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

RAFAEL DAL ZOTTO

Proposta de Mecanismo de *Checkpoint* com Armazenamento de Contexto em Memória para Ambientes de Computação Voluntária

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

Prof. Dr. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, Setembro de 2010

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Dal Zotto, Rafael

Proposta de Mecanismo de *Checkpoint* com Armazenamento de Contexto em Memória para Ambientes de Computação Voluntária / Rafael Dal Zotto. – Porto Alegre: PPGC da UFRGS, 2010.

134 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2010. Orientador: Cláudio Fernando Resin Geyer.

1. Computação voluntária. 2. Mecanismos para *Checkpoint*. 3. Alto Desempenho. 4. Prevalência de Objetos. I. Geyer, Cláudio Fernando Resin. II. Título.

“An education isn’t how much you have committed to memory, or even how much you know. It’s being able to differentiate between what you do know and what you don’t.”

— ANATOLE FRANCE

AGRADECIMENTOS

Nesta etapa final de minha escrita, gostaria de registrar minha gratidão a todos aqueles que participaram direta ou indiretamente dessa caminhada. Muito obrigado pelas conversas, apoio e incentivo oferecidos ao longo dessa jornada.

Foi Bertrand Russell quem disse que os nossos pais amam-nos porque somos seus filhos, é um fato inalterável. Nos momentos de sucesso, isso pode parecer irrelevante, mas nas ocasiões de dificuldade, oferecem um consolo e uma segurança que não se encontram em qualquer outro lugar. Muito obrigado aos meus pais, **Livino** e **Vera**, pelo exemplo, educação, amor e apoio incondicional. Obrigado por serem esse porto seguro na minha vida. Amo vocês.

À minha amada namorada, **Daiane**. Obrigado por estar ao meu lado nos momentos em que as dificuldades pareciam maiores do que realmente eram. Tuas palavras de incentivo, teu abraço carinhoso e o teu lindo sorriso foram combustíveis para que eu pudesse seguir em frente. Esse é o primeiro passo de uma longa caminhada que vamos trilhar juntos. Eu te amo.

Agradeço ao meu orientador, **Cláudio Geyer**, pela confiança depositada em mim. Sua generosidade, comprometimento e disponibilidade foram decisivos para a produção desse trabalho. Sem o seu direcionamento e suas sugestões precisas, essa pesquisa não teria sido possível. Muito obrigado por tudo.

Obrigado aos meus amigos, pelo incentivo durante a produção desse trabalho. Agradeço pelo interesse, pela ajuda e pelos questionamentos - eles foram fundamentais nesse processo. Obrigado também por investirem seu tempo lendo, revisando e comentando partes dessa dissertação. Obrigado.

Agradeço também à **Hewlett-Packard** por incentivar os meus estudos. Obrigado pela flexibilidade que permitiu que pudesse freqüentar as aulas e participar das reuniões do grupo de pesquisa. Muito mais que uma oportunidade para crescimento profissional, encontrei na HP um ambiente que permite o desenvolvimento pessoal.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
BOINC	Berkeley Infrastructure for Network Computing
BoT	Bag-of-Tasks
BLCR	Berkeley Lab Checkpoint/Restart
CIL	Common Intermediate Language
CORBA	Common Object Request Broker Architecture
CLR	Common Language Runtime
CPU	Central Processing Unit
GC	Global Computing
GPL	General Public License
HTTP	Hypertext Transfer Protocol
HPC	High-Performance Computing
IP	Internet Protocol
JNI	Java Native Interface
JVM	Java Virtual Machine
JRE	Java Runtime Environment
LAN	Local Area Network
LINQ	Language-Integrated Query
MFP	Multi Function Product/ Printer/ Peripheral
MHz	Megahertz
MPI	Message Passing Interface
MSIL	Microsoft Intermediate Language
MTBF	Mean Time Between Failures
NAT	Network Address Translation
NVRAM	Non-Volatile RAM
PDA	Personal Digital Assistant

P2P	Peer-to-Peer
PC	Personal Computer
PID	Process Identifier
PVM	Parallel Virtual Machine
PWD	Piecewise Deterministic Assumption
RAID	Redundant Array of Inexpensive Disk
RAM	Random Access Memory
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SGI	Silicon Graphics, Inc.
SNMP	Simple Network Management Protocol
SMP	Symmetric Multi-Processing
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TCS	Terascale Computing System
UDP	User-defined protocol
XML	eXtensible Markup Language
XOR	Disjunção Exclusiva
XW	XtremWeb

LISTA DE FIGURAS

LISTINGS

RESUMO

Computação voluntária é um tipo de computação distribuída na qual o proprietário do computador cede parte dos seus recursos computacionais, tais como poder de processamento ou armazenamento, para a execução de um ou mais projetos de pesquisa de seu interesse. Na área de processamento de alto desempenho, o modelo de computação voluntária desempenha um papel muito importante. Sistemas de computação voluntária de larga escala provaram ser mecanismos eficientes para resolução de problemas complexos. Em tais sistemas, que são essencialmente centralizados, centenas ou milhares de computadores são organizados em rede para processar uma série de tarefas, encaminhadas e distribuídas por um servidor central.

Nesse tipo de solução, é imprescindível ter um mecanismo para a persistência dos resultados intermediários produzidos, de maneira periódica, para evitar a perda de informações em caso de falhas. Esse mecanismo, chamado de *checkpoint*, também é importante, em ambientes de computação voluntária, para garantir que no momento em que o proprietário do recurso retomar sua utilização, os resultados intermediários produzidos sejam armazenados para uma posterior recuperação. Sem um mecanismo de *checkpoint* consistente, resultados produzidos pelos nodos de computação voluntária podem ser perdidos, gerando um desperdício do poder de computação.

A pesquisa contemplada nessa dissertação tem por objetivo propor um mecanismo de *checkpoint* baseado no armazenamento do contexto de execução, através da prevalência de objetos. Essa abordagem proporciona a participação, em sistemas de computação voluntária, de recursos com capacidades limitadas de processamento, memória e espaço em disco que possuam curtos, porém frequentes, períodos de inatividade. Dessa forma, esses recursos poderão realizar *checkpoints* rápidos e frequentes, produzindo resultados efetivos.

Palavras-chave: Computação voluntária, Mecanismos para *Checkpoint*, Alto Desempenho, Prevalência de Objetos.

A Proposal for a Checkpoint Mechanism based on Memory Execution-Context Storage for Volunteer Computing Environments

ABSTRACT

Volunteer computing is a type of distributed computing in which resource owners donate their computing resources, such as processing power and storage, to one or more projects of interest. In the high-performance computing field, the volunteer computing model has been playing an important role. On current volunteer computing systems, which are essentially center-based, hundreds or thousands of computers are organized in a network to process a series of tasks, originally distributed by a centralized server.

For this kind of environment, it is essential to have a mechanism to ensure that all intermediate produced results are stored, avoiding the loss of already processed data in case of failures. This mechanism, known as checkpoint, is also important in volunteer computing environments to ensure that when the resource owner takes control of the activities, all intermediate results are saved for later recovery. Without a consistent checkpoint mechanism, already produced data could be lost, leading to waste of computing power.

The research done on this dissertation aims mainly at introducing a checkpoint mechanism based on context execution storage, through object prevalence. On it, resources which usually have limited processing power, memory and storage and with small but frequent periods of inactivity could be allowed to join volunteer computing environments. This is possible because they would be able to execute fast and frequent checkpoint operations in short period of times and therefore, be able to effectively produce results during its inactivity periods.

Keywords: Volunteer computing, checkpoint, high-performance, object prevalence.

1 INTRODUÇÃO

O acesso a grandes quantidades de poder computacional tem sido, por diversas décadas, um dos objetivos centrais de muitos cientistas da computação. Desde a década de 60, visões sobre componentes e ferramentas computacionais, sejam elas simples ou mais complexas, têm guiado os esforços de usuários e engenheiros de sistemas (ORGANICK, 1972).

Na década de 70, iniciaram-se as idéias cujos pensamentos centrais estavam focados no objetivo de atingir esse poder computacional. A idéia se apresentava através da coleção e utilização de diversos recursos pequenos e baratos em detrimento de poucos e caros supercomputadores (STERLING et al., 1995). A partir desse pensamento, o interesse em esquemas para o gerenciamento desses processadores distribuídos se tornou cada vez mais popular (THAIN et al., 2005).

Uma das abordagens para resolução de problemas de maneira distribuída, é através da utilização de recursos computacionais cedidos pelos proprietários, através de uma solução de computação voluntária. O termo computação voluntária define um modelo computacional que provou sua capacidade de agrupar recursos distintos e oferecer ambientes para processamento de alto desempenho (P.ANDERSON; FEDAK, 2006). A idéia por trás da computação voluntária, é eliminar passos complexos de configuração de ambientes, fazendo com que a participação de qualquer usuário, independente do conhecimento técnico que possua, seja facilitada.

