criado um *checkpoint* global, em seguida os processos continuam suas execuções [PINGALI 2003].

Já na forma não bloqueante, os processos armazenam seus estados e suas mensagens individualmente, necessitando para isso um protocolo de coordenação global; o protocolo Chandy-Lamport [CHANDY 1985] é o mais usado dos protocolos não bloqueantes [PINGALI 2003].

Alguns tipos de técnicas buscam reduzir a carga produzida pelos *checkpoints* no sistema. Como exemplos destas técnicas podem ser citados o *forked*

checkpointing, o *incremental checkpointing*, o *main memory checkpointing* e o *compressed checkpointing*.

O *Forked Checkpointing* utiliza uma instrução *fork* para criar um processo filho que se encarrega de tomar os *checkpoints* gravando-os sempre em um mesmo arquivo, sendo que, enquanto o contexto do processo estiver sendo gravado no arquivo, o mesmo deverá estar parado ou suspenso. Isto torna-se necessário em razão de se preservar o estado local do processo, ou seja para que, entre a tomada do estado local e a efetiva gravação do mesmo, não ocorra nenhuma alteração.

Já no *Incremental Checkpointing* cria-se um arquivo de *checkpoint* principal o qual armazena os estados e alguns outros arquivos de *checkpoints* secundários que armazenam as diferenças entre os estados atuais e os anteriores.

O *Main Memory Checkpointing* utiliza um *thread* para realizar uma cópia do espaço de dados do programa no arquivo de *checkpoint* sem interromper o processo que está sendo registrado.

E o *Compressed Checkpointing* utiliza um utilitário de compressão de dados para diminuir o tamanho do arquivo de *checkpoint*.

As técnicas de *checkpointing* e registro em *log* podem ser classificadas sob dois esquemas, conhecidos como *pessimista* e *otimista* [MAIER 1993] [TREASTER 2005].

O modo pessimista registra cada processamento, transmissão ou recepção de mensagens que o processo realizar, sincronizadamente. Isto aponta para uma rápida recuperação, porém aponta também para uma sobrecarga no sistema ao se registrar praticamente todas as ações e isso pode ser proibitivo pois pode ocasionar uma diminuição no desempenho das aplicações. Este modo recebe o nome de “pessimista” por assumir uma alta probabilidade de ocorrência de falhas em sistemas distribuídos ou paralelos.

Já o modo otimista registra o processamento, as transmissões e recepções apenas de tempos em tempos, assíncronamente. Isto causa uma pequena sobrecarga durante o processamento normal da aplicação e permite que os processos determinem o momento em que se deve ou não registrar seus estados e mensagens. O termo “otimista” vem do pressuposto de que o aparecimento de falhas tem uma baixa probabilidade de ocorrer em sistemas distribuídos ou paralelos.

A recuperação para ambos os modos é feita apenas quando se deseja recuperar o estado armazenado no último *checkpoint*.

2.5.7.3. Técnica de Espelhamento

A técnica de espelhamento consiste em executar várias cópias das mesmas instruções em vários elementos de processamento para verificar se seus resultados são os mesmos.

Os resultados emitidos pelos vários processadores são comparados uns com os outros e a resposta que possuir um maior número de ocorrência será a aceita (maioria votante).

Além de garantir a tolerância a falhas em decorrência da replicação, garante-se também a alta confiabilidade da resposta, bem como a alta disponibilidade de processamento.

2.5.7.4. Técnica de Gravação Atômica

Quanto à técnica gravação atômica, um procedimento iniciado pelos participantes de um serviço apenas será concluído quando todos os participantes concluírem seus procedimentos individuais e esta técnica é, tradicionalmente,

empregada em serviços de banco de dados.

Pesquisadores têm proposto protocolos para esta técnica e implementado os mesmos em alguns sistemas. Infelizmente algumas destas implementações exibem uma propriedade bloqueante em caso de falhas.

O bloqueio em questão é presenciado nos participantes que devem aguardar pela recuperação do participante que falhou para concluir o procedimento executado antes da falha [RAYNAL 2001].

2.5.7.5. Técnica de Rejuvenescimento

A técnica de rejuvenescimento apresenta que, quando o *software* é executado continuamente, ele começa a "envelhecer" [DOSS 2001] sendo que este envelhecimento ocorre devido às condições de erros que se acumulam com o tempo e com o uso.

Exemplos desse envelhecimento são a fragmentação excessiva e as áreas de memórias que não são liberadas após o uso, o estouro nos tempos de escalonamento da CPU, a perda de ponteiros, o mau uso dos registradores, as falhas no arredondamento entre outras.

As falhas ocorrem devido à deterioração dos recursos do sistema operacional, como registros inicialmente travados e não mais liberados, além de dados corrompidos e a indisponibilidade de certos recursos [DOSS 2001].

Esta técnica é considerada como preventiva e defende que, para se evitar o aparecimento de falhas, deve-se interromper a execução do *software* periodicamente, armazenando seus estados para recuperação futura e então reiniciá-lo.

2.5.8. Considerações sobre a Tolerância a Falhas em *software*

Para se atingir a tolerância a falhas em *software* deve-se analisar algumas questões, sendo que [CRISTIAN 1991] apresenta três observações que devem ser feitas.

A primeira delas preocupa-se em como os membros de um grupo executando em diferentes processadores irão manter a consistência de seus estados locais na presença de membros com falhas, com a junção de novos membros e com falhas ocorrendo na comunicação.

Já a segunda observação aponta para a quantidade de membros necessários para manter um grupo em funcionamento, independente das falhas que vierem a ocorrer com o sistema operacional, com os membros dos grupos ou com a comunicação.

E a terceira preocupa-se em definir como os servidores do grupo devem comunicar-se, definindo-se um protocolo para a transferência de mensagens.

Para qualquer serviço, a política de sincronização dos membros do grupo deve descrever o grau de sincronização do estado local do serviço, e este deve ser o mesmo em todos os membros que compõem o grupo ou disponibilizam um determinado serviço. Esta política de sincronização pode ser de dois tipos: *close synchronization* e *loose synchronization* [PARK 2000] [CRISTIAN 1991].

No primeiro caso (*close synchronization* – também conhecida por mascaramento ou redundância ativa), os estados dos membros estão diretamente sincronizados uns com os outros, fazendo com que todos os membros executem os mesmos serviços requisitados em paralelo e efetuem a mesma sequência de transições de estados.

Neste caso, as saídas geradas serão na mesma proporção do número de membros do grupo; deve-se então possuir uma política de validação do resultado.

Um exemplo desta política implementada poderia ser a da maioria votante, onde o resultado válido seria aquele emitido pelo maior número de participantes do grupo, caso a maioria dos membros venha a falhar, a saída do grupo poderá ser incorreta. Uma outra alternativa poderia ser obtida considerando o resultado do membro mais rápido ou então o mais poderoso em termos de recursos computacionais.

No segundo caso (*loose synchronization* – também chamada de redundância de espera ou dinâmica), faz-se uma classificação dos membros do grupo com respeito a como eles estão sincronizados com o estado mais atual do serviço corrente. Nesta classificação elege-se um membro para ser o servidor primário e este será responsável por receber as requisições de serviço, registrar o estado do serviço como um estado local e repassá-los, estado e requisições, para os demais componentes do grupo.

Sobre a quantidade de membros necessária para compor um grupo, tem-se que, quanto maior o número de membros no grupo, maior será a capacidade de executar a requisição paralelamente e um maior número de falhas serão toleradas. Porém, quanto maior o número de membros, maior o custo com a comunicação a ser realizada para a sincronização entre os membros, além do custo da redundância propriamente dita.

2.5.9. Comunicações Síncrona e Assíncrona

Uma vez que os sistemas com alta disponibilidade ou disponibilidade contínua podem disponibilizar serviços para aplicações com tempos rígidos ou flexíveis, torna-se interessante entender como os serviços de *comunicações síncronas* (com tempo rígido) e *assíncrona* (com tempo flexível) podem ser utilizados [RAYNAL 2001].

O modelo de sistemas síncronos assume a certeza de que a comunicação entre os membros ocorre dentro de um intervalo de tempo especificado [CRISTIAN

1996]. Normalmente, uma rede de comunicação é considerada de difusão-síncrona se conseguir garantir que alguma transmissão iniciada por um membro operacional atingirá todos os demais membros dentro de um tempo esperado.

Já no modelo assíncrono, os membros do grupo trocam mensagens através de um serviço de comunicação por transmissão e recepção de pacotes sobre um subsistema de comunicação.

No modelo assíncrono, admite-se que a mensagem pode sofrer atrasos para atingir o seu destino em decorrência de um possível volume de tráfego excessivo no subsistema de comunicação ou ainda perder-se ao longo do caminho em razão de alguma instabilidade no meio de comunicação. Também é conhecido que os processadores podem atrasar-se na execução e emissão de resultados devido ao tempo gasto pelos mesmos para realizar o escalonamento entre os processos.

O modelo assíncrono pode ainda ser subdividido em duas outras modalidades, a assíncrona temporizada (ou *timed*) e a assíncrona de tempo livre (ou *free time*).

O serviço especificado na modalidade temporizada descreve não somente as transições de estados e saídas que ocorrem em resposta a algum evento, mas também o intervalo de tempo necessário para que elas ocorram (modalidade determinística), caso contrário serão abortadas por tempo (*timeout*).

Já na modalidade de tempo livre, para cada estado ou entrada há somente um próximo estado ou saída, sem qualquer imposição de tempo. Elas simplesmente ocorrerem em algum momento, no próximo minuto, no próximo ano, ou algo assim (modalidade não determinística). Não se pode garantir que os membros do grupo estão funcionando, pois eles tem um tempo arbitrário para responder.

Analisando pelo tempo de resposta aos estados e pelas entradas pode-se verificar que torna-se um tanto difícil a implementação de serviços tolerantes a falhas

em sistemas assíncronos de tempo livre [CRISTIAN 1996]. Porém, os mesmos serviços podem ser implementados em sistemas assíncronos temporizados desde que certas condições, como o baixo congestionamento enfrentado nas comunicações, a estabilidade do grupo e a disponibilidade do serviço, sejam mantidas.

2.5.10. Sistemas de Arquivos Tolerantes a Falhas

A existência de um sistema de arquivos estável e tolerante a falhas é imprescindível em razão de que os arquivos produzidos e recuperados pelas técnicas de tolerância em *software* serão armazenados e recuperados do mesmo, além dos demais programas e dados que venham a ser compartilhados pelos usuários. Assim, é importante que ele esteja sempre disponível para atender às requisições e, quando existirem, consiga atender o volume necessário em um tempo aceitável.

Para se garantir a tolerância a falhas e, conseqüentemente, a alta disponibilidade no âmbito de discos e sistemas de arquivos, várias soluções podem ser empregadas.

Algumas destas soluções são viabilizadas através de *hardware*, tais como a técnica de RAID (*Redundant Array of Independent Disks*) [RAID 2003], multiplexação de discos e replicação de primeira classe e outras viabilizadas por *software* como por exemplo a replicação de segunda classe [KRISTLER 1990] [SATYANARAYANAN 1989].

No caso da replicação de primeira classe, o servidor é completamente replicado e, conseqüentemente, o sistema de arquivos também e na replicação de segunda classe utiliza-se a técnica conhecida como *caching* do cliente ou *journal file system*; são exemplos desta replicação os sistemas de arquivos CODA [SATYANARAYANAN 1989] [CODA 2003] [BRAAM 1999] e AFS [FAQ 2003] [NCSA

2003] [AFS 2003] [IBM 2003].

Os dois tipos são complementares, sendo que a replicação do servidor aumenta a disponibilidade dos arquivos armazenados e o *caching* permite a operação durante períodos de desconexão do sistema.

Estes períodos de desconexão experimentados pelos clientes podem ser voluntários, ou seja, devido à retirada do computador do subsistema de comunicação ou involuntários, por conta de falhas nos servidores replicados ou no próprio subsistema de comunicação.

Os períodos de desconexão devem passar despercebidos pelos clientes, principalmente os causados involuntariamente; para tanto torna-se necessária uma operação de desconexão transparente.

Para obter essa operação transparente os clientes normalmente implementam três fases no sistema de gerenciamento de seus caches, são as fases de acumulação, de emulação e de reintegração [KRISTLER 1990], conforme pode ser observado na figura 2.8.

Figura 2.8: Fases do sistema de gerenciamento de caches em *journal*.

Durante a fase de acumulação, os clientes estão conectados ao subsistema

de comunicação e também aos servidores do sistema de arquivos e assim mantêm atualizados os seus caches locais.

Na fase de emulação, os clientes estão desconectados do subsistema de comunicação e, conseqüentemente, dos servidores do sistema de arquivos. Todos os acessos requeridos pelos clientes serão atendidos pelos seus caches locais, emulando a presença dos servidores. Caso o cliente solicite algum arquivo que não esteja em seu cache local, receberá uma notificação da indisponibilidade do serviço de arquivo.

E na fase de reintegração, os clientes voltam a estar conectados ao subsistema de comunicação e, conseqüentemente aos servidores, e então, devem sincronizar seus caches locais com os servidores do sistema de arquivos. Nesta fase podem ocorrer conflitos durante a sincronização, desta forma, procedimentos automatizados pelo próprio sistema de arquivos podem ser tomados, ou então solicitar a intervenção manual do operador para propor soluções.

No que diz respeito aos procedimentos automatizados de reintegração, uma das técnicas empregadas é a de refazer os passos registrados em um *log* do *pseudo*-servidor com os passos registrados no *log* do sistema ao qual ele deverá ser reintegrado.

Caso não existam inconsistências ou o usuário que está sendo reintegrado não possua mais direito de reintegração devido a um grande período de ausência, a junção ou não junção ocorrerá sem maiores problemas, ou seja, no primeiro caso, ocorrerá a sincronização e no segundo caso ela não ocorrerá.

Caso a inconsistência ainda permaneça, pode-se utilizar uma heurística baseada na comparação de nomes dos arquivos, datas de alterações, quantidades de alterações ou o último autor de mudanças, por exemplo.

Se esta heurística ainda não conseguir resolver os problemas, pode-se

necessitar da intervenção do usuário para realmente oficializar ou não a junção; nesta situação corre-se o risco de perder possíveis alterações no sistema de arquivos [KRISTLER 1990].

2.5.10.1. RAID - *Redundant Array of Independent Disks*

A técnica empregada no RAID consiste em agrupar uma série de discos em um único local e disponibilizá-los na forma de uma imagem única de sistema de armazenamento e recuperação de informações [RAID 2003].

Sua implementação pode ser viabilizada por *hardware*, ou seja, um dispositivo com finalidade única de agrupar os discos e realizar o gerenciamento dos mesmos, estando este dispositivo ligado através de um barramento de alta velocidade aos nós de computação.

Uma outra modalidade de RAID é a implementação por *software*, sendo que o agrupamento dos discos é feito dentro de um computador hospedeiro e a função de gerenciamento é realizada por intermédio de uma aplicação processada por este mesmo computador. O acesso de outros computadores à respectiva imagem do disco é feito através de um subsistema de comunicação.

As configurações de RAID mais comuns são as de nível 1 (figura 2.9), onde ocorre o espelhamento dos discos e a de nível 5 (figura 2.10), onde algumas partições dos discos são utilizadas como áreas de armazenamento para a recuperação de informações.

A configuração de RAID de nível 0 (figura 2.11) também é bastante comum, porém não implementa mecanismos de recuperação em caso de falhas de alguma unidade pois apenas "soma" os espaços de armazenamento.

A multiplexação de discos complementa a técnica de RAID. Ela consiste em

utilizar duas ou mais controladoras de discos dentro de um mesmo computador para que as mesmas atuem como imagens umas das outras, sendo que estas controladoras podem gerenciar um único disco por controladora ou até um RAID por *software*.

Figura 2.9: RAID de Nível 1 [ACNC 2004].

Figura 2.10: RAID de Nível 5 [ACNC 2004].

Figura 2.11: RAID de Nível 0 [ACNC 2004].

No caso de uma das controladoras falhar, ela será isolada imediatamente das demais e as controladoras sobreviventes irão manter o sistema de arquivos disponível.

2.5.10.2. AFS - *Andrew File System* e CODA *File System*

Tanto o *Andrew File System*, também conhecido por AFS, como o CODA são sistemas de arquivos distribuídos que permitem aos *hosts* (clientes e servidores) cooperarem para compartilhar de maneira eficiente os recursos de seus sistemas de arquivos através de uma rede local ou de longa distância, estando a tolerância a falhas presente no CODA mas não no AFS [FAQ 2003] [NCSA 2003] [AFS 2003] [IBM 2003].

O CODA é um sistema de arquivos resistente a falhas que geralmente ocorrem em um ambiente com estações de usuários, ou seja, falhas como não encontrar o servidor de arquivos ou então falhas no subsistema de comunicação. No caso, ele contorna estes problemas utilizando-se de dois mecanismos, a replicação de servidores (replicação de primeira classe) e a operação desconectada (ver figura 2.8) [SATYNARAYANAN 1989] [SATYNARAYANAN 1989] [CODA 2003] [BRAAM 1999].

O sistema de arquivos CODA utiliza muitas das características encontradas também no *Andrew File System*; na verdade o CODA é derivado do AFS versão 2 [CODA 2003]. Dentre as afinidades podem ser destacadas a distinção entre servidores e clientes, o tratamento empregado para garantir a segurança, a forma de comunicação utilizando-se de um mecanismo de chamada remota de procedimento, a autenticação e a utilização de um único *namespace*.

Os arquivos disponibilizados pelo AFS e pelo CODA são armazenados em diretórios localizados na raiz da estrutura de diretórios do sistema de arquivos do cliente com os nomes de */afs* ou */coda*, respectivamente. Sob os mesmos, os

clientes visualizam os diretórios dos diversos servidores que estão compartilhando seus sistemas de arquivos locais (figura 2.12).

Estes servidores podem estar em uma rede local ou em uma rede geograficamente dispersa e a utilização pode ser feita através de comandos tradicionais de manipulação de arquivos e diretórios, respeitando os direitos de acesso disponibilizados pelo sistema operacional local.

Figura 2.12: Servidores do *Andrew File System* (AFS) e do *CODA File System* ligados à estrutura de diretórios do cliente.

Dentre as funcionalidades proporcionadas pelo *Andrew File System* e pelo CODA destacam-se o gerenciador de cache, a independência de localização, a imagem única do sistema e a capacidade de gerenciamento do sistema [FAQ 2003].

Ambos possuem um *Gerenciador de Cache*, ou seja, as máquinas clientes sob estes sistemas de arquivos executam um processo denominado Gerente de Cache (*Cache Manager*).

Este processo mantém informações sobre as identidades dos usuários que estiverem conectados à máquina, buscando e encontrando dados, manipulando arquivos e diretórios. O efeito disto é que, quando um arquivo remoto é acessado

através do cache, uma parte dele é copiada para o cache localizado no disco local (processo *cold read*) e então acessada (processo denominado *warm read*). Este acesso é tão rápido como se fosse um acesso ao disco local e consideravelmente mais rápido que uma leitura realizada através da rede.

Tanto o CODA como o AFS operam com uma *independência de localização*, ou seja, mapeiam os sistemas de arquivos remotos em um diretório local, no caso do AFS este diretório é o /afs e no caso do CODA seria o /coda.

A independência da localização faz com que um usuário não necessite saber qual servidor possui o arquivo, mas apenas o caminho para o mesmo (facilidade de endereçamento). A independência da localização implementada pelos dois sistemas de arquivos facilita a adição de novos servidores, atendendo também a escalabilidade.

Sobre a *Segurança*, ambos os sistemas de arquivos utilizam os mecanismos de segurança disponibilizados pelos próprios sistemas operacionais locais, como o *Kerberos*, para identificar e autenticar os usuários.

Utilizam também o conceito de autenticação mútua, ou seja, tanto o fornecedor do serviço (no caso o servidor) como o requisitante do serviço (no caso o cliente) comprovam suas identidades. Também podem utilizar listas de controle de acesso (ACL - *Access Control List*) para habilitar usuários a acessos restritos, como seu próprio diretório de trabalho (*home directory*).

A *Imagem Única do Sistema (SSI - Single System Image)* também é uma característica comum aos dois sistemas de arquivos. Ela viabiliza a mesma visão dos arquivos para cada cliente e servidor da rede e pode ser considerada uma maneira de simplificar o acesso. Isto é importante pois permite ao usuário mover-se de uma estação-cliente para outra e ainda possuir a mesma visão do sistema de arquivos.

E quanto à *Capacidade de Gerenciamento do Sistema*, os administradores de sistemas podem realizar mudanças nas configurações sem que exista a necessidade de "logar" nos servidores dos arquivos e pode ser feito a partir de qualquer estação cliente.

2.5.10.3. NFS - Network File System

O Sistema de Arquivos de Rede (*Network File System* – NFS) da *Sun Microsystems*, é considerado o padrão *de fato* para o compartilhamento de arquivos pela rede e é suportado nativamente na maioria das versões do UNIX e Linux, estando disponível para praticamente qualquer ambiente operacional e opera baseado no modelo cliente–servidor [DEITEL 2005].

O projeto original do NFS tinha o TCP/IP como base e assim pôde ser facilmente adaptado para conectar sistemas de arquivos geograficamente dispersos através da *Internet* e outras redes de longa distância.

A versão original do NFS foi liberada pela SUN em 1984 e foi licenciada gratuitamente para a indústria de computadores. Em 1985, foi revisado para a versão 2 e integrado à vários sistemas operacionais, em 1989 o NFS-2 tornou-se a primeira versão de NFS a ser padronizada pelo *Internet Engineering Task Force* (IETF). Uma nova revisão foi feita em 1995 com a liberação da versão 3.

Tanto o NFS-2 como o NFS-3 admitem uma implementação de servidor sem estado (*stateless*), o que torna mais fácil implementar a tolerância a falhas. Caso um servidor de arquivos falhar, o cliente pode simplesmente tentar novamente até o servidor responder.

As verificações de permissões utilizadas nas versões 2 e 3 confiam apenas na informação de autenticação da chamada remota de procedimento a qual determina se o cliente pode ou não acessar certas informações e funções.

Quanto ao protocolo de transporte empregado, ambos podem utilizar tanto o UDP como o TCP, porém o protocolo UDP é indicado para redes locais e para redes onde a taxa de erros e a latência são baixas enquanto o TCP é indicado para redes propensas a erros, alta latência e de longa distância.

Uma questão importante a ressaltar é que nenhuma das versões define uma política de cache para garantir a consistência do servidor e das cópias do lado dos clientes do arquivo.

2.5.11. *Heartbeat*

Na replicação de primeira classe os servidores são replicados para fornecer a alta disponibilidade do *hardware* e dos serviços, dessa forma torna-se necessário um mecanismo que permita conhecer o estado dos equipamentos e dos serviços executados nas mesmas.

Para tanto, utiliza-se frequentemente uma técnica de enviar sinais constantemente para os equipamentos e serviços que se deseja monitorar; no caso de uma resposta positiva e dentro de certos limites de tempo será interpretado que eles estão operacionais e disponíveis (figura 2.13); caso contrário, será interpretado que o componente ou serviço estão indisponíveis. A este mecanismo de verificação dá-se o nome de *heartbeat* [ROBERTSON 2003].

Figura 2.13: *Heartbeat* com envio e recebimento.

Uma variação de implementação desta técnica é fazer com que os elementos

monitorados enviem, de tempos em tempos, sinais para um componente central, sendo este um equipamento ou mesmo um serviço (figura 2.14).

A interrupção no envio deste sinal será interpretada como uma inoperância dos mesmos. Esta variação mostra-se mais interessante que a anterior, visto que irá gerar uma quantidade de tráfego de rede menor, pois será feita uma única transmissão, enquanto que na primeira alternativa (figura 2.13), existe um envio e um retorno de sinal [ROBERTSON 2003].

Figura 2.14: *Heartbeat* somente com retorno.

2.6. Implementações da Técnica *Checkpointing/Restart*

Conforme mencionado, *Checkpoint/Restart* consiste em uma técnica que cria um arquivo onde é armazenado o estado do processo em execução, sendo possível reconstruir tal processo através do conteúdo deste mesmo arquivo. Ela pode ser aplicada para realizar *checkpointing* em processos seqüenciais ou mesmo paralelos, atuando em ambientes de *cluster*, de migração de processos ou ainda como *backups* periódicos de processos [DUELL 2004].

O arquivo gerado durante a fase de *checkpoint* contém as pilhas, *heap* e registradores do processo, além do estado dos sinais pendentes, manipuladores de sinais, registros de uso e estado do terminal.

Durante a fase de *restart*, o sistema operacional recria o processo e quaisquer outros objetos descritos no arquivo de *checkpoint*, isto permite ao processo continuar sua execução do ponto onde o último *checkpoint* foi tomado.

São quatro os principais pontos que devem ser considerados para o processo de *checkpoint/restart*, a saber, a transparência, o paralelismo das aplicações, o desempenho e a portabilidade.

A transparência requer que as aplicações do usuário devam ter suas informações registradas e reiniciadas sem que sejam feitas modificações profundas no código-fonte.

A utilização da tomada de *checkpoints*, com o efetivo registro nos arquivos, não deve minimizar o impacto sobre o desempenho final da aplicação paralela ou seqüencial e também deve possuir funcionalidades que possam ser facilmente estendidas para outros sistemas operacionais.

2.6.1. Berkeley Labs Linux Checkpoint/Restart - BLCR

O projeto *Berkeley Labs Linux Checkpoint/Restart* (BLCR) [BLCR 2004] é uma implementação da técnica de *checkpoint/restart* empregada quando se trabalha com aplicações paralelas usando-se o LAM/MPI [LAM/MPI 2005] ou então com aplicações seqüenciais.

Ele pode ser utilizado em sistemas isolados para aplicações processadas em um único nó ou então em ambientes interconectados utilizados por aplicações paralelas processadas em múltiplos nós.

A técnica de *checkpoint/restart* é bastante popular em ambientes que executam aplicações paralelas e distribuídas. Nestes ambientes os usuários ou administradores podem tomar *checkpoints* de seus processos para preservar

execuções já realizadas, prevenindo-se contra possíveis falhas nos nós e, conseqüentemente, a perda do processamento já realizado.

A implementação do BLCR pode ser utilizada como módulo do *kernel* ou como uma biblioteca do usuário. A utilização como módulo do *kernel* tem como benefícios fazer com que suas funcionalidades possam ser utilizadas pelos usuários sem que mudanças sejam feitas nos códigos-fonte das aplicações e a implementação disponível para a camada do usuário permite a liberdade de decisão para os desenvolvedores de quando e onde os *checkpoints* devem ser tomados.

O BLCR utiliza um procedimento manual para se tomar os *checkpoints* das aplicações, sejam elas seqüenciais ou paralelas, ou seja, o usuário deve executar um comando externo ou embutir uma chamada em seu código-fonte para que o *checkpoint* da aplicação seja tomado ou então seja recuperado.

Antes que os *checkpoints* sejam tomados, são necessários, inicialmente, carregar dois módulos de *kernel*, o *cr* e o *vmadump* [BLCR 2005].

Para que uma aplicação seqüencial tenha o seu *checkpoint* tomado com sucesso a mesma deve conter os códigos providos pelo BLCR e isso pode ser feito através da maneira como se solicita a execução da aplicação, ou seja, através de um comando especial que, no caso seria:

\$ cr_run aplicação [argumentos_da_aplicação]

O *cr_run* é o responsável por carregar a biblioteca juntamente com a aplicação seqüencial e através desta modalidade não existe a necessidade de alterar-se o código-fonte da aplicação.

Conforme mencionado, o BLCR utiliza uma forma manual para se tomar o *checkpoint* da aplicação sendo o comando responsável para essa ação demonstrado a seguir:

\$ cr_checkpoint *pid_da_aplicação*

O comando *cr_checkpoint* registra o contexto do processo cujo PID coincidir com o *pid* fornecido e tal contexto do processo é armazenado em um arquivo que tem como nome *context.pid*.

Para recuperar, ou reinicializar, um arquivo de contexto, o BLCR utiliza também um comando inicializado manualmente pelo usuário denominado *cr_restart*, sendo sua sintaxe a apresentada logo abaixo:

\$ cr_restart *context.pid*

O comando *cr_restart* lê o conteúdo do arquivo *context.pid* recriando o processo e reativando sua execução a partir do último estado armazenado no referido arquivo.

Caso se deseje trabalhar com aplicações paralelas empregando-se o BLCR e o LAM/MPI, deve-se inicializar a aplicação utilizando-se a maneira tradicional do MPI, ou seja:

\$ mpirun [*opções_do_mpirun*] aplicação [*opções_da_aplicação*]

E, para se tomar o *checkpoint* da aplicação paralela deve-se digitar o comando:

\$ cr_checkpoint *pid_do_mpirun*

Uma observação em relação à tomada de *checkpoints* para aplicações paralelas é que deve ser informado o *pid* do *mpirun* e não o da aplicação paralela propriamente dita.

O comando anterior produz um arquivo de contexto para o *mpirun* e outro

para cada um dos processos paralelos que compõem a aplicação. Assim cada processo paralelo armazenar seu estado local em seu respectivo arquivo de contexto.

Para se reinicializar a aplicação paralela utiliza-se também o comando *cr_restart*, porém referenciando o arquivo de contexto gerado para o *mpirun*. A sintaxe assume a seguinte forma:

\$ cr_restart context.pid_do_mpirun

Assim, todos os processos do *job mpirun* voltam a ser recriados e reiniciados a partir do último estado local armazenado em seus arquivos de contexto.

O LAM/MPI utiliza a abordagem coordenada bloqueante para trabalhar com o *checkpoint* de seus *jobs*, sendo que a atual implementação do LAM trabalha com um subsistema de comunicação baseado no TCP.

A tomada de *checkpoint* de um *job* MPI é iniciada por um usuário ou por um escalonador que envia uma requisição de *checkpoint* para o MPIRUN e o mesmo a propaga para todos os processos envolvidos naquele *job*.

Após receberem esta requisição de *checkpoint*, todos os processos MPI interagem uns com os outros para garantir que seus *checkpoints* locais irão refletir um *checkpoint* global consistente.

Esta abordagem adotada no LAM certifica que todos os canais de comunicação existentes entre os processos sejam esvaziados antes que um *checkpoint* seja tomado.

Durante o procedimento de reinício, todos os processos retomam suas execuções a partir dos estados armazenados, com os canais de comunicação restaurados a partir de estados conhecidos, vazios no caso. O algoritmo utilizado

para esvaziar os canais de comunicação pode ser encontrado em [SANKARAN 2004].

2.6.2. Outras implementações da técnica *Checkpointing/Restart*

Outras implementações da técnica de *checkpoint/restart* podem ser encontradas na literatura [ROMAN 2005] e algumas são apresentadas a seguir.

A *libckpt* [PLANK 1995] é uma das primeiras bibliotecas implementadas para o UNIX. Ela possui algumas otimizações para reduzir o tamanho dos arquivos de *checkpoint*, sendo que uma delas consiste em não armazenar páginas de memórias que não foram usadas ou que não sofreram modificações desde o último *checkpoint*. Ela implementa as técnicas de *checkpoint Incremental*, *Forked* e Síncrono e necessita que o código-fonte seja alterado.

No *Condor* [LITZKOW 2004], a implementação viabiliza a migração de processos para o sistema de balanceamento de carga. Ela necessita que a aplicação seja ligada a uma biblioteca e não requer muitas alterações no código-fonte.

A implementação *libckp* [WANG 1995] é uma biblioteca desenvolvida para tolerância a falhas e armazena também o conteúdo dos arquivos abertos como parte do estado do processo.

A *libtckpt* [DIETER 2004] toma *checkpoints* de aplicações *multithreads* utilizando o *Linux Threads* ou *Solaris Threads*. Esta biblioteca requer modificações no código-fonte e a aplicação deve incluir um arquivo de *header*.

A *Score* [TAKAHASHI 2004] é uma biblioteca que implementa *checkpoints* em aplicações paralelas sem necessitar que mudanças sejam feitas no código-fonte.

A implementação *CoCheck* [PRUYNE 2004] implementa *checkpoint/restart* para aplicações paralelas sob o sistema *Condor*. Nesta implementação um processo gerenciador de recurso é usado para auxiliar a sincronizar o estado da rede e facilitar o reinício da aplicação paralela e para evitar gargalos no servidor de arquivos e na rede, utiliza-se um servidor de armazenamento exclusivo para os arquivos *checkpoints*.

2.6.3. A abordagem utilizada para o trabalho

A abordagem apresentada neste trabalho pode interagir com arquiteturas fisicamente ou logicamente paralelas, ou seja, pode existir replicação de elementos de computação para que a execução da aplicação paralela seja acelerada ou então esta replicação pode ser simulada dentro de um ambiente monoprocessado.

Não é adotada neste trabalho a abordagem de replicar a mesma execução em vários elementos (normalmente encontrada em sistemas tolerantes a falhas), mas sim a de distribuir as tarefas individualmente para elementos de computação distintos, conforme defende a abordagem distribuída, buscando com isso uma descentralização dos controles e um melhor aproveitamento do poder computacional de cada nó de processamento.

Para interligar estes elementos de computação fisicamente replicados é utilizada uma estrutura frouxamente acoplada. Esta escolha foi feita em função do uso dos equipamentos responsáveis pela computação, pelo armazenamento e pela comunicação serem facilmente encontrados no mercado e também estas serem as características básicas de um *cluster* de computadores.

Com respeito às classes de tolerância a falhas, o trabalho aborda apenas a classe *fail stop*, ou seja, a interrupção do funcionamento de todo o elemento de computação e não apenas CPU ou memória. O que motivou a escolha desta classe de falhas é o fato de ser de fácil detecção e ser a que ocorre frequentemente.

A detecção desta classe de falhas é facilmente percebida pelos parceiros que compõem o ambiente distribuído, ou seja, quando um dos participantes sofre uma falha desta classe, imediatamente ele interrompe suas atividades, inclusive aquelas que anunciam para os demais participantes do ambiente computacional sobre sua operacionalidade. Com relação à frequência, as falhas desta classe são as que costumam aparecer nos sistemas de computação de uso geral.

Quanto às questões de interrupção dos serviços baseados em *software*, é empregada a técnica de *checkpointing* para registros e recuperações futuras de aplicações minimizando as perdas de tempo e da computação já realizada.

A utilização de *checkpointing* é viabilizada através do projeto BLCR e também disponibilizada por uma biblioteca, uma vez que deseja-se que o desenvolvedor tenha total liberdade de escolher onde e quando os *checkpoints* devem ser tomados ou não.

Na verdade, o BLCR é utilizado apenas como elemento tomador de *checkpoints* dos processos, agregando-o a um ambiente que utilize uma abordagem automática para estes registros, liberando o usuário da tarefa tomá-los manualmente. Uma outra razão importante para a escolha do BLCR é a de que o mesmo está em conformidade com o LAM/MPI e também trabalha com os registros de contextos de processos paralelos.

Com respeito às aplicações paralelas, as mesmas utilizam a estrutura provida pelo LAM/MPI para viabilizar a execução paralela e a troca de mensagens, portanto, o resultado do trabalho deve ser totalmente aplicável a estas aplicações e formas de funcionamento.

3. AMBIENTE PARA MINIMIZAÇÃO DO IMPACTO DE FALHAS PARA APLICAÇÕES PARALELAS

Uma vez que a execução das aplicações paralelas requer mais recursos dos sistemas de computação em razão dos problemas computacionais que se propõe a resolver e, em caso de falhas em algum dos componentes do sistema distribuído ou da própria aplicação, tem-se a necessidade de reiniciar todo o processamento do ponto de partida, desperdiçando-se todo o tempo e processamento já realizados.

Desta forma, torna-se interessante aplicar técnicas e utilizar mecanismos de tolerância a falhas que procurem minimizar estes desperdícios. Estas técnicas e mecanismos podem ser viabilizados tanto por *hardware* como por *software*.

No caso do *hardware*, os computadores paralelos, quando projetados, são preparados para utilizar a redundância de componentes de *hardware*, procurando evitar a interrupção do processamento ou indisponibilidade de um serviço. A utilização de um *hardware* de computador especialmente projetado para atender esta necessidade pode fazer com que o valor final do equipamento seja elevado.

Já no caso do *software*, algumas técnicas têm sido discutidas e podem ser empregadas para detectar o aparecimento das falhas, tratá-las no nível da informação e manter a continuidade da execução.

A idéia presente no controle por *software* está em utilizar como *hardware* de computador componentes facilmente encontrados no mercado, diminuindo o valor final do ambiente computacional, porém mantendo um grau de confiabilidade próximo ao dos computadores de alto custo.

O ambiente de execução e monitoramento, aqui denominado *AMTF* (**A**mbiente de **M**onitoramento **T**olerante a **F**alhas), utiliza técnicas de tolerância a falhas por *software* e disponibiliza um mecanismo para se detectar falhas da classe *fail stop* em processos paralelos que compõem aplicações paralelas executadas em *cluster* de computadores e permite que estas mesmas aplicações paralelas possam ser recuperadas e reiniciadas, praticamente, a partir do ponto onde foram interrompidas, preservando assim todo o processamento realizado até o último estado armazenado.

Para tanto, a detecção das falhas nos processos paralelos é feita mediante a utilização da técnica de monitoramento de processos por *heartbeat* e para permitir a continuidade da execução dos processos, sem perda do processamento já realizado, é empregada a técnica de tomada e recuperação de *checkpoints*, disponibilizada pelo projeto BLCR sobre o LAM/MPI.

Desta forma, são apresentados como objetivos deste trabalho a disponibilização de um ambiente de execução e monitoramento de aplicações paralelas que utilize mecanismos para a detecção de falhas da classe *fail stop* em processos paralelos, a complementação do mecanismo utilizado pelo projeto BLCR para tomada de *checkpoints* e restauração da execução dos mesmos tornando esses procedimentos automáticos e a proposição de uma biblioteca que permita ao desenvolvedor de aplicações paralelas adicionar ao código-fonte instruções que forcem a tomada de *checkpoint* em determinados pontos da aplicação onde julgue conveniente.

Para atingir estes objetivos e obter a tolerância a falhas, além da continuidade de funcionamento dos processos paralelos com a minimização da perda do tempo de processamento realizado, foi desenvolvido um ambiente de execução e monitoramento, aqui chamado de AMTF. O ambiente em questão é composto por um conjunto de módulos de *software* e possui a estrutura apresentada na figura 3.1 e detalhada nas seções seguintes.

3.1. Estrutura do Ambiente AMTF

Na figura 3.1, vê-se a divisão do ambiente AMTF em quatro grandes blocos, sendo que cada um deles tem uma função específica e encontra-se apresentada, resumidamente, a seguir.