Como o próprio nome define, a disponibilidade de execução em ambientes de computação voluntária existe apenas enquanto o usuário proprietário está cedendo o seu recurso. Isso significa que a execução de um projeto, dentro desse tipo de ambiente, está sujeita a inúmeras interrupções que não podem ser determinadas com antecedência. Dessa forma, uma das preocupações que devem ser endereçadas nesse tipo de ambiente, é o armazenamento dos resultados intermediários produzidos, para que o processamento não seja perdido por completo a cada interrupção de execução. Para esse fim, as principais plataformas de computação voluntária oferecem mecanismos de *checkpoint* que podem ou não serem utilizados, de acordo com as necessidades do projeto.

Dentro das principais plataformas para computação voluntária, o XtremWeb deve ser mencionado. Trata-se de um software de código aberto, desenvolvido pela universidade de Paris Süd, para construir *desktop grids* através da utilização de recursos disponíveis dos computadores pessoais. A plataforma XtremWeb permite transformar um conjunto de recursos voláteis, distribuídos através de uma LAN ou mesmo da internet, em um ambiente integrado para execução de aplicações paralelas. Embora cumpra com os objetivos de agregar recursos dispersos e transformá-los em participantes de um projeto voluntário, os resultados intermediários produzidos dentro desse modelo não são armazenados. Isso significa que sempre que o proprietário de um recurso solicitar a retomada de execu-

ção, toda e qualquer informação produzida é perdida, gerando um desperdício de poder computacional.

Quando se fala em ambientes de computação voluntária, a imagem imediatamente associada é a da utilização de *desktops* pessoais que, quando fora de utilização por parte dos proprietários, podem participar de maneira produtiva de um ambiente voluntário. A associação é válida e correta pois é exatamente esse o tipo de recurso predominante no ambiente voluntário. Entretanto, a participação em um ambiente desse tipo não é exclusiva de computadores pessoais, de modo que qualquer recurso com alguma capacidade de processamento, pode participar. Tomando essa premissa como verdadeira, é correto afirmar que essa abrangência inclui a possibilidade de, em teoria, utilizar recursos com capacidades limitadas de processamento e memória como PDAs ou até mesmo impressoras.

É exatamente no contexto de utilizar impressoras multifuncionais como recursos em ambientes de computação voluntária, que a pesquisa apresentada nessa dissertação se insere. Uma impressora multifuncional pode ser definida como uma máquina que incorpora a funcionalidade de múltiplos equipamentos em apenas um, oferecendo recursos e funcionalidades de impressora, scanner, foto-copiadora, fax e e-mail. Em termos de *hardware*, as multifuncionais modernas apresentam configurações semelhantes a *desktops*, o que possibilitaria a sua utilização em ambientes de computação voluntária. Entretanto, esse tipo de recurso não é utilizado principalmente pelo fato de apresentarem curtos, porém freqüentes, períodos de disponibilidade. Isso significa que, mesmo que uma multifuncional fosse incluída em um ambiente de computação voluntária, ela dificilmente ficaria disponível tempo suficiente para processar resultados válidos.

O primeiro aspecto para possibilitar a utilização de multifuncionais em ambientes de computação voluntária, é disponibilizar um mecanismo de *checkpoint* rápido e eficiente. A pesquisa realizada ao longo do mestrado e apresentada nessa dissertação, propõe a criação de um mecanismo de *checkpoint* que possa ser realizado de maneira freqüente e que não acarrete *overhead* em termos de tempo de execução ao processamento normal do recurso. Além dessas premissas, a solução proposta deve estar adaptada às limitações, em termos de memória e capacidade de armazenamento, do recurso ao qual estará vinculada.

O modelo criado a partir dessas restrições faz uso do conceito de prevalência de objetos e *diskless checkpoint*, combinado com persistência eventual em disco, para armazenamento das informações intermediárias produzidas. As aplicações executadas dentro do modelo de *checkpoint* proposto são encapsuladas dentro de transações de prevalência, possibilitando que as informações intermediárias produzidas possam ser armazenadas e, em caso de falhas ou interrupções, recuperadas. Durante esse processo, a memória é o recurso utilizado para o armazenamento, para oferecer uma solução mais robusta para armazenamento das informações. Em contrapartida, o disco físico pode ser eventualmente utilizado para armazenar o estado geral da aplicação.

1.1 Objetivos e Contribuição

Envolvido nesse contexto, o objetivo principal dessa pesquisa consiste em propor um mecanismo de *checkpoint* rápido em memória, através da utilização de prevalência de objetos, para recursos com capacidades limitadas de memória e processamento. Tipicamente, dentro dessa classificação, estão incluídos recursos com curtos, porém freqüentes, períodos de disponibilidade para o processamento voluntário.

Dessa forma, o mecanismo de *checkpoint* em questão foi projetado, implementado e

avaliado dentro de um contexto de execução que simula as configurações de *hardware* e sistema operacional de uma multifuncional moderna. Além do projeto e implementação do mecanismo de *checkpoint*, foram realizados estudos detalhados da plataforma para computação voluntária XtremWeb, para a qual foram projetados os ajustes necessários para a utilização do modelo.

Como objeto final do estudo, tem-se um mecanismo de *checkpoint* implementado, testado e avaliado em um ambiente de execução muito próximo do qual se propõe ser aplicado. Além disso, tem-se o projeto para utilização desse novo mecanismo de *checkpoint* dentro da plataforma XtremWeb, que por padrão não possui nenhum mecanismo para armazenamento de resultados intermediários produzidos. Isso significa que essa proposta permitirá a realização de *checkpoints* de maneira rápida e freqüente ao *worker* do XtremWeb.

Como resultado da possibilidade de se realizar operações de *checkpoint* freqüentes e de maneira rápida, recursos computacionais que até então eram descartados em ambientes de computação voluntária, passam a ser opções válidas. Dessa forma, o ambiente voluntário poderá contar com mais recursos para a realização do processamento das atividades, aumentando a capacidade total de processamento.

1.2 Estrutura do Texto

Para oferecer um entendimento geral e proporcionar uma melhor leitura, esta dissertação está organizada em mais cinco capítulos, que oferecem uma visão geral dos conceitos envolvidos no estudo, chegando aos detalhes de projeto e implementação do modelo proposto. O capítulo 2 apresenta uma visão geral sobre soluções de computação distribuída. Nesse capítulo, são apresentados e definidos conceitos relacionados com computação em cluster, *peer-to-peer* e computação em grade, chegando no objetivo principal do estudo, que é a computação voluntária. É dentro do modelo de computação voluntária que a proposta para checkpoint com prevalência de objetos se aplica.

Em seguida, o capítulo 3 apresenta um estudo sobre soluções e implementações de mecanismos de *checkpoint* e *restart*. Esse capítulo apresenta uma fundamentação teórica sobre mecanismos de *checkpoint* e *restart* e classifica as implementações em quatro grandes grupos: soluções para plataformas de computação voluntária, implementações em nível de usuário, implementações em nível de sistema operacional e persistência ortogonal. Para cada um desses grupos, as principais soluções foram estudadas e apresentadas.

O capítulo 4 apresenta o estado da arte sobre *diskless checkpoint*. Nesse capítulo são apresentados os principais modelos de solução para *checkpoint* em memória: *neighor-based*, *parity-based*, *checksum-based* e *weighted-checksum*. Ao final deste capítulo, são apresentadas algumas vantagens e limitações do modelo, quando comparado ao *checkpoint* tradicional em disco.

O capítulo 5 entra em detalhes de projeto e implementação do modelo de *checkpoint* proposto nessa dissertação. É nesse capítulo onde a origem dessa proposta surgiu e também os conceitos de prevalência de objetos são estabelecidos. Também é mostrado o projeto e implementação do modelo proposto, bem como as adaptações necessárias ao *worker* do XtremWeb.

Por fim, o capítulo 6 utiliza uma aplicação real para validar o conceito proposto. Nesse capítulo, uma aplicação real é adaptada e utilizada dentro do conceito de *checkpoint* com prevalência de objetos e os resultados de desempenho são coletados e apresentados. É nesse capítulo que o desempenho, em termos de performance, é medido para a aplicação

modificada e executada dentro do modelo de *checkpoint* proposto e comparado com a execução da aplicação original. Para que os resultados avaliados se aproximassem do tipo de recurso para o qual o modelo de *checkpoint* foi proposto, os testes descritos nesse capítulo foram executados em um ambiente que simula uma impressora multifuncional.

No capítulo 7, são apresentadas as conclusões do estudo realizado nessa dissertação, além de indicações das contribuições obtidas. Além disso, esse capítulo apresenta indicativos para trabalhos futuros, onde o modelo pode ser ampliado e adaptado para atender outros problemas.

2 COMPUTAÇÃO DISTRIBUÍDA

O acesso a grandes quantidades de poder computacional tem sido, por diversas décadas, um dos objetivos centrais de muitos cientistas da computação. Desde a década de 60, visões sobre componentes e ferramentas computacionais, sejam elas simples ou mais complexas, têm guiado os esforços de usuários e engenheiros de sistemas (ORGANICK, 1972).

Na década de 70 surgiram as idéias cujos pensamentos centrais estavam focados no objetivo de atingir esse poder computacional. A idéia se apresentava através da coleção e utilização de diversos recursos pequenos e baratos em detrimento de poucos e caros supercomputadores (STERLING et al., 1995). A partir desse pensamento, o interesse em esquemas para o gerenciamento desses processadores distribuídos se tornou cada vez mais popular (THAIN et al., 2005).

Quando a utilização de múltiplos computadores para resolver um único problema computacional se tornou possível, o foco passou para o estudo e pesquisa na utilização máxima dos ciclos de CPU desses equipamentos.

Para as diversas formas de organização em rede para resolução de problemas de maneira distribuída, atribui-se o nome de computação distribuída. Esse capítulo apresenta alguns dos principais conceitos relacionados à computação distribuída: *Cluster Computing* e suas classificações, *peer-to-peer*, computação em grade e computação voluntária.

2.1 Computação em *Cluster*

De acordo com (TOTH, 2004), um cluster pode ser inicialmente definido como um grupo de computadores que trabalham em conjunto de uma maneira fortemente acoplada que, em alguns aspectos, pode ser visto e entendido como um único computador. Os componentes de um cluster são ligados entre si, tipicamente, por uma rede local de alta velocidade (TOTH, 2004).

Dentre as diversas aplicações que utilizam cluster, cabe destaque às soluções nos ramos de simulações, biotecnologia, mercado de finanças, geologia, *data mining*, processamento digital e mesmo para computação de jogos pela internet (TOTH, 2004).

De acordo com (BAKER et al., 1999) um cluster em sua formatação mais simples pode ser estabelecido através dos computadores interconectados em uma rede interna de uma empresa ou universidade. Essa formatação é definida por Buyya et al em (BAKER et al., 1999) como um workstation *cluster*, que é sinônimo para um cluster. Em (BAKER et al., 1999), o artigo estabelece que além da conexão de hardwares, um cluster também inclui a utilização de um *middleware* que permite aos computadores participantes do cluster atuarem como nodos de um sistema distribuído, atuando sobre aplicações que foram designadas para rodarem neles.

Um sistema de cluster, segundo (TOTH, 2004), pode ter seus nodos alocados de acordo com duas abordagens simples: estática ou dinâmica. Na alocação estática, os nodos computacionais do cluster são organizados de maneira próxima, tendo como objetivo a execução de uma tarefa complexa qualquer, já conhecida. Através da alocação dinâmica, a arquitetura de alocação dos nodos do cluster somente é conhecida momentos antes da tarefa de processamento ser iniciada.

Segundo definido por (BARAK et al., 1999), (BAKER et al., 1999) e (TOTH, 2004), um cluster é composto por todos os componentes que podem ser encontrados em uma LAN de computadores e *workstation*s: computadores individuais com seus processadores, memória e discos; cartões de rede; cabeamentos; bibliotecas de software e sistemas; sistemas operacionais; *middlewares* e quaisquer outras ferramentas ou utilitários.

De acordo com (BARAK et al., 1999) *Computing Clusters* são compostos por coleções de *workstations* sem compartilhamento de recursos (*share-nothing workstations*) e servidores, com velocidades e quantidade de memória semelhantes. Um cluster é definido como heterogêneo, para as situações em que os nodos que o compõem são formados por diferentes tipos de computadores enquanto que um cluster homogêneo é composto por nodos de um único tipo de computadores (TOTH, 2004). Na maioria das configurações, os *computing clusters* são organizados para que suportem múltiplos usuários, em ambientes que sejam propícios para compartilhamento de tempo (BARAK et al., 1999). Em *computing clusters* os usuários dos recursos são responsáveis por alocar os processos desejados nos nodos do cluster e manter e gerenciar tanto os recursos quanto os nodos durante a execução. De acordo com (BARAK et al., 1999), mesmo que todos os nodos do cluster rodem o mesmo sistema operacional, a cooperação entre os nodos é relativamente limitada pois a maioria dos serviços oferecidos pelo sistema operacional em questão são limitados e confinados a cada nodo.

Soluções de cluster podem ter diferentes tamanhos e, por isso, uma das maiores vantagens de soluções que sigam essa abordagem é a escalabilidade (RECHERCHE NUCLéAIRE, 2009), pois o crescimento de um cluster está vinculado tão somente a adição de novos computadores (nodos) a ele. Essa abordagem, no entanto, possui limites pois de alguma forma os computadores desse cluster precisam se comunicar entre si e isso começa a trazer problemas, quando se fala de cluster com muitos computadores (RECHERCHE NUCLéAIRE, 2009).

O artigo (JATIT, 2009) define algumas características básicas que configuram e definem um *Cluster Computing*, as quais são transcritas abaixo e complementadas com as características estabelecidas em (AHMAD, 2000):

- Sistemas fortemente acoplados, o que resulta em uma arquitetura fortemente centralizada;
- Configura (e deve ser visto como) uma imagem única de sistema;
- Gerenciamento de trabalhos centralizado;
- Sistema de escalonamento centralizado;
- Deve prover escalabilidade, permitindo ao sistema utilizar mais ou menos recursos, de acordo com a necessidade;
- O sistema operacional utilizado e o mecanismo de comunicação, por serem únicos a todo o cluster,

- devem ser suficientemente eficientes para remover ou mitigar gargalos de performance.

De acordo com (JATIT, 2009), é possível afirmar que quando dois ou mais computadores são utilizados em conjunto para a resolução de um problema, a arquitetura formada a partir da junção destes pode ser chamada de *computer cluster*. Por essa definição, entende-se que *Cluster Computing* é a ação de executar algum problema em um cluster (JATIT, 2009). Em (RECHERCHE NUCLéAIRE, 2009) e (JATIT, 2009), é feita uma definição em relação à abrangência de um cluster. De acordo com os autores, um cluster está contido em um local ou um complexo definido e fixo, como, por exemplo, a rede de uma universidade ou computadores de uma empresa.

Diferente da estrutura simples que se encontra na rede de uma universidade ou na rede de uma empresa, onde cada computador é administrado e gerenciado individualmente por uma pessoa, em um cluster todos os nodos são administrados, tipicamente, por uma única pessoa. Ao contrário de executar tarefas específicas e particulares de um usuário, os nodos de um cluster trabalham em conjunto para realizar tarefas de computação intensa, procurando resolver tarefas que não podem ser executadas em máquinas individuais, devido ao baixo poder de processamento ou devido a questões de tempo (TOTH, 2004).

Mesmo que todos os nodos de cluster precisem ser acionados para resolver um determinado problema, o poder de um cluster está justamente na habilidade de cada nodo trabalhar em conjunto para resolver um pedaço específico do problema, resultando em um ganho de performance que pode ser comparado a um supercomputador. Ao contrário dos caros supercomputadores, um cluster pode ser formado por dezenas de centenas de computadores pequenos e baratos, configurando uma solução barata quando comparada aos supercomputadores (TOTH, 2004).

De acordo com (TOTH, 2004) e baseado em (STEEN, 2007), os *clusters* podem ser categorizados em:

2.1.1 Cluster de Alta-Disponibilidade

O objetivo principal dessa arquitetura de *clusters* é oferecer um sistema de alta disponibilidade, tolerante a falhas. A melhoria de disponibilidade desse tipo de cluster é obtida através do estabelecimento de nodos redundantes para cada serviço de processamento que está sendo executado. Por padrão, determina-se a existência de 2 nodos para cada serviço - que, por conseqüência estabelece o limite de tolerância existente (se os dois nodos falharem, o *cluster* deixa de ser de alta-disponibilidade). Existem diversas implementações comerciais de *clusters* HA no mercado, para diversos sistemas operacionais. Um exemplo é o Linux-HA, que é uma solução *open source* para cluster de alta disponibilidade para o sistema operacional Linux.

2.1.2 Cluster com Balanceamento de Carga

O objetivo desse tipo de arquitetura de cluster é dividir a carga de trabalho computacional entre os nodos, objetivando ganhos de performance e redução do tempo de processamento, além de uma melhor divisão na carga de trabalho para cada nodo. Esse tipo de arquitetura de cluster é também chamado de *server-farm* (TOTH, 2004). Existem no mercado diversas soluções de software que contemplam esse tipo de arquitetura, como por exemplo *Sun Grid Engine, Moab Cluster* e *Maui Cluster Scheduler*.

2.1.3 Cluster de Alto Desempenho

O objetivo desse tipo de arquitetura de cluster é dividir a tarefa computacional em diversas sub-tarefas e encaminhar cada uma delas para os diversos nodos do cluster. As aplicações executadas em *clusters* HPC devem ser planejadas para que, quando divididas, minimizem o custo de comunicação entre os nodos pois, tipicamente, o resultado do processamento de um nodo é importante para as atividades de outros nodos.É comum encontrar *clusters* HPC construídos sob linux, que utilizam a biblioteca MPI para as atividades de comunicação.