Figura 3.1: Estrutura do Ambiente AMTF.

O bloco 1 consiste do módulo **Executor** e sua função dentro do ambiente é relativamente simples, ou seja, ele é o responsável por ativar o procedimento de inicialização da aplicação paralela (bloco 2) solicitada pelo usuário, bem como a execução do *Monitor do Grupo de Processos* (bloco 3) para a respectiva aplicação paralela.

O bloco 2 consiste da **Aplicação Paralela** propriamente dita e é composto pelo processo MPIRUN e pelos processos paralelos que constituem a aplicação paralela. É importante lembrar que o LAM/MPI necessita do MPIRUN para inicializar uma aplicação paralela e o módulo BLCR utiliza-o como instrumento para enviar os pedidos de tomada de *checkpoints* locais aos processos paralelos, bem como para a tomada do *checkpoint* global da aplicação paralela.

O bloco 3 consiste do **Monitor do Grupo de Processos** e suas funções são as de receber as sinalizações provenientes dos processos paralelos que compõem a aplicação paralela (como por exemplo o *heartbeat*), solicitar que os *checkpoints* sejam tomados periodicamente, enviar sinalização ao *Monitor Central* (bloco 4) e em caso de falha de algum dos processos paralelos, executar o procedimento de *failover* e *failback* da aplicação paralela. Cada aplicação paralela terá um monitor de grupo responsável por monitorar os processos paralelos do seu grupo em particular.

O bloco 4 consiste dos módulos **Monitor Central**, **Monitor de Réplica**, **Controlador AMTF** e **Sistema de Arquivos Tolerante a Falhas** e tem por função a centralização do recebimento das sinalizações provenientes de todos os *Monitores dos Grupos de Processos* executados no nó específico, a execução de procedimentos de *failover* e *failback* para estes mesmos monitores quando necessário, o auto-monitoramento, a inicialização e encerramento do ambiente de execução, a reativação das aplicações interrompidas em decorrência de uma operação de *shutdown* ou queda do *cluster*, além da disponibilização de acesso ao sistema de arquivos para todos os arquivos produzidos pela tomada de *checkpoints* e pelos arquivos de *log* mantidos pelos monitores.

O módulo que compõe o bloco 1 está sob o controle do operador, ou seja, o usuário será responsável por ativá-lo diretamente através da linha de comando; já os módulos dos blocos 2 e 3 são acionados automaticamente pelo módulo do bloco 1 e os módulos do bloco 4 estão sob o controle do sistema operacional, sendo ativados durante o carregamento do mesmo, preferencialmente na forma de serviços.

3.2. Descrição dos Módulos

Uma representação de todos os módulos que compõem o ambiente AMTF, bem como suas respectivas interações, formas de comunicação e arquivos de *log*,

pode ser vista na figura 3.2, sendo que as seções seguintes apresentam um detalhamento sobre a composição e as funcionalidades dos mesmos.

Figura 3.2: Interação e Comunicação entre os módulos do Ambiente AMTF.

3.2.1. O Controlador AMTF

O módulo *Controlador AMTF* é o componente que tem por função inicializar ou encerrar o ambiente AMTF e deve ser acionado logo após o sistema operacional ser carregado, preferencialmente na forma de um serviço do sistema operacional, fazendo com que o ambiente se torne ativo assim que o sistema estiver disponível.

Durante o procedimento de carga do sistema operacional, cada nó que compõe o *cluster* de computadores solicita a execução deste serviço localmente e assim é ativado um módulo *Monitor Central* em cada um deles. A junção de todos esses *Monitores Centrais* locais resultam na constituição de um monitor distribuído

chamado de *Monitor Central Virtual*.

Caso seja necessário executar uma operação de *shutdown* no *cluster* de computadores deve-se, previamente, realizar um procedimento de encerramento do ambiente de execução e monitoramento; assim o módulo *Controlador AMTF* pode ser acionado automaticamente durante o procedimento de encerramento do sistema operacional ou então acionado pelo operador a partir de qualquer nó do *cluster*, através da linha de comando.

Quando este procedimento de encerramento é acionado, o módulo *Controlador AMTF* envia uma mensagem de encerramento através de um *socket* para todas as réplicas dos *Monitores Centrais* que compõem o *Monitor Central Virtual* (figura 3.3) e estas executam os devidos procedimentos internos de encerramento.

Figura 3.3: Interação e Comunicação entre o *Controlador AMTF* e *Monitor Central*.

O módulo *Controlador AMTF* tem conhecimento de onde as réplicas dos *Monitores Centrais* estão executando e em qual porto eles esperam pelas mensagens graças a um arquivo (REPLICAS) onde ficam registrados os endereços dos nós participantes do ambiente AMTF e seus respectivos portos.

É importante ressaltar que os mecanismos e os canais de comunicação empregados aqui para difundir o pedido de encerramento do módulo *Controlador AMTF* para os módulos *Monitores Centrais* são utilizados também pelos mesmos

para enviar e receber *heartbeats* uns para os outros.

Um detalhamento das ações tomadas pelo *Controlador AMTF* está presente na seção 3.5, onde são apresentados os ciclos de vida de todos os módulos do ambiente AMTF em diversas situações.

3.2.2. O Monitor Central

O *Monitor Central* é o módulo de *software* acionado pelo módulo *Controlador AMTF*, sendo que sua função básica é a de monitorar o funcionamento dos *Monitores do Grupo de Processos* que estão em execução em seu nó do *cluster* de computadores e, conseqüentemente, executar os procedimentos de *failover* e *failback* dos mesmos quando for necessário.

Como função secundária, o *Monitor Central* deve monitorar o módulo de *software Monitor de Réplica*, as demais cópias dos *Monitores Centrais* e eventualmente executar os procedimentos de *failover* e *failback* das mesmas, além de reativar aplicações interrompidas em decorrência do procedimento de *shutdown* ou queda que o *cluster* de computadores venha a sofrer.

As interações e as comunicações que o módulo *Monitor Central* implementa são descritas a seguir e estão representadas na figura 3.4.

Este módulo conta com a manipulação de seis arquivos de apoio, sendo um primeiro arquivo (REPLICAS) onde ficam armazenados os endereços dos nós nos quais as réplicas dos *Monitores Centrais* estão sendo executadas, um segundo arquivo (AMTF.CNF) onde se recupera a periodicidade com que os *checkpoints* automáticos deverão ser tomados, um terceiro arquivo (MONITORES DE GRUPO) onde ficam registradas informações sobre os *Monitores do Grupo de Processos* que devem ser monitorados, um quarto arquivo (LOG), onde ficam registradas todas as ações percebidas e tomadas pelo *Monitor Central*, um quinto arquivo (SHMIDs) onde

são registrados os endereços das áreas de memória compartilhada que serão utilizadas para a comunicação com o *Monitor de Réplica* e o sexto arquivo (APLICAÇÕES INTERROMPIDAS) onde serão registradas as informações relativas às aplicações em execução e interrompidas em razão de um procedimento de *shutdown* do *cluster* de computadores.

Figura 3.4: Interações e Comunicações do módulo *Monitor Central*.

O módulo em questão também possui capacidade de comunicação, utilizando-se dos mecanismos *socket* e memória compartilhada para enviar e receber mensagens para e dos módulos *Controlador AMTF*, *Monitores Centrais* e *Monitor de Réplica*.

Apesar de existirem cópias dos *Monitores Centrais* em cada um dos nós que compõem o *cluster*, eles atuam como se existisse apenas um único *Monitor Central*, virtual e distribuído, ou seja, existe um auto-monitoramento entre estas cópias para reiniciá-las em caso de interrupção das mesmas.

Quando acionado, O *Monitor Central* recupera os endereços das demais réplicas dos *Monitores Centrais* e abre um canal de comunicação, via *socket*, com as

mesmas a fim de iniciar o envio e recebimento de sinais (*heartbeats*) para as referidas cópias.

Deve também, ao ser inicializado, ativar o seu *Monitor de Réplica*, realizando o seu monitoramento e ficar aguardando pelo aparecimento de algum *Monitor do Grupo de Processos*, observando o arquivo onde são registrados tais monitores.

Ao ser colocado em execução algum *Monitor do Grupo de Processos*, o mesmo inicia a troca de mensagens (*heartbeats*), via memória compartilhada como o *Monitor Central*. Caso esta troca de mensagens seja interrompida em algum momento, é interpretada como uma falha operacional do respectivo *Monitor de Grupo de Processos* (procedimento de *failover*). Em resposta a esta interrupção, o *Monitor Central* inicia uma nova cópia do referido *Monitor do Grupo de Processos* para que a continuidade de sua execução seja possível (procedimento de *failback*).

De maneira análoga, quando o *Monitor de Réplica* é inicializado ele também inicia uma troca de mensagens (*heartbeat*), via memória compartilhada, com o *Monitor Central* e este, ao perceber a interrupção desta comunicação (procedimento de *failover*) inicia uma nova cópia do *Monitor de Réplica* (procedimento de *failback*).

Eventualmente, o *Monitor Central* pode receber uma mensagem, via *socket*, vinda do módulo *Controlador AMTF* solicitando seu encerramento; quando isso ocorre cabe ao *Monitor Central* solicitar para todos os *Monitores do Grupo de Processo* sob seu monitoramento que providenciem a tomada de *checkpoints* das aplicações paralelas que estão em execução para que as mesmas também sejam encerradas.

Por razões que possam interessar ao usuário ou ao administrador dos recursos computacionais providos pelo ambiente computacional, pode haver a necessidade do *cluster* executar um procedimento de *shutdown* em algum momento e, neste caso, as aplicações que estiverem em execução necessitam ter seus estados armazenados e suas execuções interrompidas. Quando o *cluster* voltar a

funcionar, as mesmas devem ter suas execuções retomadas a partir do último ponto armazenado nos arquivos de contextos.

Cabe ao *Monitor Central* a função de reativar as *Aplicações Paralelas* suspensas em razão de um procedimento de *shutdown* do *cluster* e seu respectivo retorno ao funcionamento.

Para realizar estas reativações, este módulo recupera informações de um arquivo de apoio (APLICACOES INTERROMPIDAS) criado durante o processo de encerramento do ambiente de execução e, todas as aplicações ali registradas são recolocadas em funcionamento.

Até este momento o *Monitor Central* foi apresentado como sendo um processo que executa localmente em cada nó de processamento e monitorado, também localmente, por um *Monitor de Réplica*.

Porém amplia-se esse conceito de *Monitor Central* estendendo-o para uma abordagem de monitoramento distribuída criando-se um *Monitor Central Virtual*, que consiste na junção de todos os *Monitores Centrais* locais, comunicando-se uns com os outros e realizando um auto-monitoramento remoto.

O objetivo de se criar esse *Monitor Central Virtual* reside no fato de que as falhas podem afetar não apenas o *Monitor Central* local, mas também o *Monitor de Réplica* e, neste caso, estando ambos sem execução não há outra maneira de reativá-los. Com este novo monitor a interrupção pode ser percebida pelas outras cópias dos *Monitores Centrais* que estão sendo processados em outros nós e, algum deles encarrega-se de reativar os processos locais interrompidos.

Este *Monitor Central Virtual* não é um módulo do ambiente AMTF, mas sim uma funcionalidade que pode ser extraída do trabalho em conjunto de todos os

Monitores Centrais locais.

O procedimento de reativação é de responsabilidade de um dos *Monitores Centrais* que compõem o *Monitor Central Virtual*, este encarrega-se da reativação do parceiro falho, ou seja, cria-se uma rotina de reativação porém não concentrando essa tarefa em um único monitor.

Uma alternativa viável é a de atribuir um identificador para cada *Monitor Central* local (ranqueamento) e utilizá-lo para definir quem será o responsável por reativar os outros monitores.

A implementação realizada no ambiente AMTF é a do responsável pela reativação do *Monitor Central* no nó cujo ranqueamento seja 1 ser o *Monitor Central* de ranqueamento 0, o de ranqueamento 2 ser o de ranqueamento 1 e assim sucessivamente, até atingir o último ranqueamento que seria responsável por reativar o *Monitor Central* de ranqueamento 0.

O *Monitor Central* encarregado de reativar o parceiro falho envia um comando remoto ao nó em questão e o sistema operacional do mesmo ativa o processo correspondente ao *Monitor Central* local e este, ao ser iniciado, ativa também o seu *Monitor de Réplica*.

Assim, como mencionado na seção anterior, um detalhamento das ações tomadas pelo *Monitor Central* local e pelo *Monitor Central Virtual* é apresentado na seção 3.5, através dos ciclos de vida de todos os módulos do ambiente AMTF em diversas situações.

Conforme mencionado no início desta seção, o *Monitor Central* também registra todo o seu comportamento em um arquivo de *log*, permitindo assim consultas futuras sobre seus procedimentos executados.

3.2.3. O Monitor de Réplica

O *Monitor de Réplica* é um processo inicializado pelo *Monitor Central* e sua função básica reside em monitorar o funcionamento (procedimento de *failover*) do módulo *Monitor Central* e eventualmente realizar o procedimento de *failback* do mesmo.

As interações e comunicações implementadas por este módulo são descritas a seguir e representadas na figura 3.5.

Figura 3.5: Interações e Comunicações do módulo *Monitor de Réplica*.

Este monitor utiliza dois arquivos de apoio, o primeiro arquivo (**SHMIDs**) registra os endereços das posições de memória compartilhada a serem utilizadas para a comunicação com o *Monitor Central* e o segundo arquivo (**LOG**) será utilizado como um arquivo de *log* do referido monitor.

O *Monitor Central* envia periodicamente sinais (*heartbeats*) ao *Monitor de Réplica*, via memória compartilhada, para indicar o seu funcionamento e na ausência do recebimento destes sinais, o *Monitor de Réplica* inicializa uma nova cópia do *Monitor Central*, visto que interpreta que o módulo original apresentou uma falha.

O *Monitor de Réplica* também registra todas as sinalizações emitidas ou

recebidas, bem como ações tomadas durante sua execução em seu arquivo de *log*, visando consultas futuras sobre o seu funcionamento.

Da mesma forma que o módulo *Monitor de Réplica* atua como um monitor do módulo *Monitor Central*, este último atua como um monitor do primeiro, viabilizando um auto-monitoramento.

Um detalhamento das ações desempenhadas por este módulo pode ser visto na seção 3.5, onde são apresentados os ciclos de vida de todos os módulos que compõem o ambiente AMTF.

3.2.4. O Executor

O módulo *Executor* é um módulo de *software* que deve ser acionado pelo usuário e sua função básica é a de colocar em execução a *Aplicação Paralela* e ativar o *Monitor do Grupo de Processos* para a referida aplicação. A figura 3.6 apresenta suas interações.

Figura 3.6: Interações do módulo *Executor*.

Ele possui interação com um único arquivo de apoio (LOG) onde estão

registradas todas as referências à execução de aplicações solicitadas pelo usuário.

Originalmente, o BLCR, atuando junto ao LAM/MPI, necessita que os pedidos de tomada de *checkpoints* sejam encaminhados para o MPIRUN e este os replique para seus processos paralelos para que os estados locais dos mesmos sejam armazenados em seus respectivos arquivos de contexto, porém, para enviar esta requisição, o BLCR necessita conhecer o número atribuído ao PID do MPIRUN.

Desta forma, o operador, ao invés de solicitar a execução da aplicação paralela diretamente através do MPIRUN como é de costume no LAM/MPI, executa o módulo *Executor* e este por sua vez ativa a *Aplicação Paralela* e recebe como retorno da mesma o PID atribuído ao processo MPIRUN. Em seguida coloca em funcionamento o *Monitor do Grupo de Processos* para a aplicação paralela iniciada e informa a ele o PID do MPIRUN recebido (figura 3.6).

3.2.5. A Aplicação Paralela

A Aplicação Paralela consiste em uma aplicação desenvolvida utilizando-se o LAM/MPI, além de uma biblioteca disponibilizada ao desenvolvedor da aplicação paralela por este ambiente.

Há a necessidade do desenvolvedor adicionar algumas chamadas ao seu código-fonte para que o mesmo possa emitir as sinalizações necessárias que, por sua vez, são monitoradas pelo *Monitor do Grupo de Processos* criado para a respectiva aplicação paralela.

Estas chamadas abrem e fecham um canal de comunicação (*socket*) com *Monitor do Grupo de Processos* e podem, futuramente, ser embutidas no MPI_INIT() e no MPI_FINALIZE(), reduzindo ainda mais a intervenção do desenvolvedor junto ao código-fonte da aplicação.

3.2.6. O Monitor do Grupo de Processos

O módulo *Monitor do Grupo de Processos* é inicializado tanto pelo módulo *Executor*, quando uma nova aplicação paralela for inicializada, como pelo módulo *Monitor Central*, caso o *cluster* de computadores esteja sendo reativado após uma operação de *shutdown*.

Sua função básica resume-se a monitorar os processos paralelos que compõem a aplicação paralela, executando procedimentos de *failover* e *failback* para os mesmos.

A figura 3.7 representa as interações e comunicações que o módulo *Monitor do Grupo de Processos* desempenha, sendo as mesmas comentadas a seguir.

Figura 3.7: Interações e Comunicações do módulo *Monitor do Grupo de Processos*.

O módulo *Monitor do Grupo de Processos* possui interação direta com três arquivos de apoio, o primeiro (**MONITORES DE GRUPO**) onde, assim que inicia sua execução, registra-se para informar ao módulo *Monitor Central* sobre sua existência, o segundo (**LOG**) é um arquivo de *log* para o módulo e o terceiro arquivo (**AMTF.CNF**) possui informações sobre a periodicidade a ser utilizada para a tomada

de *checkpoints*.

É importante ressaltar que, para registrar-se no arquivo de monitores de grupos ativos e conseqüentemente interagir com o *Monitor Central*, o monitor de grupo necessita de um identificador de grupo e o mesmo é atribuído dinamicamente através de uma função de geração de números aleatórios. Uma abordagem alternativa a este procedimento de atribuição de identificador de grupo pode ser o registro em algum arquivo de um número inteiro e consecutivo e incrementá-lo à medida que novos grupos são criados.

Este módulo também possui capacidade de comunicação, utilizando os mecanismos de *socket* para receber os *heatbeats* provenientes dos processos paralelos e o mecanismo de memória compartilhada para interagir com o módulo *Monitor Central*.

Foi escolhido o mecanismo *socket* pois os vários processos que compõem a aplicação paralela são processados em nós remotamente dispersos e este mecanismo de comunicação viabiliza a troca de informações remotas mais facilmente. Para o estabelecimento deste *socket* é empregado o protocolo de transporte UDP e como número do porto é utilizado o PID do processo MPIRUN acrescido de um valor constante, buscando com isso evitar a utilização de algum porto previamente alocado ao sistema operacional.

Ao ser inicializado pelo módulo *Executor*, o *Monitor do Grupo de Processos* registra suas informações no arquivo de controle de monitores de grupos, lido pelo *Monitor Central*, abre um canal de comunicação (por *socket*) com a aplicação paralela e entra em um estado de espera pelas sinalizações de *heartbeat* provenientes dos processos paralelos.

O recebimento destas sinalizações, representado na figura 3.8, é interpretado como a continuidade de funcionamento dos respectivos processos paralelos sob seu monitoramento e a ausência entendida como uma falha ocorrida em algum deles.

Estas sinalizações são a base para a execução dos procedimentos de *failover* e *failback*.

Eventualmente, visando evitar transformar o subsistema em um ponto de gargalo, e conseqüentemente em um ponto de falha, pode-se enviar estes *heartbeats* através de uma rede alternativa que interligue os nós que compõem o *cluster* de computadores. Um procedimento análogo pode ser realizado para se enviar as mensagens que deverão ser enviadas pelos monitores aos seus arquivos de *log*.

Figura 3.8: Sinalização de *heartbeat*.

Caso uma falha da classe *fail stop* ocorra em algum dos processos paralelos, o mesmo deixa de sinalizar sua capacidade operacional ao *Monitor do Grupo de Processos* e este inicia o procedimento de *failover*, ou seja, solicita que todos os processos paralelos da aplicação sejam interrompidos e em seguida, solicita que a aplicação seja reiniciada a partir do último *checkpoint* registrado pelos processos paralelos e armazenados em seus respectivos arquivos de contexto.

O ambiente de execução e monitoramento AMTF trata apenas das falhas da classe *fail stop*, sendo que outros tipos de falhas como a bizantina e *crash failure* não são cobertas pelo mesmo.

Além das sinalizações de continuidade de funcionamento (*heartbeat*), cada um dos processos paralelos deve sinalizar também sua autenticação e seu encerramento junto ao *Monitor do Grupo de Processos*, sendo que todas as sinalizações recebidas ou emitidas por este monitor são registradas em um arquivo de *log* para acompanhamento de sua execução.

Da maneira análoga ao que acontece com os processos paralelos, cada *Monitor do Grupo de Processos* deve também sinalizar constantemente, via memória compartilhada, para o *Monitor Central* informando sobre sua capacidade operacional, conforme figura 3.8.

Uma outra tarefa desempenhada pelo *Monitor do Grupo de Processos* é a de enviar, periodicamente, solicitações ao MPIRUN para a tomada de *checkpoints* dos processos paralelos (estados locais) e da aplicação como um todo (estado global). Uma funcionalidade que o ambiente disponibiliza é a flexibilização desta periodicidade, deixando que o usuário a especifique por intermédio de uma chamada disponibilizada pela biblioteca do ambiente e também através de um arquivo de apoio, sendo que o conteúdo deste arquivo é recuperado no início de cada execução do *Monitor de Grupo de Processos*.

Quando o *Monitor do Grupo de Processos* sinaliza ao MPIRUN o momento de se tomar um *checkpoint*, este deve solicitar aos processos paralelos que registrem seus estados locais em arquivos e em seguida também faz o mesmo, gerando um arquivo com o estado global da aplicação. Estes arquivos serão armazenados no *Sistema de Arquivos Tolerante a Falhas* e o procedimento é demonstrado na figura 3.9.

O procedimento adotado pelo *Monitor do Grupo de Processos* é o de *checkpointing* coordenado e bloqueante, ou seja, o *checkpoint* é tomado após todos os processos estarem sincronizados e suas mensagens terem sido transmitidas (procedimento coordenado), sendo que para a sincronização utiliza-se o mecanismo nativo disponibilizado pelo BLCR.

A razão pela adoção deste procedimento ocorreu devido ao fato do LAM/MPI utilizando BLCR trabalhar desta maneira e também pelo fato de que as técnicas coordenadas e bloqueantes evitam o aparecimento do efeito dominó [JOHNSON 2002].

Figura 3.9: Solicitação para tomada de *checkpoints* locais e global.

Toda sinalização recebida ou emitida pelo *Monitor do Grupo de Processos*, por *socket* ou memória compartilhada, é armazenada em um arquivo de *log* juntamente com outras ações desempenhadas pelo módulo, permitindo assim consultas sobre o comportamento do respectivo monitor e da aplicação.

Um detalhamento das ações tomadas pelo *Monitor do Grupo de Processos* pode ser visto na seção 3.5, onde são apresentados os ciclos de vida de todos os módulos do ambiente AMTF em diversas situações.

3.2.7. O Sistema de Arquivos Tolerante a Falhas

Para o armazenamento dos arquivos de *checkpoints* podem ser utilizados dispositivos de armazenamento estáveis, centralizados como os dispositivos de RAID ou então distribuídos como os sistemas de arquivos do tipo *journal* (CODA, por

exemplo). Qualquer que seja a técnica escolhida, ela deve garantir a estabilidade, a disponibilidade de funcionamento e a capacidade de atendimento, além da confiabilidade das informações armazenadas e recuperadas.

O ambiente AMTF não exige um ou outro tipo de sistema de arquivos pois ele será apenas um usuário do mesmo, porém é importante que o sistema de arquivos seja eficiente no atendimento das requisições, seja estável e compartilhável.

3.3. Alocação dos módulos aos nós do *Cluster* de Computadores

Para fins de exemplificação, a figura 3.10 demonstra como pode ser uma possível distribuição dos módulos de *software* que compõem o ambiente AMTF através dos nós de um *cluster* de computadores.

Na figura 3.10 observam-se os módulos *Monitores Centrais* e *Monitores de Réplicas* posicionados em cada um dos nós que compõem o *cluster*, eles são carregados quando o sistema operacional foi colocado em funcionamento.

Através de um nó do cluster, no caso o “Nó 0”, o usuário solicita a execução de uma aplicação paralela via o módulo *Executor* e este por sua vez, aciona o MPIRUN e os demais processos que compõem a aplicação, distribuindo-os pelos nós. Após o módulo *Executor* ativar a *Aplicação Paralela*, ele ativa também, no mesmo nó, o módulo *Monitor do Grupo de Processos*.

Caso seja necessário promover um desligamento do *cluster*, o usuário pode, a partir de qualquer um dos nós ativar o módulo *Controlador AMTF* ou então deixar esta execução a cargo do próprio sistema operacional.

Figura 3.10: Distribuição dos módulos pelos nós de um *Cluster*.

A figura 3.11 demonstra um possível cenário de como ficariam os módulos de *software* do ambiente AMTF com seus respectivos mecanismos de comunicação sobre os nós do *cluster*.

As linhas pontilhadas demonstram a utilização do mecanismo de comunicação via memória compartilhada e as linhas tracejadas demonstram o mecanismo de comunicação por *socket*. É importante observar que a comunicação via memória compartilhada é feita apenas quando existe a necessidade da comunicação local entre os processos (no caso, entre o *Monitor Central* / *Monitor de Réplica* e *Monitor Central* / *Monitor do Grupo de Processos*) e a realizada por *socket* quando for necessária a comunicação remota entre os processos (no caso, *Monitor Central* / *Monitor Central*, *Processos* / *Monitor do Grupo de Processos* e *Controlador AMTF* / *Monitores Centrais*).

Figura 3.11: Módulos do AMTF com os mecanismos de comunicação em um *Cluster*.

3.4. A Biblioteca AMTF

Apesar da tomada de *checkpoints* ser feita automaticamente, pode ser interessante que o desenvolvedor decida em quais pontos de seu código-fonte o contexto de sua aplicação necessite ser registrado e também adequar a periodicidade dos registros conforme as características de sua aplicação.

Para a situação da tomada de *checkpoint* manual, o ambiente disponibiliza uma chamada simplificada fornecida por uma biblioteca complementar que obriga a tomada do registro quando processada. Uma vez tomado o *checkpoint*, em decorrência desta chamada, a periodicidade da tomada automática começa a ser

calculada novamente a partir do zero.

Na situação onde se deseja alterar a periodicidade das tomadas de *checkpoints* em tempo de execução, o desenvolvedor também pode utilizar uma outra chamada disponibilizada pela biblioteca que realiza esta função.

A biblioteca contém basicamente quatro chamadas, uma de autenticação, uma de tomada de *checkpoint*, uma de alteração da periodicidade e uma de encerramento. A listagem 3.1 demonstra a utilização destas chamadas em um código-fonte.

```
#include <mpi.h>
#include <stdio.h>
#include "/amtf/bibliotecas/amtf.h" // Obtendo acesso às chamadas.

#define MAX 256
#define tag 1

int main(int argc, char **argv) {

    int passos=15, quantos, rank, retorno, linhas=100, colunas=100, i=0, j=0, etapa=1;
    float a[linhas][colunas], acc_trabalhador=0;
    char mensagem[MAX];

    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &quantos);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    amtf_autenticar(argv[argc -1]); // Abertura de Canais e Iniciar Heartbeats.

    if (quantos != 2) {
        if (rank == 0) fprintf(stderr, "Este programa funciona apenas com 2 processos\n");
    } else {
        system("clear");

        amtf_periodicidade(60); // Tomar checkpoint automaticamente a cada 60 segundos.

        for (i=0;i < linhas;i++) {
            for(j=0;j < colunas;j++) {
                a[i][j]=3.1415;
            }
        }

        amtf_tomar_checkpoint(); // Tomar checkpoint deste ponto.

        retorno=0;
```

```

amtf_periodicidade(60); // Tomar checkpoint automaticamente a cada 60 segundos.

while ( retorno <= passos ) {
    if (rank == 0) {
        MPI_Recv(mensagem, MAX, MPI_CHAR, 1, tag, MPI_COMM_WORLD, &status);
        printf("Etapa: %d - Passo: %d - Recebi o valor: %s\n",etapa,retorno,mensagem);
    } else {
        for (i=0;i < linhas;i++) {
            for(j=0;j < colunas;j++) {
                a[i][j]=3.1415;
                acc_trabalhador=acc_trabalhador+a[i][j];
            }
        }
        retorno++;
        sleep(2);
        sprintf(mensagem,"%f",(acc_trabalhador/1950));
        MPI_Send(mensagem, strlen(mensagem)+1, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
    }

amtf_tomar_checkpoint(); // Tomar checkpoint deste ponto.

}

amtf_encerrar(); // Encerrar o envio de Heartbeats e fechar os Canais de comunicação.

if (rank == 0) printf("Fim do Processamento.\n");
MPI_Finalize();
return 0;
}

```

Listagem 3.1: Utilização das chamadas disponibilizadas pela biblioteca AMTF.

A chamada de autenticação tem como função abrir os canais de comunicação e iniciar o envio dos *heartbeats* da aplicação para o seu respectivo *Monitor do Grupo de Processos* sendo obrigatório o seu uso se for desejado que a aplicação seja monitorada. É importante lembrar que a solicitação desta chamada poderia ser inserida na função `MPI_INIT()` do LAM/MPI e assim a mesma seria acionada automaticamente pela aplicação.

A chamada de tomada de *checkpoint*, como o próprio nome diz, é a responsável por solicitar ao *Monitor do Grupo de Processos* que um *checkpoint* seja tomado. Deve ser utilizada apenas quando o desenvolvedor deseja forçar a tomada do *checkpoint*.

A chamada de alteração da periodicidade, também como o próprio nome já adianta, altera a periodicidade da tomada automática de *checkpoints*. Ela deve ser utilizada pelo desenvolvedor quando as características da aplicação paralela exigirem um intervalo diferente daquele especificado como padrão.

E a chamada de encerramento é responsável por sinalizar ao *Monitor do Grupo de Processos* que a aplicação paralela está encerrando seu processamento e fechar os canais de comunicação que foram abertos. Analogamente à chamada de autenticação, a chamada de encerramento poderia ser inserida na função `MPI_FINALIZE()` do LAM/MPI e ser acionada automaticamente quando a referida função fosse solicitada no código da aplicação.

3.5. Ciclos de Vida

Torna-se interessante descrever o ciclo de vida e o comportamento do ambiente AMTF na ausência e presença de falhas para melhor entender o seu funcionamento. Desta forma, as figuras 3.12 até a 3.23, apresentam algumas situações de aplicação e do próprio ambiente sem e com a presença de falhas e quais procedimentos são tomados.

O procedimento padrão de funcionamento do ambiente e da aplicação é descrito pela figura 3.12. Nela vê-se o módulo *Monitor Central* enviando *heartbeats* para o módulo *Monitor de Réplica* (1) e vice-versa (2), este é o procedimento de auto-monitoramento implementado para estes dois módulos.

Em seguida, tem-se o módulo *Executor* inicializando uma *Aplicação Paralela* (3) e seu respectivo *Monitor do Grupo de Processos* (4), sendo que a primeira ação tomada por este último módulo é o de registrar-se (5) perante o *Monitor Central* para

que possa ser monitorado pelo mesmo. Os processos paralelos executam uma ação idêntica, ou seja, autenticam-se, porém junto ao seu *Monitor de Grupo de Processos* (6) para que também possam ser monitorados.

Figura 3.12: Ciclo de vida padrão dos módulos.

Após a etapa de autenticação, tanto o *Monitor de Grupo de Processos* como os processos paralelos começam a enviar os sinais de *heartbeat* (7 e 8) para os seus respectivos monitores.

Periodicamente o *Monitor de Grupo de Processos* também emite requisições (9) para que sejam tomados os *checkpoints* da *Aplicação Paralela* (10).

Os passos 7, 8, 9 e 10 são repetidos até que a *Aplicação Paralela* solicite seu encerramento (11 – figura 3.13) ao *Monitor do Grupo de Processos* e este solicite também sua finalização (12) junto ao *Monitor Central*. Após esta troca de mensagens, tanto a *Aplicação Paralela* como o seu respectivo *Monitor de Grupo de Processos* são encerrados (13 e 14).

Estes últimos passos (11, 12, 13 e 14) estão representados na figura 3.13, que demonstra o ciclo de vida completo da aplicação e dos monitores sem o aparecimento de falhas.

Figura 3.13: Ciclo de vida sem a presença de falhas.

A partir deste ponto e para todas as figuras até a 3.17, os passos de 1 a 10 são os mesmos comentados no ciclo de vida padrão, sendo que a alteração se dará com o passo 11, significando uma falha na *Aplicação Paralela*, ou no *Monitor do Grupo de Processos*, ou no *Monitor Central* ou ainda no *Monitor de Réplica*.

No caso do aparecimento de falhas na *Aplicação Paralela*, tem-se como comportamento os procedimentos descritos na figura 3.14.

Por conta do aparecimento desta falha (11), o processo interrompido deixa de emitir seu *heartbeat* ao *Monitor do Grupo de Processos* e este interpreta a ausência desta sinalização como uma falha do referido processo.

Como resposta, o *Monitor do Grupo de Processos* emite uma requisição de recuperação (12) da *Aplicação Paralela* como um todo, assim, antes da recuperação deve ser feita uma interrupção geral da *Aplicação Paralela*, para em seguida iniciar o procedimento de *restart* (13). Este procedimento de *restart* consiste em recuperar os

últimos estados locais dos processos paralelos armazenados em seus respectivos arquivos de contexto, bem como o estado global da aplicação.

Figura 3.14: Ciclo de vida com a presença de falhas na *Aplicação Paralela*.

Após a reativação da aplicação, a mesma retorna ao seu ciclo normal, ou seja, os processos retomam o envio dos *heartbeats* ao *Monitor do Grupo de Processos* e este a tomada de novos *checkpoints* (passos 7, 8, 9 e 10).

Assim que a *Aplicação Paralela* tenha todo o seu processamento realizado, ela envia o sinal de encerramento de execução (14) ao seu *Monitor de Grupo de Processos* e este também o faz (15) para o *Monitor Central*. Em seguida, tanto a *Aplicação Paralela* como o *Monitor do Grupo de Processos* correspondente encerram suas atividades (16 e 17).

A figura 3.15 representa a ocorrência de uma falha no *Monitor do Grupo de Processos* e descreve o procedimento tomado pelo *Monitor Central*.

Na referida figura, o passo 11 demonstra o surgimento de uma falha no *Monitor do Grupo de Processos*. Quando esta falha ocorre, o referido monitor perde sua capacidade de enviar os sinais de *heartbeat* ao *Monitor Central* e este interpreta esta ausência como uma interrupção do *Monitor do Grupo de Processos*.

Em resposta, o *Monitor Central* coloca em execução (12) uma nova cópia do

Monitor do Grupo de Processos, mantendo suas características iniciais como identificador, portas e endereços de memória compartilhada, visto que tais informações ficam registradas no arquivo de monitores de grupos. Esta nova cópia consiste na verdade de um novo processo, que opera com as informações que pertenciam ao *Monitor do Grupo de Processos* afetado pela falha.

Figura 3.15: Ciclo de vida com a presença de falhas no *Monitor do Grupo de Processos*.

A partir do início da execução desta cópia do monitor, os procedimentos voltam a ser os mesmos de uma execução sem falhas, ou seja, envio de *heartbeats* por parte dos processos paralelos e do novo *Monitor do Grupo de Processos* (7 e 8), pedidos de tomada de *checkpoint* (9 e 10), solicitações de encerramento (13 e 14) e o encerramento propriamente dito (15 e 16).

A próxima representação do ciclo de vida, apresentada na figura 3.16, demonstra como o ambiente se comporta com o surgimento de uma falha no *Monitor Central*.

Mais uma vez, o passo 11 é marcado como aparecimento de uma falha, agora no módulo *Monitor Central*.

A falha faz com que o *Monitor Central* interrompa o envio de *heartbeats* ao *Monitor de Réplica* e este aciona uma nova cópia do *Monitor Central* (12), mantendo suas configurações iniciais.

Figura 3.16: Ciclo de vida com a presença de falhas no *Monitor Central*.

Os demais passos, de 13 a 16, seguem também o comportamento padrão do ambiente representado e comentado nas figuras anteriores.

E na figura 3.17, tem-se uma representação análoga a da figura 3.16, porém representando a falha do módulo *Monitor de Réplica*.

A ação tomada aqui é similar àquela descrita para a situação anterior (de falha no *Monitor Central*), ou seja, a interrupção do *Monitor de Réplica* faz com que os sinais de *heartbeat* emitidos pelo mesmo ao *Monitor Central* deixem de ser enviados.

Dessa forma, o *Monitor Central* interpreta isso como uma falha no *Monitor de Réplica* e ativa uma nova cópia do mesmo, mantendo suas configurações iniciais, e os passos seguintes seguem novamente o comportamento padrão.

Figura 3.17: Ciclo de vida com a presença de falhas no *Monitor de Réplicas*.

As figuras 3.18, 3.19 e 3.20 demonstram os passos para se ativar, interromper e reativar o ambiente AMTF.

Sobre a ativação do ambiente (figura 3.18), o procedimento é bastante simples; ela é realizada através da execução do *Controlador AMTF*, que, ao ser acionado emite o comando de inicialização (1) do *Monitor Central* e encerra-se (6). Este por sua vez, recupera as informações para a inicialização (2), aciona o *Monitor de Réplica* (3) e ambos permanecem em execução (4 e 5) até que um comando de encerramento seja emitido ou uma falha os interrompa.

Figura 3.18: Ciclo de vida da ativação do Ambiente AMTF.

No caso de encerramento do ambiente AMTF, representado pela figura 3.19, o *Controlador AMTF* envia um pedido para tal ação ao *Monitor Central* (1) e este replica o pedido para todos os demais monitores em execução (3 e 4) no respectivo nó, porém antes desta ação, o *Monitor Central* deve registrar em arquivo (2) as informações sobre todos os *Monitores dos Grupos de Processos* e suas respectivas aplicações monitoradas e que foram interrompidas.

Figura 3.19: Ciclo de vida do encerramento do Ambiente AMTF.

Após realizar a difusão do pedido de encerramento, todos encerram suas execuções (5, 6 e 7).

Eventualmente, ao ser solicitado o encerramento do ambiente AMTF, alguma aplicação pode estar sendo processada e o encerramento provocaria uma interrupção em seu processamento. Quando o ambiente viesse a ser colocado em execução, novamente tal aplicação deveria ser reativada automaticamente e continuar o seu processamento do ponto onde parou.