2.1.4 Cluster com *Web-Service*

Essa classificação de *clusters* diz respeito à organização sistemática que é feita para permitir que a descoberta e utilização de *web services* possa ser feita através de métodos de mineração de textos, análises lingüísticas e técnicas estatísticas combinadas (LIO, 2005).

2.1.5 Cluster em Bases de Dados

De acordo com (DOHERTY, 2003), *Database clusters* podem ser definidos como uma classificação de *clusters* que compartilham o acesso, manipulação e gerenciamento de informações armazenadas em bases de dados, tendo como objetivo principal a melhoria de performance.

Conforme definido em (CHANDRASEKARAN; KEHOE, 2002), os *clusters* de bases de dados podem ser divididos em duas categorias:

- *Shared Nothing clusters*: nesse tipo de arquitetura, os arquivos da base de dados são distribuídos através das diversas instâncias de bases de dados que são executadas nos nodos dos *clusters*. Cada instância tem total controle sobre um conjunto específico de dados e todo e qualquer acesso sobre estes, deve ser feito através do nodo proprietário (CHANDRASEKARAN; KEHOE, 2002);

- *Shared Disk clusters*: nessa arquitetura compartilhada, os arquivos da base de dados são logicamente distribuídos através dos nodos. Através dessa abordagem, todos os nodos do cluster têm acesso a todos os dados (DOHERTY, 2003). O acesso a disco compartilhado, segundo (CHANDRASEKARAN; KEHOE, 2002), é feito através de acesso direto a hardware ou através de uma camada de abstração no sistema operacional, que oferece uma visão simples de todas as instâncias, de todos os nodos.

2.2 Peer to Peer Computing

O termo *peer-to-peer* se refere a uma classe de sistemas e aplicações que empregam recursos distribuídos para executar uma determinada função ou atividade de uma maneira descentralizada (MILOJICIC et al., 2003).

Enquanto um *peer* é um computador que se comporta como um cliente em um modelo cliente/servidor, ele também possui uma camada adicional de software que permite, no modelo P2P, que se comporte para determinadas funções como um servidor. Os *peers* podem então responder às requisições de outros *peers*, como se fossem servidores respondendo a chamadas dos clientes, na arquitetura cliente servidor tradicional. O escopo das

chamadas e das respostas e a maneira como são executadas, são de domínio específico da aplicação (BARKAI, 2000).

Segundo (MILOJICIC et al., 2003), muitos especialistas na área acreditam que não existem muitas novidades na arquitetura P2P e que existe muita confusão e indefinição em relação ao que ela se difere no modelo tradicional de cliente-servidor distribuído. Em (BARKAI, 2000), esse questionamento é totalmente respondido quando o autor define que as soluções P2P se apresentam como uma alternativa complementar à arquitetura tradicional de cliente-servidor. Segundo (BARKAI, 2000), P2P oferece um modelo computacional ortogonal em relação ao modelo cliente-servidor tradicional, permitindo que os dois modelos possam coexistir, se cruzarem e se complementarem.

No modelo cliente-servidor tradicional, o cliente faz uma requisição ao servidor. O servidor, tipicamente um recurso dedicado, responde à requisição e atua, de acordo com o que foi solicitado pelo cliente. Na abordagem das arquiteturas P2P, cada computador participante, chamado de peer, funciona como um cliente com uma camada de funcionalidades de servidor. Isso permite ao peer, atuar tanto como cliente - fazendo requisições - como servidor, respondendo à requisições de outros clientes, dentro do contexto de uma aplicação P2P (BARKAI, 2000).

As aplicações P2P podem ter funções de armazenamento, processamento de informações, troca de mensagens, segurança e, principalmente, compartilhamento de arquivos. Para todas essas funções, as soluções feitas com uma arquitetura P2P permitem que os *peers* troquem informações diretamente entre si, sem a necessidade da utilização de um ou mais servidores específicos, centralizados.

Como mencionado anteriormente, um peer pode iniciar requisições e pode também responder a requisições enviadas à rede por outros *peers*. Essa habilidade de executar trocas de informações diretamente com outros usuários (*peers*) faz com que não seja mais necessária a dependência de um servidor central (BARKAI, 2000). Isso permite um maior grau de autonomia e controle, por parte dos usuários, sob os serviços que estão disponíveis para atualização.

Como citado em (BARKAI, 2000) um dos maiores benefícios das arquiteturas P2P é a possibilidade de organização dos *peers* em comunidades. Isso permite aos usuários mecanismos para que se organizem em grupos *ad hoc* que podem, de maneira segura e eficiente, compartilhar recursos e colaborar entre si para resolução de um problema comum. Além desse benefício (BARKAI, 2000) e (MILOJICIC et al., 2003) citam como recursos e facilidades de uma arquitetura P2P:

- possibilidade de agregar novos recursos através de um custo de interoperabilidade baixo;
- custo baixo em relação ao compartilhamento de informação, através da utilização de infra-estruturas já existentes;
- permite anonimato e privacidade, através da incorporação desses cuidados nos algoritmos e programas P2P.

Entretanto, (MILOJICIC et al., 2003) destaca que sistemas P2P despertam nos usuários algumas preocupações no que diz respeito a segurança das informações. Porém, em contraponto, o mesmo autor cita que como trata-se de uma tecnologia em pleno desenvolvimento, esforços estão sendo feitos para mitigar esse problema.

As arquiteturas P2P podem, em algumas situações, fazer uso de servidores centralizados. Em relação a essa abordagem, alguns especialistas em P2P traçam uma clara

distinção de termos, classificando os sistemas em "Sistemas P2P Puros"e "Sistemas P2P Híbridos". O termo "puro"denota os modelos onde todos os computadores participantes são *peers*. Nos sistemas ditos puros, não existe um servidor responsável por coordenar, controlar ou gerenciar as trocas de dados e informações entre os *peers*. De outro lado, o termo híbrido define justamente as arquiteturas P2P que fazem uso desses servidores centralizados, para a execução de algumas atividades. O grau de envolvimento desse servidor centralizado varia de acordo com a aplicação. Em (BARKAI, 2000) é usado o exemplo do Napster, onde é requerido ao usuário que se conecte a um servidor de controle, para então obter acesso aos recursos, propriamente ditos.

2.2.1 Objetivos de um sistema *peer-to-peer*

Como todo e qualquer sistema computacional, os sistemas P2P têm alguns objetivos principais, para garantir que a solução satisfaz as necessidades dos usuários. Escolher a arquitetura P2P para implementação de algum recurso, na maioria das vezes, é guiada por alguns dos objetivos (BARKAI, 2000):

- Redução do custo de compartilhamento: sistemas centralizados que possuem muitos clientes, tipicamente, possuem um custo alto para manter recursos compartilhados;
- Aumento da disponibilidade: sem a necessidade de um recurso central dedicado (que pode falhar) a responsabilidade se divide em inúmeros *peers* que, individualmente podem falhar sem comprometer a disponibilidade;
- Aumento da escalabilidade: a adição de novos nodos a rede P2P é descomplicada e fácil, fazendo com que novos participantes sejam agregados e possam rapidamente participar de maneira ativa;
- Privacidade: um usuário pode querer que nenhum serviço saiba de sua participação ou envolvimento;
- Dinamismo: sistemas P2P assumem que o ambiente é altamente dinâmico. Isso significa estar preparado para que um grande número de recursos entre e saia de maneira dinâmica e indeterminada;
- Agregação de Recursos: por não ter um serviço centralizado para controle, agregar novos componentes ou recursos é uma tarefa fácil em ambientes P2P;
- Autonomia: os usuários não precisam confiar em um único serviço centralizado. A informação pode estar disponível em outros *peers* ou até mesmo para consulta e manipulação local;

2.3 Computação em Grade

De acordo com (STEEN, 2007), as grades computacionais podem ser classificadas como uma caracterização de cluster, pois se assemelham em diversos aspectos: assim como os *clusters*, as grades computacionais fazem uso de diversos computadores (recursos) conectados de alguma maneira, organizados para resolver um problema de tamanho considerável (TOTH, 2004).

O termo *Grid computing* foi criado na segunda metade da década de 90 pelo dr. Ian Foster (ECONOMIST, 2008). A idéia do professor Foster era fornecer recursos computacionais dentro da mesma lógica em que se fornece energia elétrica: qualquer pessoa deveria ter acesso a recursos computacionais para resolver um problema sempre que necessário (FOSTER et al., 2001). No artigo "Anatomia de um Grid", os professores Foster, Kesselman e Teucke definiram o "problema de grid"como um "compartilhamento de recursos flexível, seguro, coordenado entre indivíduos, instituições e recursos"(FOSTER et al., 2001). A *posteriori*, Foster estabeleceu três características básicas, que definem as características de uma grade e dos problemas que nelas são executados:

1. Um grid utiliza "interfaces e protocolos genéricos, padrões e abertos"(FOSTER, 2002);

2. Os recursos utilizados não estão sob o controle de uma única entidade ou instituição;

3. Um grid gera, como resultado, serviços não triviais com qualidade"(FOSTER, 2002).

Baseado nas conclusões de (JATIT, 2009) e de acordo com as características estabelecidas por Foster, pode-se estabelecer que, ao contrário de um cluster típico, as soluções em grade são descentralizadas, diversificadas e dinâmicas, além de possuírem um gerenciamento de trabalhos e escalonamento distribuído. Entretanto, o termo *grid computing* muitas vezes é utilizado para descrever muitos tipos de arquiteturas que não estão de acordo com as definições de Foster *et al*, causando dúvidas sobre a real definição do termo (FOSTER et al., 2001).