Este procedimento está descrito na figura 3.20. É importante lembrar que, quando a operação de encerramento foi solicitada, as informações sobre os grupos de processos e aplicações que eventualmente estivessem em execução foram armazenadas em um arquivo de aplicações interrompidas e é este arquivo que deve ser lido pelo *Monitor Central* para se reativar os mesmos grupos de processos e

aplicações.

Figura 3.20: Ciclo de vida da reativação do Ambiente AMTF.

No caso da figura 3.20, o *Controlador AMTF* deve colocar o ambiente em operação acionando o *Monitor Central* (1) e encerrando-se (6), o *Monitor Central* recupera as informações de inicialização (2) e ativa o *Monitor de Réplica* (3) ficando ambos em execução (4 e 5); estes passos são os mesmos descritos em uma inicialização tradicional (figura 3.18).

A diferença do procedimento de reativação reside no fato do *Monitor Central* recuperar as informações armazenadas no arquivo de aplicações interrompidas (7) e usá-las para recolocar os grupos de processos e suas respectivas aplicações em execução (8).

As próximas figuras, mais especificamente as figuras 3.21 e 3.22 descrevem o funcionamento do *Monitor Central Virtual* sem o aparecimento de falhas nos *Monitores Centrais* locais e com o surgimento de falha em algum participante do grupo de *Monitores Centrais*.

Na figura 3.21 identifica-se os passos (1) e (2) como sendo o procedimento que todos os *Monitores Centrais* locais e de *Réplicas* utilizam para o auto-monitoramento local (*heartbeat* através da memória compartilhada), porém,

periodicamente, todos os *Monitores Centrais* enviam para todas as demais cópias sinais de *heartbeat* através de *sockets*, passos (3) até (8), que indicam a sua operacionalidade.

Figura 3.21: Ciclo de vida do funcionamento do *Monitor Central Virtual*.

Já a figura 3.22 representa uma situação de falha ocorrida nos processos *Monitor Central* local e de *Réplicas* que executam em um dos nós de computação. De maneira semelhante ao da figura 3.21, os passos (1) e (2) demonstram o auto-monitoramento local entre os *Monitores Central* e de *Réplica*.

Figura 3.22: Ciclo de vida com falha em uma cópia do *Monitor Central Virtual*.

No caso desta representação, têm-se que o *Monitor Central 1* e seu respectivo *Monitor de Réplica* sofreram uma falha e encerraram suas execuções. Em

razão desta dupla interrupção, nenhum destes monitores tem condições de reativar um ao outro.

Uma vez que todos os *Monitores Centrais* trocam sinais de *heartbeat* uns com os outros (passos 3 até 8), a ausência dos sinais do *Monitor Central 1* é percebida pelos demais monitores (passos 5 e 6).

Em resposta a essa ausência dos *heartbeats* e após um tempo de espera, o *Monitor Central* de ranqueamento anterior ao de ranqueamento 1, o 0 no caso, é o responsável por enviar o comando remoto que reativa o *Monitor Central* daquele nó que apresentou a falha (9). Assim que a cópia do *Monitor Central* volta a funcionar ela ativa o seu *Monitor de Réplica* (10).

Nos cenários abordados pelos ciclos de vida anteriores trabalhou-se apenas com a interrupção dos *Monitores Central*, de *Réplica* o do *Grupo de Processos* ou da *Aplicação Paralela* e ainda com a operação de *shutdown* do *cluster* e sua reativação.

Porém, não apenas os processos que compõem o ambiente AMTF podem falhar, mas também o *hardware* dos nós de computação onde estes processos estão sendo executados podem vir a sofrer uma interrupção e muitas vezes o próprio *cluster* como um todo pode ser interrompido. Como exemplo destas interrupções podem ser citadas a interrupção do funcionamento ocorrido por uma queda de energia generalizada ou mesmo queda de um nó em particular.

Assim torna-se interessante descrever quais seriam as ações que o ambiente AMTF tomaria quando eventos desta natureza viessem a ocorrer.

A figura 3.23 descreve os passos que o ambiente AMTF tomaria caso um nó fosse abruptamente interrompido, por uma queda de energia por exemplo, e como seria a restauração das aplicações que estivessem sob o seu gerenciamento.

O comportamento é similar ao descrito na reativação do ambiente AMTF

(figura 3.20), ou seja, quando o sistema operacional volta a ser carregado pelo computador, o *Controlador AMTF* é acionado na forma de um serviço do sistema operacional e este por sua vez ativa o módulo *Monitor Central* (1). O *Monitor Central* faz uma leitura dos arquivos de configuração (2) para poder colocar-se operacional e inicia a execução de seu *Monitor de Réplica* (3).

Figura 3.23: Ciclo de vida com reativação de nó após interrupção.

A partir deste momento, o *Monitor Central* começa a enviar seus *heartbeats* para os demais monitores centrais e mantém-se em execução, juntamente como seu *Monitor de Réplica* (4) e (5) e o *Controlador AMTF* encerra seu processamento (6).

Em seguida, o *Monitor Central* recupera as informações sobre os *Monitores do Grupo de Processos* que estavam ativos no momento da interrupção e que estavam armazenadas no arquivo correspondente (7) e começa a ativá-los um a um (8) e estes iniciam a reativação de suas respectivas aplicações paralelas.

O estágio atual do ambiente AMTF não permite que uma aplicação seja encerrada através do comando *kill* de maneira convencional, isto porque ela está vinculada a um *Monitor do Grupo de Processos* e este detectará esta interrupção e acionará uma reinicialização da mesma. O mesmo acontecerá se uma aplicação gerar um *Segmentation Fault*.

3.6. A Implementação dos Módulos do Ambiente AMTF

O ambiente AMTF foi planejado para utilizar ambientes distribuídos, porém ambientes não distribuídos também podem utilizá-lo, e executando o sistema operacional Linux. Assim, o ambiente AMTF faz uso intensivo das facilidades e funcionalidades fornecidas por este ambiente operacional, utilizando-se muitas vezes programas disponibilizados pelo sistema operacional.

Para a materialização dos módulos de *software* e suas funcionalidades, utiliza-se basicamente a linguagem C e o compilador GCC, além de outras ferramentas disponibilizadas pelo sistema operacional *Linux*, como o *ksysguard* utilizado para monitorar a CPU e a rede de comunicação, o interpretador de comandos com interface caracter para inicializar e interromper o ambiente e as aplicações paralelas, além de outros aplicativos.

É imprescindível viabilizar a comunicação entre os módulos sendo que para atender a esta exigência adota-se duas abordagens, uma que viabiliza a comunicação local entre processos e outra para a comunicação remota.

A primeira abordagem, destinada aos processos que executam no mesmo nó, utiliza-se como mecanismo de comunicação a memória compartilhada, sendo que essa escolha ocorreu em razão da facilidade de manipulação e da velocidade de transferência da informação proporcionada por este meio.

A segunda abordagem, indicada para a comunicação entre os processos que executam em nós diferentes, necessitando assim de enviar mensagens através de um subsistema de comunicação, utiliza-se a comunicação baseada em *sockets*. Para esta comunicação utiliza-se o método não orientado a conexões, fornecido pelo UDP, em razão dos nós comunicantes estarem em um ambiente confinado e

controlado, assim não há problemas com a perda de pacotes,

Para viabilizar a tomada dos *checkpoints* dos processos utiliza-se programas disponibilizados pelo projeto BLCR, visto que os mesmos já estão prontos e adaptáveis ao LAM/MPI além de atenderem às necessidades exigidas pelo ambiente AMTF. Porém, o BLCR utiliza uma maneira manual para a tomada de *checkpoints* e restauração dos processos e o ambiente AMTF complementa-o com uma forma automática.

E finalmente, utiliza-se o LAM/MPI para o desenvolvimento das aplicações paralelas que são utilizadas pelo desenvolvedor e monitoradas pelo ambiente.

4. DESENVOLVIMENTO E TESTES

Para o desenvolvimento e implementação do ambiente AMTF utilizou-se como plataforma computacional um *cluster* de computadores composto por quatro nós, sendo que três deles atuaram como nós de computação (NC0, NC1 e NC2) e um outro como nó de armazenamento (NA0), conforme demonstra a figura 4.1.

Figura 4.1: Representação do *Cluster* de Computadores utilizados nos testes.

Os nós de computação executaram o sistema operacional Linux e foram configurados para trabalhar com o LAM/MPI, com o BLCR e atuaram como clientes NIS e NFS do nó de armazenamento NA0, sendo que este último tinha também o Linux como sistema operacional e atuava como servidor de NIS e NFS.

Os módulos de *software* ativados durante o carregamento do sistema operacional junto aos nós de computação e aqueles solicitados pelos usuários foram posicionados em uma área acessível por todos os nós.

Esta área estava fisicamente localizada junto ao sistema de arquivos do nó de armazenamento e com o compartilhamento habilitado. Assim esta área comum foi

logicamente montada no sistema de arquivos local dos nós de computação, conforme mostra a figura 4.2.

Figura 4.2: Representação lógica do sistema de arquivos dos nós.

Também foram compartilhados os diretórios dos usuários (*home directories*), permitindo que os mesmos pudessem obter acesso aos seus arquivos pessoais a partir de qualquer nó de computação do *cluster* que viessem a utilizar.

4.1. Uso do Ambiente AMTF

Para se utilizar o ambiente foi necessário ativá-lo e isso ocorreu por intermédio do módulo *Controlador AMTF*. Sua ativação pôde, como mencionado em

seções anteriores, ser feita através do sistema operacional (na forma de serviço) ou então através da linha de comando, bastando adicionar nos arquivos do sistema ou digitar o comando */amtf/amtf start*, respectivamente. Este procedimento colocou em execução os módulos *Monitores Central e de Réplica*.

Para executar uma aplicação paralela utilizando-se das funcionalidades do ambiente AMTF, inicializou-se a mesma através do módulo *Executor*. Um exemplo de como isso pode ser feito é através da seguinte linha de comando:

```
$ /amtf/executor mpirun [opções do mpirun] programa_mpi [parâmetros do  
programa_mpi] pid
```

As partes marcadas em destaque são obrigatórias, principalmente a opção *pid* colocada no final do comando, pois este parâmetro será utilizado pelo código do módulo *Monitor do Grupo de Processos*.

Foram realizadas duas baterias de testes sendo que a primeira consistiu em verificar qual seria o impacto produzido pelo ambiente no *cluster*, principalmente quanto à utilização da rede, da capacidade de processamento e do acesso ao sistema de arquivos. Já a segunda bateria focou na funcionalidade (interrupção e reativação dos módulos – *failover* e *failback*) do ambiente e na verificação da materialização dos ciclos de vida propostos.

4.2. Padrões de Comportamento do Ambiente AMTF

O que motivou a primeira bateria de testes foi o surgimento de questionamentos relacionados à constante comunicação produzida pelos módulos para viabilizar o mecanismo de *heartbeat*, pela adição de novos processos ao sistema e, conseqüentemente a carga de processamento exigida pelos mesmos

além do procedimento constante de gravação das informações nos arquivos de *log* e dos arquivos de *checkpoint*.

Buscando esclarecer estes questionamentos foram realizados cinco experimentos, utilizando-se como aplicação o código-fonte descrito na listagem 4.12, estando seus resultados descritos adiante.

A aplicação utilizada para esta bateria de testes calculava o valor aproximado de PI e foi escolhida em razão de exigir uma quantidade de computação considerável (dependendo da aproximação desejada), do tempo de execução necessário para ser concluída e também de poder ser utilizada em múltiplos nós. Com respeito à comunicação que ela necessitava, esta dependia da quantidade de nós e processos que seriam utilizados para os cálculos.

Para realizar os monitoramentos dos experimentos foram utilizados os *softwares ksysguard*, que permite a visualização na forma de gráficos de certos objetos do ambiente computacional e o *ksnapshot* que permite capturar imagens da tela e gravá-las em arquivos; ambos os *softwares* estão presentes no ambiente gráfico *kde*, encontrado no Linux.

Nos experimentos, quando o objeto de análise era a CPU utilizou-se como unidade de mensuração a porcentagem de utilização da mesma, quando o objeto da observação era a rede de comunicação utilizou-se a quantidade de pacotes recebidos e transmitidos e, quando o objeto era a quantidade de processos, utilizou-se a quantia dos mesmos presentes no sistema. A descrição dos experimentos bem como de seus respectivos resultados é apresentada na seqüência.

4.2.1. O Experimento I

O Experimento I consistiu em monitorar o comportamento do sistema computacional sem a presença do ambiente AMTF, de aplicações paralelas ou da

inserção de novos processos no sistema, visando estabelecer um padrão de comportamento para o ambiente computacional.

Neste experimento, estavam presentes apenas os processos do próprio sistema operacional, o *software* de monitoramento *ksysguard*, o *ksnapshot* e o *daemon* do LAM/MPI (o *lamd*). Os resultados deste monitoramento podem ser vistos nas figuras 4.3, 4.4, 4.5 e 4.6.

A figura 4.3 demonstra como foi a utilização típica do processador sendo que no referido gráfico a escala varia de 0% a 24%; optou-se por não utilizar a escala de 0 a 100% em razão da melhor visualização com a escala menor.

Observa-se que em alguns momentos a utilização da CPU atinge picos de aproximadamente 12% (exceto no início do monitoramento onde o pico atingiu cerca de 20%), porém na maior parte do tempo a utilização do processador gira em torno de 2 a 3% de sua capacidade de processamento.

Já as figuras 4.4 e 4.5 representam a utilização da rede; a escala utilizada ficou entre 0 e 24 pacotes também em razão da melhor visualização. A figura 4.4 demonstra a quantidade de pacotes recebidos e a figura 4.5 a quantidade de pacotes transmitidos.

Nas figuras em questão observa-se que a utilização da rede durante o monitoramento foi mínima, esporadicamente eram vistos entre 5 a 6 pacotes trafegando pela rede; provavelmente este tráfego era gerado pela atualização dos cachês entre o servidor e seus clientes.

Já a figura 4.6 demonstra a quantidade de processos presentes no sistema durante o monitoramento, sendo que a escala utilizada foi de 0 a 100 processos (esta escala foi mantida ao longo dos cinco experimentos).

Nela observou-se que a quantidade de processos não sofreu variação durante

o período em que o monitoramento foi feito, mantendo a mesma quantidade de processos ao longo de todo o experimento.

Figura 4.3: Utilização da CPU durante o Experimento I.

Figura 4.4: Quantidade de pacotes recebidos durante o Experimento I.

Figura 4.5: Quantidade de pacotes transmitidos durante o Experimento I.

Figura 4.6: Quantidade de processos no sistema durante o Experimento I.

4.2.2. O Experimento II

O Experimento II teve como objetivos verificar qual o impacto produzido pelos processos *Monitor Central* e *Monitor de Réplica* no funcionamento e no desempenho do ambiente computacional e também determinar um padrão de comportamento para o mesmo. Os resultados obtidos por este segundo experimento são representados nas figuras 4.7, 4.8, 4.9 e 4.10.

Neste experimento estavam presentes os processos do próprio sistema operacional, os *softwares* *ksysguard*, *ksnapshot* e o *daemon* *lamd* (a exemplo do experimento anterior) e foram adicionados os *Monitores Central* e de *Réplica*, porém não foi inicializado nenhum processo novo ou aplicação ao longo do monitoramento.

Observa-se na figura 4.7, quando comparada à figura 4.3, que os picos de processamento atingem os 20%, porém percebe-se ao longo do monitoramento um pequeno aumento na utilização da CPU, permanecendo na maior parte do tempo aproximadamente, entre 3 e 4%, em média.

Com base nesta comparação pode-se concluir que a carga de processamento imposta pelos *Monitores Central* e de *Réplica* não é substancial (cerca de 1% a mais) a ponto de comprometer a capacidade de processamento do sistema computacional.

As figuras 4.8 e 4.9 demonstram a utilização da rede e, quando comparadas às figuras 4.4 e 4.5, apresentam a um aumento considerável na utilização da mesma, ficando a maior parte do tempo entre 6 e 7 pacotes recebidos e 9 pacotes transmitidos. Enquanto que as figuras 4.4 e 4.5 mostram uma utilização esporádica da rede, as figuras 4.8 e 4.9 demonstram uma utilização mais constante.

Esta utilização constante ocorre em razão do *Monitor Central* ficar periodicamente enviando e recebendo *heartbeats* para/dos *Monitores Centrais* localizados nos demais nós de computação do *cluster* de computadores e também

pelo fato dos *Monitores Central* e de *Réplica* locais registrarem, em seus arquivos de *log*, as operações realizadas.

Figura 4.7: Utilização da CPU durante o Experimento II.

Figura 4.8: Quantidade de pacotes recebidos durante o Experimento II.

Figura 4.9: Quantidade de pacotes transmitidos durante o Experimento II.

Figura 4.10: Quantidade de processos no sistema durante o Experimento II.

É importante lembrar que os arquivos de *log* são fisicamente mantidos na área compartilhada pelo nó de armazenamento aos nós de computação e,

conseqüentemente, a rede deve ser utilizada para que as operações de manipulação dos referidos arquivos possam ser realizadas.

Esta transmissão constante de informações para o nó de armazenamento pode ser a razão de se realizarem mais transmissões do que recepções. Essa diferença pode ser vista confrontando-se os gráficos das figuras 4.8 e 4.9, pois observa-se um maior número de transmissões do que de recebimentos de pacotes. Esses pacotes extras mostrados pela figura 4.9 são, provavelmente, aqueles que levam as informações para os arquivos de *log*.

Ainda com base nas figuras 4.8 e 4.9 pode-se concluir então que os *Monitores Central* e de *Réplicas* acabam por adicionar uma carga para a rede por conta de seus *heartbeats* e principalmente pela manipulação dos arquivos de *log*, porém esta carga não é suficiente para comprometer o funcionamento do sistema computacional e pode ser reduzida caso se redefina o que efetivamente deve ser registrado nos arquivos de *log*, bem como a periodicidade do *heartbeat*.

Quanto ao monitoramento da quantidade de processos, a figura 4.10 mostra que, no início do monitoramento, o sistema contava com uma certa quantidade de processos. Ao ser solicitado o início da execução dos *Monitores Central* e de *Réplica* esse número aumentou e, no término de suas execuções o sistema voltou a ter a quantidade de processos iniciais.

4.2.3. O Experimento III

O objetivo do Experimento III foi o de determinar qual a necessidade de processamento e de rede uma aplicação utilizando-se do LAM/MPI, não utilizando as chamadas disponibilizadas pela biblioteca do ambiente AMTF, exigiria do sistema além de determinar também um padrão de comportamento para a mesma.

Uma vez iniciado o monitoramento, a aplicação paralela foi ativada através do

interpretador de comandos e os resultados estão representados nas figuras 4.11, 4.12, 4.13 e 4.14.

Figura 4.11: Utilização da CPU durante o Experimento III.

Figura 4.12: Quantidade de pacotes recebidos durante o Experimento III.

Figura 4.13: Quantidade de pacotes transmitidos durante o Experimento III.

Figura 4.14: Quantidade de processos no sistema durante o Experimento III.

Neste experimento estavam presentes os processos do próprio sistema operacional, os *softwares* de monitoramento *ksysguard* e do *ksnapshot* e o *daemon lamd* (a exemplo dos experimentos anteriores) e foram adicionados um interpretador de comandos e uma aplicação paralela (o código-fonte da listagem 4.12, mas sem as chamadas da biblioteca AMTF).