As definições muitas vezes não são suficientes para esclarecer todas as dúvidas e confusões que surgem a partir da tentativa de diferenciação dos termos *Cluster Computing* e *Desktop Computing*. De acordo com (TOTH, 2004) e (CROGRID, 2009), algumas características podem ser apontadas como fatores de diferenciação:

- Grids são totalmente heterogêneos, enquanto que *clusters*, em sua grande maioria, são homogêneos;

- As grades computacionais podem fazer uso do tempo ocioso dos computadores que a compõem, permitindo que exista um processamento "extra-grade"desses recursos; nos *clusters*, o trabalho dos nodos é totalmente dedicado a uma única atividade e nada mais;

- As grades computacionais podem ser compostas por computadores localizados ao redor do mundo, através da internet, enquanto que os *clusters*, usualmente são montados dentro de uma única localidade ou complexo;

- Não existe confiança entre os recursos que compõem uma grade computacional, ao contrário do que existe nos *clusters*.

Em (TOTH, 2004), o autor destaca que as grades computacionais são otimizadas para executarem tarefas de processamento que consistem em diversos *jobs* ou pacotes de trabalho independentes, onde não é necessário existir troca de dados e de mensagens entre os pacotes de trabalho durante o processamento das atividades. As grades são responsáveis por gerenciar a alocação dos pacotes de trabalho nas estações de trabalho que irão resolver o problema, independente do comportamento dos demais recursos da grade computacional (TOTH, 2004).

2.4 Computação Voluntária

O termo computação voluntária define um modelo computacional emergente que se adapta facilmente ao modelo de processamento de *work-pool*. Esse conceito provou sua capacidade de agrupar recursos e oferecer ambientes para processamento de alto desempenho em projetos de sucesso, como por exemplo SETI@Home, Folding@Home entre outros (P.ANDERSON; FEDAK, 2006). A título de exemplificação, o projeto SETI@Home agrega, ao redor do mundo mais de 1,5 milhão de desktops que, de maneira voluntária, cedem suas capacidades de processamento para utilização científica (ANDERSON et al., 2002) e (PRESS, 2003). Esse grande número de dispositivos agregados levam a uma capacidade de processamento de aproximadamente 250 TeraFlops, de acordo com estatística disponível [1]. Esse poder de processamento pode ser comparável ao IBM Blue Gene, que oferece uma velocidade de 360 TeraFlops.

2.4.1 Computação Voluntária como Metacomputação

Em (SARMENTA, 2001), o autor define computação voluntária como uma forma de metacomputação, que maximiza o modo como os usuários podem fazer com que suas máquinas façam parte de um metacomputador. O termo metacomputação tipicamente denota uma organização estrutural que configura um computador virtual, consistido de computadores individuais localizados de maneira geograficamente distribuída (possivelmente), interconectados através de uma rede de alta velocidade. Metacomputação é, portanto, motivada pela real necessidade de acessar recursos que freqüentemente não estão localizados em um único sistema computacional (FOSTER; KESSELMAN, 1997).

Metacomputação permite que aplicações de larga escala utilizem recursos computacionais de alta performance de uma maneira desacoplada (KESSELMAN et al., 1998). Esse tipo de sistema esconde toda a complexidade de gerenciamento do usuário, permitindo que este permaneça focado em seu negócio, em sua ciência e menos preocupado com os detalhes de implementação e configuração do ambiente. O objetivo principal de utilizar um sistema de metacomputação é permitir que recursos que tipicamente estariam indisponíveis, estejam disponíveis permitindo que sejam utilizados pelos usuários, para o processamento de uma determinada tarefa. Essa utilização, por sua vez, esconde a complexidade do gerenciamento dos recursos do usuário, facilitando as atividades e fazendo com que ele foque os seus esforços única e exclusivamente nos seus objetivos científicos (CRONK et al., 2000).

Através da utilização dos conceitos de metacomputação, é possível que problemas muito grandes ou complexos de serem executados em sistemas de alta performance tradicionais, sejam executados com sucesso (CRONK et al., 2000) nesse ambiente de execução. Os sistemas de metacomputação existentes atualmente assumem sua implementação baseada no conceito de NOW. Essa arquitetura permite às pessoas conectarem os seus recursos em uma rede de processamento maior, objetivando a participação em uma solução paralela distribuída ampla. Essa abordagem evita a necessidade de investimentos em um supercomputador (FIELDS, 2008), tornando a solução efetiva e financeiramente viável.

Enquanto outras formas de metacomputação, como *grid computing*, buscam fazer com que o uso do poder de processamento de computadores em uma rede seja facilitado, o processo de configuração desses computadores não é algo trivial. Atualmente, a maioria desses projetos foca em disponibilizar computadores ou recursos de uma rede para utilização de outros. De modo que essas máquinas, por muitas vezes não estarem em mãos de

[1] Estatística disponível em http://boincstats.com/stats, acessado em Fev. 2009

especialistas, fazer um processo de instalação simples não é uma prioridade. É justamente essa complexidade de instalação e configuração que faz com que usuários comuns fiquem sem participar desses sistemas.

A idéia por trás da computação voluntária é eliminar esses passos complexos de configuração, fazendo com que a participação de qualquer usuário, independente do conhecimento técnico que possua, seja facilitada. Isso acarreta como resultado o fato de que todos, mesmo usuários casuais de sistemas de informática sem muito conhecimento técnico, possam participar de sistemas desse tipo (SARMENTA, 2001). Por permitir que usuários casuais possam contribuir com o tempo ocioso de suas máquinas, o uso de ambientes de computação voluntária torna possível juntar o processamento de centenas ou mesmo milhares de computadores, objetivando o desenvolvimento de um determinado projeto de interesse (SARMENTA, 2001).

2.4.2 Computação Global

Os conceitos de Computação Global e Computação Voluntária são semelhantes e, de certa forma complementares. De acordo com (NERI et al., 2001), a idéia central de Computação Global é otimizar o máximo possível o uso dos computadores que podem estar largamente distribuídos ao redor do mundo, conectados através da internet.

Essa otimização e utilização de recursos largamente distribuídos poderia ser utilizada para executar uma aplicação distribuída de grande porte qualquer. Ainda baseado em (NERI et al., 2001), todo o poder de processamento a ser utilizado é cedido por computadores voluntários, que estariam dispostos a oferecer uma fatia do tempo de processamento ocioso em prol da execução de parte dessa grande aplicação distribuída. Através da definição oferecida é possível compreender que os conceitos se assemelham bastante. O termo global pode, então, ser entendido como uma utilização, em termos globais, de computadores que cooperam entre si de maneira voluntária, interconectados por uma rede de computadores, como a internet, por exemplo.

2.4.2.1 Problemas Relacionados à Computação Global

Quando se fala em projetos de aplicações construídos sob o conceito de computação global através da utilização de recursos voluntários, é importante endereçar algumas preocupações tipicamente relacionadas. Em (NERI et al., 2001), o autor enumera essas preocupações que as aplicações devem ter:

1. Escalabilidade: deve escalar para centenas de milhares de nodos, com um aumento de performance correspondente;

2. Heterogeneidade: deve ser heterogêneo em relação a hardware, sistema operacional e softwares básicos;

3. Disponibilidade: o proprietário do recurso é quem determina a política de uso do mesmo. E o sistema global em questão, deve garantir e obedecer toda e qualquer restrição imposta;

4. Tolerância a falhas: a MTBF de uma máquina conectada na internet é de aproximadamente 13 horas. Dessa forma, o sistema global deve ser tolerante a esse constante número de falhas e, mesmo assim, manter um nível de performance aceitável;

5. Segurança: todos os nodos participantes devem estar protegidos de execuções de códigos maliciosos ou manipulações equivocadas;

6. Dinamicidade: o sistema deve ser suficientemente dinâmico para acomodar um número freqüentemente variável de recursos disponíveis, bem como possíveis mudanças freqüentes em relação a velocidade e largura de banda das conexões;

7. Usabilidade: deve ser fácil para utilização e o *deploy* da aplicação ou projeto deve ser descomplicado.

2.4.2.2 Exemplos de Projetos

Existem inúmeros exemplos de aplicações criadas utilizando *frameworks* para computação voluntária, que podem ser citados como exemplos de soluções que se enquadram no conceito de *Global Computing*. Em (FEDAK et al., 2001) o autor cita algumas, como:

2.4.2.2.1 SETI@Home

Trata-se de um dos projetos mais populares desenvolvidos através da utilização da plataforma BOINC. O projeto SETI@home tem por objetivo analisar os sinais de rádio freqüência coletados do espaço, para busca de padrões que possam identificar a existência de vida extra-terrestre.
Nro. Ativo de usuários: 199,000
*Sistemas Operacionais Suportados:*Linux, Mac OS X, Solaris e Windows

2.4.2.2.2 GIMPS

Esse projeto de computação voluntária visa concentrar esforços na busca de números Mersennes que sejam primos. Em matemática, pode ser definido como um número primo, definido através da seguinte forma:

$$M_n = 2^n - 1$$

Esse projeto então visa encontrar números n, que sejam primos. Para esse objetivo, a solução é fazer uso do conceito de computação voluntária.
Nro. Ativo de usuários: desconhecido
Sistemas Operacionais Suportados: Linux, FreeBSD, OS/2 e Windows (Versões "extra-oficiais"de outros Sistemas Operacionais).