A utilização da CPU está representada na figura 4.11. Diferentemente dos experimentos anteriores notou-se uma utilização considerável da CPU enquanto a aplicação paralela esteve em funcionamento.

Nota-se na figura que, antes da execução da aplicação paralela, o uso da CPU ficava em torno de 15% e após a sua inicialização a utilização saltou para, aproximadamente, 96% e assim ficou até o seu encerramento.

Assim, conclui-se aqui que a aplicação em questão utilizou efetivamente a CPU para o seu processamento, acrescentando uma carga de processamento expressiva.

Quanto à utilização da rede neste experimento, as figuras 4.12 e 4.13 apresentam os resultados obtidos.

Observa-se nas respectivas figuras que a utilização da rede foi mínima, bem próximo dos resultados obtidos no experimento I, sendo que a diferença está em um pico (cerca de 32 pacotes, que obrigou a alteração na escala em relação aos Experimentos I e II) de utilização ocorrida no momento em que a aplicação foi solicitada e colocada em processamento.

Este pico pode ser justificado em razão da necessidade de se recuperar o código da aplicação, que se encontrava no nó de armazenamento, através da rede e também da comunicação entre os processos relacionados com a aplicação paralela.

Os resultados sobre a utilização da rede neste experimento demonstram que

a aplicação em questão não utilizou os recursos de rede constantemente, apenas esporadicamente.

Sobre a quantidade de processos (figura 4.14), tem-se que o monitoramento começou com uma quantidade de processos, sendo a mesma acrescida pela aplicação paralela. Durante a execução da mesma, nenhum outro processo foi admitido pelo sistema e, ao término de sua execução, a quantidade voltou a ser a mesma de antes.

4.2.4. O Experimento IV

O Experimento IV teve como objetivo verificar como o sistema de computação se comportaria quando estivesse em execução os *Monitores Central* e de *Réplica* juntamente com a aplicação paralela, esta porém sem utilizar as chamadas da biblioteca do ambiente AMTF.

Neste experimento estavam presentes os processos do próprio sistema operacional, os *softwares* de monitoramento *ksysguard* e do *knapshot* (a exemplo dos experimentos anteriores) e foram adicionados um interpretador de comandos, o *daemon* do LAM/MPI (o *lamd*), os *Monitores Central* e de *Réplica* e a aplicação paralela.

As figuras 4.15, 4.16, 4.17 e 4.18 representam os resultados obtidos durante o monitoramento deste experimento.

Com respeito à utilização da CPU, não se detectou mudança significativa no uso do recurso, ou seja, utilizou-se cerca dos mesmos 96% observados no experimento anterior e o tempo de execução da aplicação paralela praticamente também não sofreu alteração.

Assim, conclui-se que a necessidade de carga de processamento exigida

pelos *Monitores Centrais* e de *Réplica* não interferiu substancialmente no tempo de execução da aplicação paralela, ou seja, suas execuções não comprometem o uso da CPU ou a execução de outras aplicações.

Figura 4.15: Utilização da CPU durante o Experimento IV.

Figura 4.16: Quantidade de pacotes recebidos durante o Experimento IV.

Figura 4.17: Quantidade de pacotes transmitidos durante o Experimento IV.

Figura 4.18: Quantidade de processos no sistema durante o Experimento IV.

Quanto ao uso da rede (figuras 4.16 e 4.17), este recurso também não sofreu alterações substanciais se comparados com os resultados obtidos no Experimento II e representados nas figuras 4.8 e 4.9, pois a aplicação paralela em questão praticamente não utilizava o recurso da rede (conforme figuras 4.12 e 4.13).

Praticamente, todo o tráfego de rede gerado foi em decorrência da troca de *heartbeats* e do acesso ao sistema de arquivos compartilhado pelo nó de armazenamento pelos *Monitores Centrais* e de *Réplica*, fato já detectado no Experimento II.

Neste experimento, a exemplo dos anteriores, também foi detectada uma maior quantidade de pacotes na transmissão do que na recepção.

Com relação à quantidade de processos no sistema, vê-se na figura 4.18 o momento em que são carregados os monitores e a aplicação paralela e, conseqüentemente, o momento em que eles deixam o sistema.

4.2.5. O Experimento V

O último dos experimentos teve como objetivo observar o comportamento do sistema computacional quando estivessem em funcionamento o ambiente AMTF juntamente com aplicações paralelas, estando estas utilizando as chamadas disponibilizadas pela biblioteca provida pelo ambiente AMTF e, conseqüentemente tomando periodicamente *checkpoints*.

Neste experimento estavam presentes os processos do próprio sistema operacional, os *softwares* de monitoramento *ksysguard* e do *ksnapshot* (a exemplo dos experimentos anteriores) e foram adicionados um interpretador de comandos, o *daemon* do LAM/MPI (o *lamd*), os *Monitores Central* e de *Réplica* e a aplicação paralela propriamente dita. Os resultados deste experimento estão representados nas figuras 4.19, 4.20, 4.21 e 4.22.

Figura 4.19: Utilização da CPU durante o Experimento V.

Figura 4.20: Quantidade de pacotes recebidos durante o Experimento V.

Figura 4.21: Quantidade de pacotes transmitidos durante o Experimento V.

Figura 4.22: Quantidade de processos no sistema durante o Experimento V.

A figura 4.19 representa como foi a utilização da CPU pelo ambiente e pela aplicação, sendo que foi observado que o percentual de utilização do processador ficou nos mesmos níveis dos experimentos III e IV, porém a aplicação levou mais

tempo para ser concluída. Detalhes sobre este atraso são apresentados mais adiante e comentados na figura 4.23.

Quanto à utilização da rede (figuras 4.20 e 4.21), nota-se um aumento em seu uso, fazendo com que, em alguns momentos, o pico chegue próximo aos 96 pacotes no caso da recepção e próximo a 100 pacotes na transmissão.

Isso ocorre em razão dos *heartbeats* enviados e recebidos. Nos experimentos II e IV apenas os *Monitores Central* e de *Réplica* os manipulavam; neste experimento, eles continuam a enviar e receber os *heartbeats* mas juntam-se a eles o *Monitor do Grupo de Processos* e os processos que compõem a *Aplicação Paralela*.

Um outro fator que contribuiu para o aumento na utilização da rede é o armazenamento dos *checkpoints* tomados. É importante lembrar que os arquivos de contextos dos processos são armazenados na área do usuário e a mesma, no ambiente computacional, localiza-se fisicamente no sistema de arquivos do nó de armazenamento.

A interação dos monitores (*Central*, de *Réplica* e do *Grupo de Processos*) com os seus respectivos arquivos de *log* também contribuiu com o aumento de utilização da rede.

Já a figura 4.22 demonstra o momento em que os *Monitores Central* e de *Réplicas*, aplicação paralela e *Monitor do Grupo de Processos* são carregados e encerrados. Um destaque especial presente na figura é para os momentos em que os *checkpoints* são tomados.

Assim, os questionamentos iniciais sobre o impacto do ambiente AMTF na utilização da rede, do processador e do acesso ao sistema de arquivos foi esclarecido, ou seja, tais recursos não foram afetados substancialmente pelos módulos de *software* do ambiente AMTF.

Figura 4.23: Comparativo dos tempos de utilização da CPU nos Experimentos III, IV e V.

Porém, conforme mencionado nessa seção, percebeu-se uma diferença no tempo de conclusão da execução ao comparar-se a utilização da CPU nos experimentos III, IV e V e esta diferença pode ser verificada através da figura 4.23, observando-se um tempo extra de cerca de 25% do tempo total exigido pelo Experimento V, referenciado pelo ponto de observação (A). É importante ressaltar que esse valor adicional de 25% não é uma constante, mas sim decorrente da

periodicidade e da quantidade de *checkpoints* que exigidos pela aplicação sob teste.

Existe uma razão para este atraso no tempo de conclusão e a mesma reside nos *checkpoints* tomados. Quando um *checkpoint* for tomado, os processos paralelos que compõem a aplicação devem sofrer uma pausa e só retornar a ser processados quando a operação de gravação dos contextos dos processos for concluída e esta parada temporária reflete diretamente no tempo final necessário para a conclusão do processamento.

Por esta razão, pode-se concluir que uma aplicação paralela que venha a utilizar o ambiente AMTF sofrerá um atraso no tempo de conclusão de seu processamento, sendo este atraso determinado pela quantidade e pela periodicidade dos *checkpoints* tomados.

Com relação ao aumento nos pacotes transmitidos e recebidos através da rede, este efeito também está relacionado com a quantidade e a periodicidade das tomadas de *checkpoint*. Desta forma, uma alteração nestes itens também influenciaria a utilização da rede.

Uma outra consideração que pode interferir no tráfego da rede reside em mudanças nos arquivos de *log* dos monitores, ou seja, ou diminuir a periodicidade com que as mensagens são enviadas para o registro ou então o tamanho destas mesmas mensagens.

4.3. Interrupção e Reativação dos Módulos do Ambiente AMTF

Com relação à segunda bateria de testes, ou seja, aquela que analisou a funcionalidade do ambiente AMTF, conduziu-se a interrupção da *Aplicação Paralela*, do *Monitor do Grupo de Processos*, do *Monitor Central* e do *Monitor de Réplica*

individualmente (através do comando *kill*) e testou-se suas respectivas capacidades de detecção e de reinicialização, além das funcionalidades da biblioteca proposta.

Utilizou-se o código-fonte descrito na listagem 3.1, sendo que os resultados desta bateria de testes estão representados nas listagens dos *logs* que se seguem.

4.3.1. Interrupção e Reativação do *Monitor Central*

As listagens 4.1 e 4.2 demonstram através de *logs* dos *Monitores Central* e de *Réplica* a detecção da interrupção e a reativação do *Monitor Central* por parte do *Monitor de Réplica*.

Monitor de Replica (RANK 1) sendo inicializado.

Recuperar informacoes sobre os IDs de memoria compartilhada.

Enviar heartbeat para o Monitor Central (RANK 1).

Receber heartbeat do Monitor Central (RANK 1).

...

Enviar heartbeat para o Monitor Central (RANK 1).

Receber heartbeat do Monitor Central (RANK 1).

Enviar heartbeat para o Monitor Central (RANK 1).

...

Enviar heartbeat para o Monitor Central (RANK 1).

Nao foi recebido heartbeat do Monitor Central (RANK 1). (1)

O Monitor Central (RANK 1) esta sendo reativado. (2)

Receber heartbeat do Monitor Central (RANK 1).

Enviar heartbeat para o Monitor Central (RANK 1).

...

Receber heartbeat do Monitor Central (RANK 1).

Enviar heartbeat para o Monitor Central (RANK 1).

O Monitor Central solicitou o encerramento do Monitor de Replica.

O Monitor de Replica foi encerrado.

Listagem 4.1: *Log do Monitor de Réplica (interrupção do Monitor Central).*

Na sequência do *log* da listagem 4.1 observa-se que o *Monitor Central* não enviou seus sinais de *heartbeat* (1) e isso foi interpretado como uma interrupção no funcionamento do mesmo; desta forma o *Monitor de Réplicas*, em resposta a esta

ocorrência, ativou uma nova cópia do *Monitor Central* (2).

Monitor Central (RANK 1) sendo inicializado.

Informacoes sobre as replicas do Monitor Central ja estao disponiveis.
Abrir Canal de Comunicacao para receber heartbeats das replicas do Monitor Central.
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
...
Receber heartbeat do Monitor de Replica.
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
Receber heartbeat do Monitor de Replica.

Monitor Central (RANK 1) sendo reinicializado. (1)

Abrir Canal de Comunicacao para receber heartbeats das replicas do Monitor Central.
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
Receber heartbeat do Monitor de Replica.
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
Receber heartbeat do Monitor de Replica.
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
...
O controlador AMTF solicitou o encerramento do Monitor Central.
O Monitor Central foi encerrado.

Listagem 4.2: *Log do Monitor Central (interrupção do Monitor Central)*.

No ponto de observação (1) da listagem 4.2 nota-se o momento em que a nova cópia do *Monitor Central* é ativada pelo *Monitor de Réplicas*.

4.3.2. Interrupção e Reativação do *Monitor de Réplica*

As listagens 4.3 e 4.4 mostram como ficaram os *logs* dos *Monitores Central e de Réplica* após a interrupção do *Monitor de Réplica* e a reativação do mesmo por parte do *Monitor Central*.

Monitor Central (RANK 1) sendo inicializado.

Informacoes sobre as replicas do Monitor Central ja estao disponiveis.
Abrir Canal de Comunicacao para receber heartbeats das replicas do Monitor Central.
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
Receber heartbeat do Monitor de Replica.
...
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.

Nao foi recebido heartbeat do Monitor de Replica. (1)

Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.

Nao foi recebido heartbeat do Monitor de Replica. (2)

Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.

O Monitor de Replica nao esta operacional. Reativar MR. (3)

Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.

Nao foi recebido heartbeat do Monitor de Replica. (4)

Receber heartbeat do Monitor de Replica. (5)

Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
...
Receber heartbeat do Monitor de Replica.
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
O controlador AMTF solicitou o encerramento do Monitor Central.
O Monitor Central foi encerrado.

Listagem 4.3: Log do *Monitor Central* (interrupção do *Monitor de Réplica*).

Os pontos de observação (1) e (2) presentes no *log* da listagem 4.3 demonstram que o *Monitor Central* percebeu que o *Monitor de Réplica* cessou o envio dos *heartbeats* interpretando assim como uma interrupção em seu funcionamento e a necessidade de reativá-lo (3).

No ponto (4) observa-se que o *Monitor de Réplica* ainda não entrou em funcionamento e o passo (5) demonstra que o mesmo já começa a emitir os sinais de *heartbeat*.

Monitor de Replicação (RANK 1) sendo inicializado.

Recuperar informações sobre os IDs de memória compartilhada.

Enviar heartbeat para o Monitor Central (RANK 1).

Receber heartbeat do Monitor Central (RANK 1).

...

Monitor de Replicação (RANK 1) sendo reinicializado.

(1)

Recuperar informações sobre os IDs de memória compartilhada.

Enviar heartbeat para o Monitor Central (RANK 1).

Receber heartbeat do Monitor Central (RANK 1).

...

Enviar heartbeat para o Monitor Central (RANK 1).

Receber heartbeat do Monitor Central (RANK 1).

O Monitor Central solicitou o encerramento do Monitor de Replicação.

O Monitor de Replicação foi encerrado.

Listagem 4.4: *Log do Monitor de Réplica (interrupção do Monitor de Réplica).*

Já a listagem 4.4 reflete o *log* do *Monitor de Réplica* e nele se observa o momento em que o referido módulo é reativado (1).

4.3.3. Interrupção e Reativação do *Monitor do Grupo de Processos*

As listagens 4.5 e 4.6 mostram como ficaram os *logs* dos *Monitores Central e do Grupo de Processos* após a interrupção do *Monitor do Grupo de Processos* e a reativação do mesmo por parte do *Monitor Central*.

O ponto de observação (1) presente na listagem 4.5 mostra o momento em que o *Monitor do Grupo* se registra junto ao *Monitor Central* e fica constantemente enviando *heartbeats*.

Já o ponto (2) demonstra que o *Monitor Central* percebeu o cessar dos *heartbeats* do *Monitor do Grupo de Processos* e determinado que uma cópia do mesmo seja reativada (3).

O ponto (4) mostra que a cópia foi reativada com sucesso pois os *heartbeats* voltaram a ser emitidos e o passo (5) demonstra o momento em que o *Monitor do Grupo de Processos* encerra sua execução.

Monitor Central (RANK 1) sendo inicializado.

Informacoes sobre as replicas do Monitor Central ja estao disponiveis.
Abrir Canal de Comunicacao para receber heartbeats das replicas do Monitor Central.
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
Receber heartbeat do Monitor de Replica.
...
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
Receber heartbeat do Monitor de Replica.

Receber heartbeat do Monitor de Grupo de Processos 176615. (1)

Receber heartbeat do Monitor de Grupo de Processos 176615.
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
Receber heartbeat do Monitor de Replica.
...
Receber heartbeat do Monitor de Grupo de Processos 176615.
Receber heartbeat do Monitor de Grupo de Processos 176615.
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
Receber heartbeat do Monitor de Replica.

Nao foi recebido heartbeat do Monitor de Grupo de Processos 176615. (2)

Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
Receber heartbeat do Monitor de Replica.

O Monitor de Grupo de Processos 176615 nao esta operacional. Reativar MG. (3)

Receber heartbeat do Monitor de Grupo de Processos 176615. (4)

Receber heartbeat do Monitor de Grupo de Processos 176615.
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
Receber heartbeat do Monitor de Replica.
...
Receber heartbeat do Monitor de Grupo de Processos 176615.
Receber heartbeat do Monitor de Grupo de Processos 176615.
Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.
Receber heartbeat do Monitor de Replica.

Receber pedido de encerramento do Monitor de Grupo de Processos 176615. (5)

Receber heartbeat do Monitor Central de rank 1.
Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.7.

Receber heartbeat do Monitor de Replica.

...

Receber heartbeat do Monitor Central de rank 1.

Enviar Heartbeat para o Monitor Central (RANK 1) no endereço 10.1.1.7.

Receber heartbeat do Monitor de Replica.

O controlador AMTF solicitou o encerramento do Monitor Central.

O Monitor Central foi encerrado.

Listagem 4.5: *Log do Monitor Central (interrupção do Monitor do Grupo de Processos).*

Na listagem 4.6 percebe-se o momento em que a cópia do *Monitor do Grupo de Processos* é reativada (1).

Abrir Canal de Comunicacao para Receber Hearbeats da Aplicação Paralela.

Enviar heartbeat para o Monitor Central.

...

Enviar heartbeat para o Monitor Central.

Receber pedido de autenticacao da aplicacao.

Enviar heartbeat para o Monitor Central.

...

Enviar heartbeat para o Monitor Central.

Receber pedido para tomar checkpoint da aplicacao.

Tomar checkpoint da aplicacao.

Enviar heartbeat para o Monitor Central.

Receber heartbeat da aplicacao.

Enviar heartbeat para o Monitor Central.

...

Enviar heartbeat para o Monitor Central.

Abrir Canal de Comunicacao para Receber Hearbeats da Aplicação Paralela. (1)

Enviar heartbeat para o Monitor Central.

...

Receber pedido para tomar checkpoint da aplicacao.

Tomar checkpoint da aplicacao.

Enviar heartbeat para o Monitor Central.

...

Enviar heartbeat para o Monitor Central.

Receber heartbeat da aplicacao.

Enviar heartbeat para o Monitor Central.

...

Enviar heartbeat para o Monitor Central.

Receber pedido para tomar checkpoint da aplicacao.

Tomar checkpoint da aplicacao.

Enviar heartbeat para o Monitor Central.

...

Enviar heartbeat para o Monitor Central.

Receber pedido de encerramento da aplicacao.

Enviar heartbeat para o Monitor Central.

O Monitor de Grupo de Processos foi encerrado.

Listagem 4.6: *Log do Monitor do Grupo de Processos (interrupção do Monitor do Grupo de Processos).*

4.3.4. Interrupção e Reativação da Aplicação Paralela

A listagem 4.7 mostra como ficou o *log* do *Monitor do Grupo de Processos* após a interrupção da *Aplicação Paralela* e a reativação da mesma por parte do *Monitor do Grupo de Processos*.

Na listagem 4.7, o ponto de verificação (1) demonstra o *Monitor do Grupo de Processos* recebendo *heartbeat* da aplicação e o ponto (2) o momento em que esse *heartbeat* deixa de ser enviado. Já o ponto (3) demonstra o procedimento de reativação da aplicação.

Abrir Canal de Comunicacao para Receber Hearbeats da Aplicação Paralela.
Enviar heartbeat para o Monitor Central.
Receber pedido de autenticacao da aplicacao.
Enviar heartbeat para o Monitor Central.
...
Enviar heartbeat para o Monitor Central.
Receber pedido para tomar checkpoint da aplicacao.
Tomar checkpoint da aplicacao.
Enviar heartbeat para o Monitor Central.
...
Enviar heartbeat para o Monitor Central.

Receber heartbeat da aplicação. (1)

Enviar heartbeat para o Monitor Central.
...
Enviar heartbeat para o Monitor Central.
Receber pedido para tomar checkpoint da aplicacao.
Tomar checkpoint da aplicacao.
Enviar heartbeat para o Monitor Central.
...
Enviar heartbeat para o Monitor Central.

A Aplicacao Paralela nao esta operacional. (2)

Restaurar a Aplicacao Paralela. (3)

Abrir Canal de Comunicacao para Receber Hearbeats da Aplicação Paralela.

Enviar heartbeat para o Monitor Central.

...

Enviar heartbeat para o Monitor Central.

Receber pedido para tomar checkpoint da aplicacao.

Tomar checkpoint da aplicacao.

Enviar heartbeat para o Monitor Central.

...

Enviar heartbeat para o Monitor Central.

Receber heartbeat da aplicacao.

Enviar heartbeat para o Monitor Central.

...

Enviar heartbeat para o Monitor Central.

Receber pedido de encerramento da aplicacao.

Enviar heartbeat para o Monitor Central.

O Monitor de Grupo de Processos foi encerrado.

Listagem 4.7: *Log do Monitor do Grupo de Processos (interrupção na Aplicação Paralela).*

4.3.5. Interrupção e Reativação do *Monitor Central Virtual*

As listagens 4.8 e 4.9 demonstram como ficaram os arquivos de *log* após o módulo *Monitor Central*, em execução no nó de computação NC1 ter detectado a interrupção do módulo *Monitor Central* em execução no nó NC0. As listagens demonstram o funcionamento do *Monitor Central Virtual*.

Na listagem 4.8, os pontos de verificação (1) e (3) mostram o *Monitor Central* do nó NC1 monitorando o *Monitor Central* do nó NC0 e recebendo o *heartbeat* do mesmo (2).

Monitor Central (RANK 1) sendo inicializado.

Informacoes sobre as replicas do Monitor Central ja estao disponiveis.

Abrir Canal de Comunicacao para receber heartbeats das replicas do Monitor Central.

Enviar Heartbeat para o Monitor Central (RANK 0) no endereco 10.1.1.7.

Receber heartbeat do Monitor Central de rank 1.

Rank 1 monitorando o MC(0).

(1)

Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.22.

Receber heartbeat do Monitor de Replica.

Receber heartbeat do Monitor Central de rank 0. (2)
Rank 1 monitorando o MC(0).
...
Receber heartbeat do Monitor Central de rank 1.
Rank 1 monitorando o MC(0). (3)
Enviar Heartbeat para o Monitor Central (RANK 1) no endereço 10.1.1.22.
Receber heartbeat do Monitor de Replica.
Enviar Heartbeat para o Monitor Central (RANK 0) no endereço 10.1.1.7.
Receber heartbeat do Monitor Central de rank 1.
Rank 1 monitorando o MC(0).
...
Falha no MC(0) - Endereço 10.1.1.7. Tentando reativacao. (4)
Enviar Heartbeat para o Monitor Central (RANK 1) no endereço 10.1.1.22.
Receber heartbeat do Monitor de Replica.
Receber heartbeat do Monitor Central de rank 0. (5)
Rank 1 monitorando o MC(0).
Enviar Heartbeat para o Monitor Central (RANK 0) no endereço 10.1.1.7.
Receber heartbeat do Monitor Central de rank 1.
Rank 1 monitorando o MC(0).
...
Enviar Heartbeat para o Monitor Central (RANK 0) no endereço 10.1.1.7.
Receber heartbeat do Monitor Central de rank 1.
Rank 1 monitorando o MC(0).
Enviar Heartbeat para o Monitor Central (RANK 1) no endereço 10.1.1.22.
O controlador AMTF solicitou o encerramento do Monitor Central.
O Monitor Central foi encerrado.

Listagem 4.8: *Log do Monitor Central do Nó 1 monitorando e reativando o Monitor Central do Nó 0.*

Já o ponto de verificação (4) demonstra o momento em que o *Monitor Central* do nó NC1 detectou a ausência dos *heartbeats* do *Monitor Central* do nó NC0, enviando assim um comando para o referido nó para que este reative uma nova cópia do monitor interrompido.

Finalmente, no ponto (5) o *Monitor Central* do nó NC1 volta a receber os *heartbeats* do monitor reativado, fazendo com que a execução continue sem problemas.

A listagem 4.9 demonstra a seqüência de eventos registrados pelo *Monitor Central* do nó NC0. O ponto (1) registra o momento em que este monitor envia o *heartbeat* para o seu monitor complementar no nó NC1 e o passo (2) o momento em que o monitoramento remoto é realizado.

Monitor Central (RANK 0) sendo inicializado.

Informacoes sobre as replicas do Monitor Central ja estao disponiveis.

Abrir Canal de Comunicacao para receber heartbeats das replicas do Monitor Central.

Receber heartbeat do Monitor Central de rank 0.

Rank 0 monitorando o MC(1).

Enviar Heartbeat para o Monitor Central (RANK 0) no endereco 10.1.1.7.

Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.22. (1)

Receber heartbeat do Monitor Central de rank 1.

Rank 0 monitorando o MC(1). (2)

Receber heartbeat do Monitor de Replica.

...

Enviar Heartbeat para o Monitor Central (RANK 0) no endereco 10.1.1.7.

Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.22.

Receber heartbeat do Monitor Central de rank 1.

Rank 0 monitorando o MC(1).

Receber heartbeat do Monitor de Replica.

Monitor Central (RANK 0) sendo reinicializado. (3)

Abrir Canal de Comunicacao para receber heartbeats das replicas do Monitor Central.

Receber heartbeat do Monitor Central de rank 0.

Rank 0 monitorando o MC(1).

Enviar Heartbeat para o Monitor Central (RANK 0) no endereco 10.1.1.7.

Enviar Heartbeat para o Monitor Central (RANK 1) no endereco 10.1.1.22. (4)

Receber heartbeat do Monitor Central de rank 1.

Rank 0 monitorando o MC(1).

Nao foi recebido heartbeat do Monitor de Replica.

Receber heartbeat do Monitor de Replica.

...

Rank 0 monitorando o MC(1).

Enviar Heartbeat para o Monitor Central (RANK 0) no endereco 10.1.1.7.

O controlador AMTF solicitou o encerramento do Monitor Central.

O Monitor Central foi encerrado.

Listagem 4.9: *Log do Monitor Central do Nó 0* após ser reativado pelo *Monitor Central do Nó 1*.

Já o ponto (3) registra o momento em que a nova cópia do *Monitor Central* do nó NC0 é ativada e o ponto (4) o momento em que o *heartbeat* volta a ser emitido.

4.3.6. Interrupção e Reativação do Ambiente AMTF

As listagens 4.10 e 4.11 relatam momentos quando o ambiente AMTF é reativado após uma solicitação de encerramento.

No caso da listagem 4.10, no momento em que o ambiente AMTF é reativado e o *Monitor Central* entra em execução, executa-se um procedimento de verificação de aplicações paralelas interrompidas (1). No caso desta listagem não havia aplicações pendentes (2).

ReExecutor (RANK 0) sendo inicializado.

Recuperar informacoes sobre aplicacoes interrompidas.	(1)
Não foram encontradas aplicacoes interrompidas.	(2)

ReExecutor sendo encerrado.

Listagem 4.10: *Log do Monitor Central* demonstrando o retorno sem a existência de aplicação paralela interrompida.

O mesmo não acontece na listagem 4.11, onde após o procedimento de verificação de aplicações pendentes (1) detecta-se a existência de algumas o que força uma reativação do *Monitor do Grupo de Processos* respectivo (2). Este monitor de grupo será o responsável por reativar a aplicação paralela.

ReExecutor (RANK 0) sendo inicializado.

Recuperar informacoes sobre aplicacoes interrompidas.	
Não foram encontradas aplicacoes interrompidas.	
ReExecutor sendo encerrado.	
Recuperar informacoes sobre aplicacoes interrompidas.	(1)
Reativando Monitor do Grupo de Processos 96954.	(2)

ReExecutor sendo encerrado.

Listagem 4.11: *Log do Monitor Central* demonstrando o retorno com a existência de aplicação paralela interrompida.

4.3.7. Interrupção e Reativação de um Nó do *Cluster* de Computadores

No caso de uma interrupção abrupta do *cluster* de computadores ou de um nó em particular, no retorno ao funcionamento haverá referências aos *Monitores de*

Grupos de Processos nos arquivos de controle dos *Monitores Centrais* e estes reativam os respectivos *Monitores do Grupo de Processos* para que as aplicações sejam reiniciadas a partir do último *checkpoint* registrado. Neste procedimento não se utiliza *log*.

Assim, com os resultados desta segunda bateria de testes verificou-se que o ambiente conseguiu detectar as interrupções sofridas pelas aplicações (procedimento de *failover*), reativando-as (procedimento de *failback*) e mantendo a continuidade de execução das mesmas, preservando o processamento já realizado e o tempo gasto para tal.

Todos os demais módulos de *software* do ambiente também conseguiram detectar as interrupções (procedimento de *failover*) e procederam as reativações (procedimento de *failback*) através de seus respectivos monitores com sucesso, não interrompendo o funcionamento geral do restante do ambiente e das aplicações.

Também foram realizados testes para se verificar a capacidade do ambiente de, ao receber uma solicitação de encerramento do módulo *Controlador AMTF*, interromper todas as aplicações paralelas em execução e em sua volta reativá-las, além dos testes envolvendo o *Monitor Central Virtual* e interrupções de nós específicos do *cluster*, comportando-se conforme o esperado.

Desta forma, conclui-se que os resultados de todos os testes realizados comprovaram os ciclos de vida apresentados no capítulo 3.

4.4. A biblioteca do Ambiente AMTF

Especialmente para o teste de adaptabilidade da biblioteca junto às aplicações já desenvolvidas foram recuperadas aplicações paralelas e nelas foram

adicionadas as chamadas disponibilizadas pela biblioteca do ambiente AMTF, tomando-se o cuidado para não alterar o código original.

Os testes realizados com estas aplicações modificadas foram os mesmos descritos na primeira bateria e tanto o funcionamento como as reativações transcorreram normalmente.

Um exemplo destas alterações realizadas no código-fonte das aplicações, e utilizadas nos testes, podem ser observadas na listagem 4.12.

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
#include "/amtf/bibliotecas/amtf.h" (1)

double f(a)
double a;
{
    return (4.0 / (1.0 + a*a));
}

int main(argc,argv)
int argc;
char *argv[];
{
    int done = 0, n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    double startwtime, endwtime;
    int namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);

    fprintf(stderr,"Process# %d with name %s on %d processors\n",
            myid, processor_name, numprocs);

    amtf_autenticar(argv[argc -1]); (2)

    n = 0;
    while (!done)
    {
        if (myid == 0)
        {
```

```

scanf("%d",&n);
printf("Number of intervals: %d (0 quits)\n", n);

    startwtime = MPI_Wtime();
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n == 0)
    done = 1;
else
{
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs)
    {
        x = h * ((double)i - 0.5);

        sum += f(x);
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (myid == 0)
    {
        amtf_tomar_checkpoint(); (3)
        printf("pi is approximately %.16f, Relative Error is %.16e\n",
            pi, (double)100 * (pi - PI25DT)/PI25DT);
        endwtime = MPI_Wtime();
        printf("wall clock time = %f\n",
            endwtime-startwtime);
    }
}
}

amtf_encerrar(); (5)

MPI_Finalize();

return 0;
}

```

Listagem 4.12: Código-fonte de aplicação paralela utilizando as chamadas do ambiente AMTF.

O ponto de observação (1) mostrado na listagem 4.12 demonstra o momento onde se referencia a biblioteca do ambiente AMTF e seu complemento. O ponto (2) apresenta a chamada responsável por inicializar o envio dos *heartbeats* (*amtf_autenticar()*). O ponto (3) demonstra a chamada responsável por tomar o *checkpoint* (*amtf_tomar_checkpoint()*) e o ponto (4) o momento em que a chamada que encerra o envio dos *heartbeats* é acionada (*amtf_encerrar()*).

No decorrer dos testes verificou-se uma nova funcionalidade, tanto a biblioteca desenvolvida como os controles propostos pelo ambiente AMTF podem ser estendidos para as aplicações seqüências promovendo uma pequena alteração no momento da inicialização da aplicação, trabalhando com a sintaxe apresentada abaixo:

`$ /amtf/executor cr_run programa [parâmetros do programa] pid`

A listagem 4.13 mostra um exemplo de como podem ser utilizadas as chamadas disponibilizadas pela biblioteca junto a aplicações seqüenciais.

```
#include "/amtf/bibliotecas/amtf.h" (1)

main(int argc, char *argv[]) {

int linha=0, coluna=0;

    amtf_autenticar(argv[argc-1]); (2)

    for(linha=0;linha < 10;linha++) {
        for(coluna=0;coluna < 100; coluna++){
            printf("Linha=%d Coluna=%d\n",linha,coluna);
        }
        sleep(10);