2.4.2.2.3 Einstein@Home

Projeto, construído na plataforma BOINC, que objetiva fracionar as ondas gravitacionais em pequenas porções e compará-las individualmente, contra modelos pré-definidos.
Nro. Ativo de usuários: 38,000
Sistemas Operacionais Suportados: Linux, Mac OS X, Solaris e Windows.

2.4.2.2.4 LHC@Home

Projeto criado sob a plataforma BOINC, que simula o movimento de partículas dentro do LHC - acelerador de partículas do CERN.
Nro. Ativo de usuários: 14,000
Sistemas Operacionais Suportados: Linux, Mac OS X, Solaris e Windows

2.4.2.2.5 Folding@Home

Um dos projetos mais antigos para simulação de proteínas. Os projetos atuais visam o estudo de doenças como Alzheimer, Câncer, Hintington e Parkinson entre outras.

Nro. Ativo de usuários: 268,000
Sistemas Operacionais Suportados: Linux, Mac OS X, Windows, PLAYSTATION 3.

2.4.2.2.6 MalariaControl.net

Projeto que foca o esforço nos estudos sobre a Malária na África. O estudo é feito levando em conta diversos fatores biológicos e sociais e os resultados ajudam os cientistas na criação de vacinas e similares.
Nro. Ativo de usuários: 8.000
Sistemas Operacionais Suportados: Linux, Mac OS X e Windows.

2.4.2.2.7 Rosetta@Home

Utiliza a plataforma BOINC para auxiliar no entendimento de doenças como HIV, Malaria, Câncer através de simulações em 3D de proteínas.
Nro. Ativo de usuários: 49.000
Sistemas Operacionais Suportados: Linux, Mac OS X e Windows

2.4.2.2.8 Xgrid@Stanford

Projeto de pesquisa para, conforme definição fornecida pela universidade de Stanford, modelar mudanças do receptor beta 2. O projeto é desenvolvido sob o XGrid e é focado para computadores Macintosh apenas.
Nro. Ativo de usuários: 500
Sistemas Operacionais Suportados: Mac OS X

2.4.2.2.9 ClimatePrediction.net

Criado sob a plataforma BOINC, esse projeto tem como objetivo executar projeções climáticas através de diferentes variáveis para verificar e medir a evolução do aquecimento global.
Nro. Ativo de usuários: 31.000
Sistemas Operacionais Suportados: Linux, Mac OS X e Windows

Além destes projetos mencionados, existem diversos outros que também utilizam frameworks e *middlewares* para computação voluntária, como por exemplo *Predictor@Home, Evolution@Home, SIMAP, EON Project, Drug Design and Optimization Lab, AfricanClimate@Home, FightAIDS@Home, Help Conquer Cancer, Human Proteome Folding project Phase 2, Seasonal Attribution, ABC@home, Pi Segment, Seventeen or Bust, DIMES, The Lattice Project, SoundExpert* e *yoyo@home* entre outros.

2.4.3 Frameworks para Computação Voluntária

Existem diversos *frameworks* de computação voluntária, existentes no mercado, disponíveis para utilização. Dentre os *frameworks*, os mais citados e utilizados são o Bayanihan, BOINC e XtremWeb, todos desenvolvidos em universidades e, em sua grande maioria, com código aberto. Entretanto, existem também soluções proprietárias, com código fechado. As seções seguintes apresentam alguns dos *frameworks* para computação voluntária disponíveis.

2.4.3.1 Bayanihan

Bayanihan foi desenvolvido pelo pesquisador Luis Sarmenta, como parte da sua tese de doutorado. Embora não seja mais utilizado formalmente, esse *framework* foi o pri-

meiro sistema genérico de computação voluntária baseado em Java, para soluções web e, segundo o autor, foi a primeira solução de computação voluntária para internet disponibilizada (SARMENTA, 2001). Bayanihan é um *framework* que permite que os usuários compartilhem seus computadores como recursos voluntários sem riscos de segurança e sem a necessidade de instalação de nenhum software extra. Para participar como recurso voluntário, bastava aos usuários visitar a página do projeto na internet e, de maneira automática, o *browser* realizava o *download* e executava uma *applet* Java com a aplicação ou projeto (SARMENTA, 2001).

2.4.3.2 BOINC

É o principal e mais utilizado *framework* para construção de projetos de computação voluntária. BOINC foi desenvolvido na Universidade da Califórnia, no laboratório de ciências espaciais, em Berkeley (ANDERSON, 2004). BOINC permite a uma pessoa com um computador simples criar um projeto de computação voluntária sem a necessidade de escrever código para o cliente e o servidor. O *framework* provê os *scripts* e ferramentas necessárias para configurar todo o ambiente, exceto o código de processamento, que é a tarefa que deve ser codificada pelo criador do projeto. O *framework* permite ainda que os participantes (voluntários) realizem ajustes customizados nas configurações, que afeta como o BOINC será executado em suas máquinas, garantindo total personalização (STAFF, 2008).

2.4.3.3 Xgrid

Xgrid é uma solução para computação voluntária desenvolvida pela Apple, focada especificamente para as arquiteturas Mac OS. De acordo com os autores do *framework*, projetos que utilizem Xgrid geram uma solução escalável e de alta disponibilidade.

2.4.3.4 GridMP

Grid MP é uma solução de *middleware* para computação voluntária comercial, desenvolvida pela empresa *United Devices*. De acordo com (DEVICES, 2008) a solução GridMP é utilizada com sucesso em projetos como grid.org, *World Community Grid*, Cell Computing e Hikari Grid. Grid MP oferece mecanismos para escalonamento de tarefas com priorização, restrições de segurança, exclusão de aplicações, detecção de atividades. GridMP pode ser utilizado para o gerenciamento de recursos como computadores pessoais, servidores ou nodos dedicados de *clusters*; esses recursos podem ser organizados em grupos organizacionais para segurança e controles administrativos (ASHOK, 2007).

2.4.3.5 Outros Frameworks

Em complemento aos *frameworks* que estão disponíveis livremente para uso, existem outros softwares e componentes que podem ser utilizados para criar projetos de computação voluntária personalizados. Existem também empresas como *United Devices* e Entropia, que produzem componentes de software que podem ser usados em soluções de computação voluntária.

2.5 Considerações Finais

Esse capítulo apresentou alguns dos principais conceitos relacionados à computação distribuída. O primeiro conceito detalhado foi o de computação em *cluster*, onde as suas

classificações foram apresentadas e definidas. O segundo conceito apresentado foi o de *peer-to-peer*, onde os objetivos desse tipo de sistema foram detalhados. Além disso, também foi introduzido o termo computação em grade.

O termo computação voluntária, por sua vez, foi apresentado em mais detalhes para que fosse possível permitir o entendimento do conceito que o define. Isso foi feito pois o trabalho proposto nessa dissertação é diretamente aplicado a plataformas de computação voluntária. O modelo de *checkpoint* com prevalência de objetos que é detalhado através da pesquisa descrita nessa dissertação, tem como objetivo principal, ser executado em plataformas de computação voluntária.

3 IMPLEMENTAÇÕES DE MECANISMOS PARA *CHECKPOINT* E *RESTART*

Dentro da área de *checkpointing*, diversos trabalhos foram propostos e inúmeros protótipos de soluções implementados. Essas soluções apresentam diferentes técnicas para oferecer a persistência da execução e dos dados dos processos e aplicações, permitindo a recuperação das informações e o reinício da execução em um estado consistente.

Esse amplo conjunto de soluções pode ser dividido em basicamente três categorias, de acordo com a arquitetura da solução e o nível de invasão ou modificação que ele gera, dentro do sistema ou aplicação ao qual está acoplado. A primeira categoria contempla as soluções onde o *checkpoint* é realizado pela própria aplicação, diretamente no código. A segunda, contempla os mecanismos onde o *checkpoint* é realizado através de bibliotecas ligadas à aplicação. A terceira categoria faz referência aos mecanismos para *checkpoint* que são implementados diretamente no sistema operacional. Podemos ainda adicionar uma quarta categoria que são os mecanismos de *checkpoint* implementados dentro dos *middlewares* para computação voluntária, que podem utilizar quaisquer das três categorias mencionadas acima.