        amtf_tomar_checkpoint(); (3)
    }
    amtf_encerrar(); (4)
}
```

Listagem 4.13: Código-fonte de aplicação seqüencial utilizando as chamadas do ambiente AMTF.

Assim como ocorreu na listagem 4.12, no código da listagem 4.13 o ponto de observação (1) representa a referência à biblioteca disponibilizada pelo ambiente AMTF. O ponto (2) ilustra a chamada responsável por iniciar o envio dos *heartbeats* para o *Monitor do Grupo de Processos*, o ponto (3) demonstra a utilização da chamada responsável por solicitar a tomada dos *checkpoints* e o ponto (4) mostra a chamada responsável por encerrar o envio dos *heartbeats*.

5. CONCLUSÕES E PROPOSTAS DE TRABALHOS FUTUROS

Com o ambiente AMTF mostrou-se, através da bateria de testes realizada, ser possível tratar falhas da classe *fail stop* através de *software* em ambientes de computação distribuída, utilizando-se a técnica de *checkpoint/restart* para aplicações paralelas construídas a partir do LAM/MPI ou para aplicações seqüenciais, e o mesmo pode ser considerado como uma contribuição para ambientes distribuídos.

Os resultados dos testes aos quais o ambiente AMTF foi submetido comprovaram o comportamento planejado para o mesmo, mantendo suas ações dentro dos ciclos de vida previamente definidos.

Os objetivos planejados inicialmente foram atingidos, ou seja, a proposição de um ambiente de execução que viabilizasse a detecção e a reativação automática (procedimentos de *failover* e *failback*) da execução de aplicações paralelas, preservando-se o processamento realizado e minimizando o tempo dispendido para tal foi concretizada, sendo esta a principal contribuição deste trabalho.

Como contribuições secundárias tem-se a disponibilização do mecanismo de tomada automática de *checkpoints*, complementando a forma manual disponibilizada pelo projeto BLCR, além de uma biblioteca que permite estender estas funcionalidades de *failover* e *failback* para aplicações paralelas e também seqüenciais, sejam elas novas ou já existentes.

Ficou comprovado também, através do ambiente e dos testes a ele aplicados, que é possível adaptar o projeto BLCR e o LAM/MPI para suportarem a técnica de *checkpoint/restart* de maneira automática sem mudanças substanciais de suas

estruturas e maneiras de funcionamento.

Como efeito colateral foi detectado, em decorrência do uso do ambiente AMTF, um “atraso” no tempo de conclusão do processamento da aplicação, isso em decorrência do tempo gasto para as tomadas de *checkpoints*. Porém esse “atraso” pode ser reduzido caso se trabalhe a questão da periodicidade e a quantidade de solicitações pela tomada destes mesmos *checkpoints*. No caso dos experimentos realizados o aumento no tempo de execução foi de 25%, mas esse percentual atingiu tal valor em decorrência das tomadas dos *checkpoints*, que foram muito frequentes.

Este “atraso” pode ser ajustado através do intervalo de tempo atribuído para a tomada dos *checkpoints*, ou seja utilizando-se intervalos mais longos, ou mesmo uma quantidade menor de solicitações, obtem-se assim um menor número de tomadas de *checkpoints* e, conseqüentemente, um menor impacto no tempo final de execução.

Apesar desse ajuste na tomada de *checkpoints*, o ambiente AMTF, caso seja utilizado em ambientes geograficamente distribuídos, pode acarretar um novo atraso da execução da aplicação, isso em decorrência da latência da rede que venha a ser utilizada para a comunicação entre os módulos e entre as aplicações paralelas ou mesmo para a gravação dos arquivos de *log* e *checkpoints*; desta maneira, o ambiente AMTF não é recomendado para situações que estejam muito dispersos.

Sobre o uso da rede e do sistema de arquivos, durante os testes não se notou um impacto que viesse a comprometer tais recursos. Eventualmente, o uso destes recursos pode, inclusive, ser minimizado reduzindo-se a quantidade ou tamanho das mensagens a serem registradas no arquivo de *log* dos monitores ou então criando-se um subsistema de comunicação alternativo, destinado apenas para uso por parte dos *heartbeats* e das mensagens de *log*.

Como propostas de continuidade deste trabalho, são apresentadas algumas

sugestões que poderiam contribuir com o trabalho original, incorporando novas funcionalidades ou até mesmo melhorando sua estrutura de implementação.

Uma questão importante quanto a estrutura do ambiente e que poderia ser viabilizada em trabalhos futuros seria a questão da otimização do código-fonte do ambiente AMTF, visto que a preocupação maior, presente durante o desenvolvimento do ambiente, era a de testar sua funcionalidade; desta maneira foram produzidos códigos relativamente simples, sem que fossem explorados recursos adicionais fornecidos pela linguagem C ou C++ ou uma lógica de programação mais eficiente e apurada.

Conforme mencionado, o trabalho desenvolvido abordou apenas falhas da classe *fail stop*, assim, como uma possível continuidade do trabalho poderiam ser implementadas controles adicionais que tratassem outras classes de falhas, como as falhas da classe bizantina, por exemplo.

Uma outra funcionalidade que poderia ser explorada futuramente seria a do ambiente auto-ajustar o intervalo de tomada dos *checkpoints* conforme o comportamento e a carga presente no ambiente computacional, além das características do programa, ou seja, tomá-los quando fosse o momento mais apropriado.

Nesta mesma linha, ou seja, do melhor momento para a tomada dos *checkpoints*, uma funcionalidade que também poderia ser explorada em trabalhos futuros seria a observação do melhor momento para o registro tomando-se como referência a utilização da memória pelo processo, visto que, com a menor ocupação da memória seriam menores os arquivos de contextos e, conseqüentemente, menor a utilização da rede no momento de armazená-lo.

Do ponto de vista de um melhor monitoramento e facilidade de leitura e interpretação, poderia ser agregada ao ambiente uma interface mais amigável para se observar o funcionamento das aplicações, dos *Monitores Centrais*, de *Réplicas* e

dos Grupos de Processos e de se recuperar os *logs* dos mesmos; provavelmente uma interface que permitisse interação via um navegador *Web* e através da Internet.

Um problema detectado ao longo do desenvolvimento e uso do ambiente foi, como forçar a interrupção das aplicações paralelas, assim, como mais uma sugestão de continuidade, poderia ser desenvolvido um comando *kill* especial para que o usuário possa encerrar suas aplicações sem que fosse interpelado pelo *Monitor do Grupo de Processos*, visto que o comando *kill* tradicional irá encerrar os processos paralelos mas o *Monitor do Grupo* da aplicação em questão irá reiniciá-la.

E, como uma última sugestão de continuidade, tem-se a necessidade de, efetivamente, testar o ambiente AMTF em situações de uso real, com um grande volume de nós de computação, de usuários e de aplicações paralelas e com um uso intensivo do ambiente, verificando-se assim o comportamento do mesmo frente a um volume considerável de execuções e requisições por procedimentos de *failover* e *failback*, além de analisar com mais detalhes o impacto produzido por este ambiente na rede, na capacidade das CPUs e no sistema de arquivos do *cluster* de computadores.

Estes testes, realizados nestas situações reais, podem contribuir para se determinar a escalabilidade tanto do ambiente AMTF como do próprio *cluster* de computadores e das aplicações a serem processadas.

REFERÊNCIAS BIBLIOGRÁFICAS

- [ACNC 2004] ACNC 2004. Advanced Computer and Network Corporation - http://www.acnc.com/04_00.html, (Dez. 2004).
- [AFS 2003] AFS 2003. *Internet Faq Consortium* - <http://nwww.faqs.org/faqs/afs-faq>, (Jun. 2003).
- [BERSTIS 2005] BERSTIS Viktors 2005. *Fundamentals of Grid Computing* - <http://www.redbooks.ibm.com/redpapers/pdfs/redp3613.pdf>, (Ago. 2005).
- [BIRMAN 1995] BIRMAN, Kenneth P. 1995. *Building Secure and Reliable Network Applications*, - <http://www.cs.cornell.edu/ken/book.pdf>, (Fev. 2003).
- [BLCR 2004] BLCR 2004, *Berkeley Lab Checkpoint/Restart* - <http://ftg.lbl.gov/twiki/bin/view/FTG/CheckpointRestart>, (Fev. 2004).
- [BLCR 2005] BLCR 2005, *Using Checkpoint/Restart: A Tutorial* - <http://ftg.lbl.gov/twiki/bin/view/Whiteboard/UsingCheckpointRestart>, (Mar. 2005).
- [BRAAM 1999] BRAAM, Peter J. 1999 - *File Systems for Cluster from a Protocol Perspective* -

<http://www.cs.cmu.edu/afs/cs/project/coda/Web/docdir/extremelinux99.pdf> , (Jun. 2003).

- [CHANDY 1985] CHANDY, M. e LAMPORT, Leslie 1985 - *Distributed Snapshots: Determining Global States of Distributed Systems*. ACM Transactions on Computing Systems, vol. 3,1, 63-75.
- [CODA 2003] CODA 2003 - *CODA File System* .
<http://www.coda.cs.cmu.edu/doc/html/coda-howto-1.html>, (Jun. 2003).
- [CRISTIAN 1991] CRISTIAN, Flavin 1991. *Understanding Fault-Tolerant Distributed Systems* - Communications of ACM, vol. 34, 2 (Fev. 1991), 56-78.
- [CRISTIAN 1996] CRISTIAN, Flavin 1996. *Synchronous and Asynchronous Group Communication* - Communications of ACM, vol. 39, 4 (Abr. 1996), 88-97.
- [DEITEL 2005] DEITEL, Harvey M. e DEITEL, Paul J. e CHOFFNES, David R. *Sistemas Operacionais – Terceira Edição* - Editora Pearson Prentice Hall, 2005.
- [DIETER 2004] DIETER, William R. e LUMPP, James E. 2004. *User-Level Checkpointing for Linux Threads Programs* -
<http://www.engr.uky.edu/~dieter/pub/dieter-jel-usenix-2001.pdf>, (Mai. 2004).
- [DOW 2002] DOW, Chyi-Ren e CHEN, Jong-Shin e HSIEH, Min-Chang 2002. *Checkpointing MPI applications on*

- symmetric multi-processor machines using SMPckpt* - The Journal of Systems and Software - vol. 63 (2002), 137-150.
- [DOSS 2001] DOSS, David e YURCIK, William 2001. *Achieving Fault-Tolerant Software with Rejuvenation and Reconfiguration* – IEEE Software, (Jul./Aug. 2001), 48-52.
- [DUELL 2004] DUELL, Jason 2004. *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart* - <http://crd.lbl.gov/~jcduell/papers/blcr.pdf>, (Fev. 2004).
- [EMMERICH 2005] EMMERICH Wolfgang 2005. *Distributed Systems Principles* - <http://www.cs.ucl.ac.uk/staff/W.Emmerich/lectures/ds98-99/dsee3.pdf>, (Abr. 2005).
- [FAQ 2003] FAQ 2003 . *AFS Frequently Asked Questions* - <http://www.angelfire.com/hi/plutonic/afs-faq.html#Contents>, (Jun. 2003).
- [GÄRTNER 1999] GÄRTNER, Felix C. 1999. *Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments* - ACM Computing Surveys, vol. 31, 1 (Mar. 1999), 1-26.
- [IBM 2003] IBM 2003. International Business Machines Corporation (IBM) - *AFS User Guide* - <http://oss.software.ibm.com/developerworks/opensource/afs/docs/PDF/UserGd/auusg000.pdf>, (Jun. 2003).

- [JOHNSON 1989] JOHNSON, David Bruce 1989. *Distributed System Fault Tolerance Using Message Logging and Checkpointing* – Ph.D thesis, Rice University, (Dez. 1989), Houston , Texas.
- [JOHNSON 2002] JOHNSON, David Bruce, ALVISI, Lorenzo *et all* 2002. *A survey of rollback-recovery protocols in message-passing systems* – ACM Computing Surveys, vol. 34, 3 (Set. 2002), 375-408.
- [KIM 2002] KIM, K. H. 2002. *Fault-Tolerant Distributed Computing: Evolution and Issues* – [dsonline.computer.org/0207/features/kim_print,htm](http://dsonline.computer.org/0207/features/kim_print.htm), (Jul. 2002).
- [KIRNER 1988] KIRNER, Claudio e MENDES, Sueli B. T. 1988. *Sistemas Operacionais Distribuídos* - Editora Campus, 1988.
- [KRISTLER 1990] KRISTLER, James J. 1990. *Increasing File System Availability Through Second-Class Replication* - Proceedings of the IEEE Workshop on Management of Replicated Data, (Nov. 1990), Houston, Texas.
- [LAMPORT 1982] LAMPORT, Leslie e SHOSTAK, Robert e PEASE, Marshall 1982. *The Byzantine Generals Problem* – ACM Transactions on Programming Language and Systems, vol. 4, 3 (Jul. 1982), 382-401.
- [LAM/MPI 2005] LAM/MPI 2005. *LAM/MPI Paralell Computing* - <http://www.lam-mpi.org>, (Ago. 2005).

- [LITZKOW 2004] LITZKOW, Michael e TANNENBAUM, Todd et. all. 2004. *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System* - <http://www.cd.wisc.edu/condor/doc/ckpt97.ps>, (Mai. 2004).
- [LVS 2005] Linux Virtual Server 2005. *High Availability* - <http://www.linuxvirtualserver.org/HighAvailability.html>, (Mar. 2005).
- [LUMPP 1998] LUMPP, James E. e DIETER, William R. 1998. *Checkpointing with Multicast Communication* – Proceedings of the 1998 IEEE Aerospace Conference, Mar. 1998.
- [MAIER 1993] MAIER, Joachim 1993. *Fault Tolerance Applied to Parallel Computing* – Proceedings of the Compcon Springs'93, IEEE Computer Society Press (1993), 244-252.
- [MASSIE 2005] MASSIE, Matthew L. e CHUN, Brent N. e CULLER, David E. – *The Ganglia distributed monitoring system: design, implementation, and experience* – <http://www.cs.berkeley.edu/~massie/papers/science.pdf>, (Fev. 2005).
- [MOGILEVSKY 2005] MOGILEVSKY, Dmitry e KELLER, Sean 2005. *SafeMPI – Extending MPI for Byzantine Error Detection on Parallel Clusters* - http://arxiv.org/PS_cache/cs/pdf/0506/0506001.pdf, (Jun. 2005).

- [NCSA 2003] NCSA 2003. *National Center for Supercomputing Applications (NCSA)* - <http://archive.ncsa.uiuc.edu/Pubs/UserGuides/AFSGuide/AFSv2.1Book.html> , (Jun. 2003).
- [PARK 2000] PARK, Taesoon e YEON, Heon Y. 2000. *Application controlled checkpointing coordination for fault-tolerant distributed computing systems* – *Parallel Computing*, vol. 26, (2000), 467-482.
- [PINGALI 2003] PINGALI, Keshav e BRONEVETSKY, Greg e STODGHILL, Paul et. al. 2003. *Automated Application-level Checkpointing of MPI Programs* - http://www.cs.cornell.edu/Nuprl/PRLSeminar/PRLSeminar02_03/Pingali/automated.pdf, (Mar. 2003).
- [PLANK 2004] PLANK, James S. e BECK, Micah et. al. 2004. *Libckpt: Transparent Checkpointing under UNIX* - <http://www.cs.utk.edu/~plank/plank/papers/USENIX-95W.html>, (Mai. 2004).
- [PRUYNE 2004] PRUYNE, Jim e LIVNY, Miron 2004. *Managing Checkpoints for Parallel Programs* - <http://www.cs.wisc.edu/condor/doc/ckpt-mgmt.ps>. (Mai. 2004).
- [RAID 2003] RAID 2003. *Advanced Computer & Network Corporation* - http://www.acnc.com/04_01_00.html (Jun. 2003).
- [RAYNAL 2001] RAYNAL, Michel e SINGHAL, Mukesh 2001. *Mastering Agreement Problems in Distributed*

Systems – IEEE Software, (Jul./Aug. 2001), 48-52.

- [RESNICK 2003] RESNICK, Ron I. 2003. *A Modern Taxonomy of High Availability* -
http://www.csse.monash.edu.au/projects/MDR/testdata/cat5200/99s2/1st_assignment/anggoro-12710660/HA.htm, (Mar. 2003).
- [ROBERTSON 2003] ROBERTSON, Alan 2003. *Linux-HA Heartbeat System Design* – <http://linux-ha.org/comm/Hbdesign.pdf>, (Fev. 2003).
- [ROMAN 2005] ROMAN, Eric 2005. A Survey of Checkpoint/Restart Implementations -
<http://ftg.lbl.gov/twiki/pub/Whiteboard/CheckpointPapers/checkpointSurvey-020724b.pdf>, (Mar. 2005).
- [SANKARAN 2004] SANKARAN, Sriram, SQUYRES, Jeffrey M. *et al* 2004. *The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing* -
<http://ftg.lbl.gov/twiki/pub/Whiteboard/CheckpointPapers/lacsi-2003.pdf>, (Fev. 2004).
- [SATYANARAYANAN 1989] SATYANARAYANAN, M. 1989. *CODA: A Highly Available File System for a Distributed Workstation Environment* – Proceedings of the Second IEEE Workshop on Workstation Operating Systems, (Set. 1989), Pacific Grove, California.
- [SATYANARAYANAN 1990] SATYANARAYANAN, M. 1990. *Scalable, Secure and Highly Available Distributed File Access* – IEEE Computer, vol. 23, 5 (Mai. 1990), 9-21.

- [SCHNEIDER 1983] SCHNEIDER, Fred. B e SCHLICHTING, Richard D. 1983. *Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems* – ACM Transactions on Computing Systems, vol. 1, 3 (Ago. 1983), 222-238.
- [SCHNEIDER 1990] SCHNEIDER, Fred B. 1990. *Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial* – ACM Computing Surveys, vol. 22, 4 (Dez. 1990), 299-319.
- [SILVA 1998] SILVA, Luis Moura e SILVA, João Gabriel 1998. *System-Level versus User-Defined Checkpointing* - <http://www.cs.cornell.edu/Courses/cs717/2001fa/papers/LSilvaEDCC3.pdf> , (Out. 2003).
- [STALLINGS 1999] STALLINGS, William 1999. *Computer Organization and Architecture* - Fifth Edition, Prentice Hall, 1999.
- [TAKAHASHI 2004] TAKAHASHI, Toshiyuki e SUMIMOTO, Shinji et. all. 2004. *PM2: High Performance Communication Middleware for Heterogeneous Network Environments* - <http://www.sc2000.org/techpaper/papers/pap-pap205.pdf>, (Mai. 2004).
- [TANENBAUM 2003] TANENBAUM, Andrew S. 2003. *Sistemas Operacionais Modernos* – Segunda Edição, Prentice Hall, 2003.
- [VAIDYA 1999] VAIDYA, Nitin H. 1999. *Staggered Consistent Checkpointing* – IEEE Transactions on Parallel and

Distributed Systems, vol. 10, 7 (Jul. 1999), 694-702.

[WANG 1995]

WANG, Yi-Min et al 1995. *Checkpointing and Its Applications* - Proc. IEEE Fault-Tolerant Computing Symp., (Jun. 1995), 22-31.

GLOSSÁRIO

ACCESS CONTROL LIST: Lista de Controle de Acesso, ou seja, uma maneira de se implementar um método de segurança baseado em uma lista.

AMNESIA CRASH FAILURE: sub-classe da classe de tolerância a falhas *crash failure*, ela ocorre quando um componente reinicia em um estado inicial pré-definido e que não depende de entradas recebidas antes da queda.

ANDREW FILE SYSTEM: também conhecido como AFS, é um sistema de arquivos distribuídos que permite aos componentes cooperarem para compartilhar de maneira eficiente os recursos de seus sistemas de arquivos.

BIZANTINO: classe de tolerância a falhas, nela o funcionamento do componente torna-se instável, imprevisível, de maneira errônea ou não, ou então maliciosa.

BLCR: denominação para o projeto *Berkeley Lab's Linux Checkpoint/Restart*, o BLCR permite que programas sendo executados no Linux possam ter seus estados armazenados em arquivos e, conseqüentemente, tenham seus estados recuperados no caso de alguma interrupção dos mesmos.

CHECKPOINT: Arquivo que possui o contexto de um processo.

CHECKPOINTING: Técnica utilizada para se registrar o contexto de um processo em um arquivo.

CHECKPOINT/RESTART: Técnica que utiliza o armazenamento do contexto de um processo em arquivo e a restauração do mesmo a partir destas informações armazenadas neste mesmo arquivo.

CLOSE SYNCHRONIZATION: componentes de *hardware* ou *software* estão diretamente sincronizados uns com os outros, executando o mesmo serviço em paralelo e efetuando a mesma seqüência de transições de estado.

CLUSTER: arranjo computacional caracterizado pelo conjunto de nós que se comunicam sobre uma rede de alta largura de banda e baixa latência. Frequentemente os nós são homogêneos tanto no *hardware* como no sistema operacional, a estrutura de comunicação raramente apresenta falhas e, tradicionalmente, o sistema é gerenciado a partir de uma única entidade administrativa.

CODA FILE SYSTEM: evolução do AFS, é também um sistema de arquivos distribuídos que permite aos componentes cooperarem para compartilhar de maneira eficiente os recursos de seus sistemas de arquivos.

COMPRESSED CHECKPOINTING: funcionalidade que consiste em compactar os arquivos de *checkpoint*.

CRASH FAILURE: classe de tolerância a falhas, nela o componente interrompe sua execução em algum ponto do processamento por um certo tempo, mas volta à execução, com se ocorresse uma pausa no processamento.

FAIL STOP: classe de tolerância a falhas, nela o componente apresenta uma falha e seu funcionamento é prontamente interrompido, não voltando mais a funcionar, sendo que isto é detectado pelos demais componentes.

FAILBACK: Procedimento de reativação de um componente de *hardware* ou *software*.

FAILOVER: Procedimento de detecção de uma falha ou falta em um componente de *hardware* ou *software*.

FAILURE: Falha, ou interrupção ocorrida no nível intermediário de abstração, ou seja, no nível da informação.

FAULT: Falta, ou interrupção ocorrida no nível inferior de abstração, ou seja, no nível do hardware.

FORKED CHECKPOINTING: esta técnica de *checkpointing* utiliza uma instrução para criar um processo filho encarregado de tomar os *checkpoints*, gravando-os sempre nos mesmos arquivos.

HEARTBEAT: técnica que emprega o envio e o recebimento de sinais para se monitorar o funcionamento ou não de equipamentos ou serviços.

INCREMENTAL CHECKPOINTING: esta técnica de *checkpointing* utiliza um arquivo de *checkpoint* principal onde armazena os estados iniciais e alguns arquivos secundários para armazenar apenas as mudanças que ocorrerem.

LIVENESS: Característica de um ambiente ou aplicação de manter-se operacional, ou em funcionamento, mesmo na presença de faltas ou falhas.

LOSE SYNCHRONIZATION: realiza-se uma classificação dos membros do grupo de acordo com suas sincronizações com o estado mais atual do serviço em questão.

MAIN MEMORY CHECKPOINTING: esta técnica de *checkpoint* consiste em realizar uma cópia do espaço de dados do programa no arquivo de *checkpoint*.

MTTF: *Mean Time To Failure*, tempo médio até um componente de *hardware* ou *software* apresentar uma falha.

MTTR: *Mean Time To Repair*, tempo médio para reparo de um componente de

hardware ou software.

NFS: *Network File System*, é um sistema que permite o compartilhamento do sistema de arquivos entre vários computadores interligados localmente ou remotamente.

NIS: *Network Information System*, é um sistema de informações sobre a rede que provê serviços para os equipamentos interligados, com exemplo destes serviços pode ser citado o serviço de autenticação de usuário.

OFF THE SHELF: Computadores conhecidos como “de prateleira”, ou seja, fáceis de encontrar.

PARTIAL-AMNESIA CRASH FAILURE: sub-classe da classe de tolerância a falhas *crash failure*, ela ocorre quando, durante o reinício da execução, parte do estado do componente é o mesmo de antes da queda e uma outra parte é iniciado para um estado inicial pré-definido.

PAUSE-CRASH: sub-classe da classe de tolerância a falhas *crash failure*, ela ocorre quando o componente reinicia sua execução no estado que possuía antes da queda.

RAID: *Redundand Array for Independent Disk*, Conjunto de discos replicados e/ou independentes que viabilizam formas de se atingir a tolerância a falhas em sistemas de arquivos.

REPLICAÇÃO DE PRIMEIRA CLASSE: técnica para sistema de arquivos tolerante a falhas que consiste em replicar o servidor como um todo e, conseqüentemente, o sistema de arquivos também.

REPLICAÇÃO DE SEGUNDA CLASSE: técnica para sistema de arquivos tolerante a falhas que consiste em replicar o sistema de arquivos através de vários

servidores, empregando para isso sistemas de *chaching* ou *journal file system*.

SAFETY: Característica de um ambiente ou aplicação que garante a veracidade dos resultados, mesmo na presença de faltas ou falhas.

SINGLE REPLACEABLE UNIT: Unidade Simples Reposicionável, ou então unidade simples de reposição ou de substituição.

SINGLE SYSTEM IMAGE: Imagem única do sistema, maneira se enxergar ou interagir com o sistema.

SHUTDOWN: Operação de desligamento do sistema.