Neste capítulo, serão apresentados os detalhes sobre os mecanismos de *checkpoint* disponíveis nas principais plataformas de computação voluntária, além de detalhar algumas soluções de *checkpoint* que são implementadas através da utilização de bibliotecas extras, que são acopladas à aplicação. Também nesse capítulo, são apresentadas algumas implementações que utilizam recursos do sistema operacional ou que sejam implementadas diretamente como módulos do mesmo. Dentro desse mesmo aspecto, também são apresentados mecanismos de *checkpoint* que são implementados através de modificações nas máquinas virtuais.

3.1 Fundamentação Teórica

Para as categorias de implementações de *checkpoints* mencionadas anteriormente, são apresentados em (ROMAN, 2002), alguns aspectos positivos e também algumas limitações.

Mecanismos de *checkpoint* implementados diretamente nas aplicações se apresentam como as soluções com a maior eficiência, pois o programador da aplicação conhece exatamente os melhores pontos ou situações, bem como as estruturas de dados utilizadas, onde a execução de uma operação de *checkpoint* é necessária. O fato de conhecer exatamente os dados que precisam ser persistidos e os que podem ser descartados, além de conhecer exatamente o momento mais adequado para a realização de *checkpoints* é a principal vantagem dessa abordagem, porém, pode se tornar um grande obstáculo. Isso porque

a aplicação em que se deseja implementar o mecanismo de *checkpoint* pode não ter o código fonte acessível, ou pode pertencer a terceiros tendo o código fonte inacessível.

Outra desvantagem que o modelo de *checkpoint* realizado diretamente dentro da aplicação apresenta, segundo (ROMAN, 2002), é que as aplicações podem oferecer restrições severas em relação ao momento em que a operação de *checkpoint* pode ser realizada. É comum que operações de *checkpoint* só possam ser realizadas depois um determinado período de tempo, o que pode tornar inviável a persistência adequada das informações, pois o tempo transcorrido pode ser muito grande, causando uma grande demora entre os períodos de *checkpoint*.

Implementações de mecanismos de *checkpoint* realizados através de bibliotecas extras endereçam soluções para algumas das deficiências existentes na abordagem de *checkpoint* em nível de aplicação. A principal delas é que a utilização de bibliotecas pode fazer com que não sejam necessárias alterações diretas no código da aplicação, o que pode reduzir o acoplamento da solução.

A utilização de bibliotecas para realização de *checkpoint* também oferece uma solução única para a execução da operação de *restart*. Enquanto que soluções de *checkpoint* realizadas na aplicação podem ter diversos mecanismos diferentes para a recuperação das informações, essa implementação em nível de biblioteca unifica o processo.

Entretanto, a bibliografia apresenta uma limitação da abordagem de *checkpoint* através da utilização de bibliotecas: o fato de utilizar uma biblioteca extra pode impor restrições em relação a quais chamadas de sistema a aplicação pode utilizar. Isso significa que a utilização de bibliotecas limita o número e a variedade de chamadas de sistema que podem ser realizadas pela aplicação. Segundo (ROMAN, 2002), essa limitação faz com que todas as formas de comunicação entre processos (por exemplo *pipes*), sejam geralmente proibidas, o que traz como resultado a impossibilidade de realizar *checkpoint* em aplicações paralelas.

A terceira categoria de mecanismos de *checkpoint* diz respeito a soluções implementadas em nível de sistema operacional, através, por exemplo, de modificações realizadas diretamente no *kernel* do mesmo. Trata-se de outra abordagem eficiente, já que a grande maioria das estruturas de dados são acessíveis ao *kernel* do sistema operacional, se reduzem a mínimas as restrições em relação ao escopo das aplicações. Mecanismos de *checkpoint* implementados na camada de sistema operacional tipicamente permitem que as informações relevantes das aplicações sejam persistidas a qualquer momento. Entretanto, a complexidade envolvida nessa abordagem faz com que os mecanismos de *checkpoint* e *restart* estejam disponíveis apenas para alguns poucos sistemas.

Com base em (SANCHO et al., 2005) os mecanismos de *checkpoint* são classificados tendo como base três dimensões: contexto, agente e implementação. Contexto refere-se ao local onde o *checkpoint* será realizado, em termos de quem é o responsável pela execução: sistema operacional ou usuário. Dependendo do contexto a ser utilizado, existem agentes específicos para a realização do *checkpoint*: programador, compilador, `LD_PRELOAD`, sistema operacional ou hardware. A última camada de classificação é baseada no agente que implementa o *checkpoint* e é responsável por definir, propriamente dito, o modelo de implementação a ser seguido: API, *signal handlers* de usuário ou *kernel*, chamadas diretas de sistema ou *threads* de *kernel*.

A figura 3.1 ilustra a classificação descrita acima. As implementações de mecanismos de *checkpoint* feitas em nível de usuário podem ser programadas no código-fonte da aplicação diretamente pelo programador da aplicação ou inseridas automaticamente por um pré-compilador. Nessas abordagens, via de regra, uma biblioteca específica para *check-*

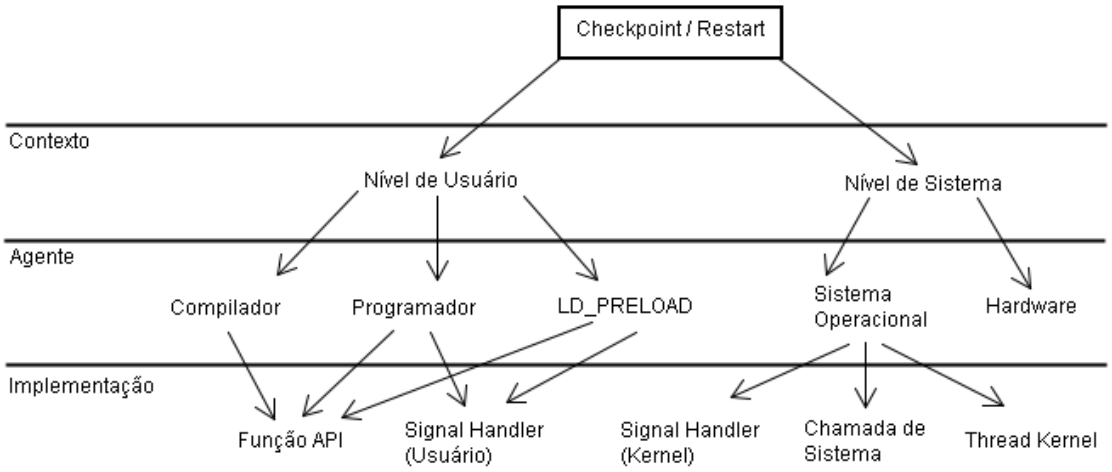


Figura 3.1: Classificação das implementações de mecanismos de *checkpoint/restart*

point é utilizada para fornecer as primitivas necessárias, eliminando a necessidade de diretamente programar essas chamadas. Uma alternativa à modificação de código-fonte da aplicação é a utilização de chamadas às primitivas de *checkpoint* através de *handlers* de sinais, em nível de usuário. Outras implementações fazem uso da variável de ambiente `LD_PRELOAD`, instalando *handlers* de sinais e realizando o carregamento das bibliotecas necessárias para a realização de *checkpoints* sem a necessidade de recompilar ou *linkar* a aplicação. Como alternativa às implementações realizadas na camada de usuário, diversos mecanismos para *checkpoint* são implementados em nível de sistema operacional ou até mesmo diretamente em hardware. Quando trabalha-se diretamente na camada de sistema operacional, existem várias técnicas que possibilitam a realização de *checkpoints*: *handlers* de sinais ligados ao *kernel* do sistema operacional, implementação de chamadas de sistemas ou utilização de *threads* do *kernel* (SANCHO et al., 2005).

3.2 Plataformas de Computação Voluntária

Mecanismos para realização de *checkpoints* também podem ser aplicados a plataformas de computação voluntária, onde também são importantes, mas nem sempre necessários. A necessidade da existência ou não de um mecanismo de *checkpoint* está diretamente vinculada com a aplicação que está sendo executada.

Abaixo são apresentadas as soluções para realização de *checkpoint* para as principais plataformas de computação voluntária.

3.2.1 Condor

O mecanismo de *checkpoint* existente na plataforma Condor oferece ao escalonador a liberdade para realizar operações de escalonamento reaproveitando informações já produzidas. Com essa idéia, caso o escalonador decida não mais executar uma determinada atividade em uma máquina específica, o *checkpoint* das informações intermediárias produzidas até o momento é realizado, e o *job* pode ser realocado em outro recurso, retomando as atividades do ponto onde foi anteriormente interrompido. Adicionalmente ao mecanismo de *checkpoint* preemptivo descrito, a plataforma oferece também a possibilidade para execução de *checkpoints* periódicos, provendo um mecanismo para tolerância a falhas. Nessa segunda abordagem, caso ocorra alguma interrupção no serviço, a execução

pode ser retomada a partir do último *checkpoint* realizado (TEAM, 2009).

Segundo (TEAM, 2009), Condor oferece serviços para realização de *checkpoint* em *jobs* de processos simples, para plataformas Unix. Para utilizar esses serviços, é necessário que o usuário da plataforma realize o *link* do programa com a biblioteca específica da plataforma, `libcondorsyscall.a`, através do compilador específico da plataforma. Isso significa que para ser possível a utilização do recurso de *checkpoint*, o usuário precisa ter acesso aos objetos do programa (para efetuar o *link*) ou diretamente ao código fonte.

Os serviços de *checkpoint* são opcionais. Dessa forma, o mecanismo pode ser utilizado de acordo com a conveniência do *job* em questão. Isso significa que algumas aplicações executadas dentro do ambiente Condor podem fazer uso do mecanismo, enquanto outras não. Entretanto, ambas classes de aplicação são executadas de maneira normal e indiferenciada na plataforma Condor, fazendo uso dos mesmos recursos e funcionalidades oferecidos pela plataforma.

O processo de *checkpoint* é implementado através de chamadas à biblioteca específica da plataforma, através de *signal handlers* (TEAM, 2009). Quando Condor envia um sinal de *checkpoint* para um processo que esteja *linkado* com essa biblioteca, o *signal handler* em questão é serializado para um arquivo físico em disco ou para um *socket* da rede, onde as informações referentes ao estado do processo são armazenadas. De acordo com (TEAM, 2009), esse estado armazenado contempla as informações da pilha e também segmentos de dados, códigos existentes em bibliotecas compartilhadas e dados mapeados dentro do espaço de endereço do processo. Além disso, (TEAM, 2009) também cita que o estado dos arquivos abertos além de sinais pendentes e *signal handlers* existentes são serializados no arquivo de *checkpoint* produzido. Quando a operação de *restart* é realizada, o processo lê essas informações do disco e recupera as informações. Nesse momento, o recurso está novamente em um estado consistente, onde a execução pode ser retomada.

A abordagem de execução de *checkpoints* periódicos pode ser configurada através da expressão `PERIODIC_CHECKPOINT`, configurada para cada *pool* Condor. Essa expressão permite definir o período (uma hora, duas horas, três horas,...) em que o *checkpoint* deva ser executado. Quando o momento de realização de um *checkpoint* periódico ocorrer, o processamento suspende momentaneamente a execução do *job*, realiza a operação de *checkpoint*, produzindo o arquivo em disco, e imediatamente retoma a execução normal do processo. Condor também oferece a possibilidade de induzir a realização imediata de um *checkpoint*, através da chamada do comando `condor_ckpt`. Esse comando permite que um usuário solicite ao Condor que uma execução de *checkpoint* periódico seja imediatamente realizada (TEAM, 2009).

O ambiente Condor faz uso de uma abordagem de *commit* para manutenção das informações de *checkpoint*. Isso significa que um arquivo de *checkpoint* somente é removido do sistema quando é comprovada a existência de uma versão mais atual, sem falhas. Dessa forma, caso a realização de um *checkpoint* seja interrompida no meio, de modo que se apresente como uma versão inconsistente, as informações são imediatamente descartadas e o *checkpoint* anterior permanece em disco e será utilizado em caso de falhas.

Em (TEAM, 2009), os autores citam que a realização de um *checkpoint* pode ser momentaneamente postergada, caso o momento não seja adequado para a interrupção da execução. Isso se aplica, por exemplo, no caso do processo estar realizando uma comunicação com outro processo no momento em que o *checkpoint* deva ser realizado. Em um cenário como esse, a realização do *checkpoint* é postergada até o término da comunicação, quando então o processo pode ser suspenso e o *checkpoint* realizado.

O comportamento padrão observado pelo Condor é o de criar o arquivo de *checkpoint* em disco, de maneira local, na máquina onde o *job* foi submetido. Entretanto, (TEAM, 2009) cita que um *pool* do Condor pode ser configurado para utilizar um ou mais servidores dedicados para armazenar as informações de *checkpoint*. Quando um servidor é configurado para ser utilizado como repositório de *checkpoints*, os *jobs* executados escrevem e buscam os arquivos de *checkpoint* nessa máquina dedicada, não utilizando mais o disco local para isso.

3.2.2 BOINC

BOINC, acrônimo para *Berkeley Open Infrastucture For Network Computing* é, como visto anteriormente, uma plataforma destinada a implementação de sistemas de computação voluntária, funcionando através de uma grade computacional de dimensões mundiais, fazendo uso de computação oportunista. Dentre os diversos recursos oferecidos por essa plataforma, um deles é a possibilidade da realização de operações de *checkpoint* e *restart* (ANDERSON et al., 2006).

Segundo (ANDERSON et al., 2006), esse recurso permite que as aplicações executadas dentro do ambiente do BOINC possam ser interrompidas e posteriormente recuperadas. Por ser um sistema de computação oportunista, é fundamental que mecanismos como esse estejam disponíveis para permitir a retomada de execução do projeto, nos momentos em que o recurso esteja disponível, sem a necessidade de reiniciar as atividades do início, sempre.

Dentro das configurações de usuários do BOINC, é possível configurar um intervalo mínimo de ciclos de disco para realização de *checkpoints*. Isso significa que é possível configurar um número de intervalos de ciclos de disco mínimo que deva ser observado para que operações de *checkpoint* sejam realizadas. Isso é especialmente interessante para ambientes executados em *notebooks*, por exemplo, pois esse tipo de recurso, em períodos de inatividade, diminui a velocidade dos seus discos para garantir uma maior economia de energia. Dessa forma, a plataforma BOINC permite a realização freqüente de *checkpoints*, entretanto, o período mínimo de ciclos de disco é sempre observado (ANDERSON et al., 2006).

A API do BOINC também oferece recursos para marcação de sessões críticas. Algumas aplicações possuem determinados estados de execução que precisam ser mantidos de maneira íntegra, e a realização de um *checkpoint* em um ponto intermediário pode gerar um arquivo de *checkpoint* inconsistente e, portanto, indesejado. Para mitigar esse problema, o BOINC oferece em sua API a possibilidade de chamadas para função para marcação de regiões críticas de execução. A função `bool time_to_checkpoint();` deve ser chamada, de maneira explícita, quando a aplicação estiver em um estado consistente, onde a realização de um *checkpoint* se apresenta como desejável. É permitido que essa chamada seja realizada de maneira muito freqüente. Entretanto, apesar de ser explicitamente ativada pelo usuário programador da aplicação, a restrição do número mínimo de ciclos de disco é sempre observado e tem prioridade sob a chamada. Se a restrição de ciclos for aceita, a aplicação pode então chamar a função `checkpoint_completed();`, indicando que o arquivo de *checkpoint* foi gerado (ANDERSON et al., 2006).

3.2.3 XtremWeb

A plataforma XtremWeb, segundo (CAPPELLO et al., 2005) e (PRUITT, 2002), não possui nenhum mecanismo de *checkpoint* implementado. Isso significa dizer que em caso de falhas ou interrupções na execução de uma aplicação, a atividade deve ser retomada

sempre do início, independente do estado em que foi interrompida ou do percentual de evolução da tarefa, antes de ser paralizada. Isso pode ser visto como uma limitação da plataforma, se comparada com outros ambientes de computação oportunística, pois os resultados intermediários produzidos nunca são considerados ou armazenados em caso de interrupções.

Entretanto, apesar de não oferecer um mecanismo de *checkpoint* para ser utilizado, a plataforma disponibiliza diversos mecanismos para tolerância a falhas, por parte dos diversos componentes da plataforma. O objetivo principal desses mecanismos é possibilitar que a execução do sistema possa ser retomada, depois de qualquer falha. Não são, entretanto, utilizadas técnicas de redundância, embora seja planejado para versões futuras. Isso significa que em caso de ocorrência de falhas nos recursos, a execução da aplicação é retomada, mas o *job* é reiniciado sempre desde o início, fazendo com que as informações intermediárias produzidas até o momento, sejam perdidas (HIGAKI et al., 1997; PRUITT, 2002).

3.3 Implementações em Nível de Usuário

Os mecanismos de *checkpoint* que fazem uso de bibliotecas extras ou extensões de bibliotecas apresentam-se como uma abordagem interessante para realização de operações de *checkpoint*. Mecanismos dessa abordagem são caracterizados por estarem desacoplados do código da aplicação, oferecendo facilidades e melhorias para a manutenção da aplicação.

Essa seção apresenta algumas das principais soluções de mecanismos de *checkpoint* implementados através da utilização de bibliotecas.

3.3.1 libckpt

Libckpt é uma das primeiras implementações de bibliotecas para *checkpoint* disponibilizadas para UNIX (ROMAN, 2002). De acordo com (PLANK et al., 1995), Libckpt oferece uma proposta para habilitar mecanismos de tolerância a falhas para aplicações de longa duração. Libckpt implementa grande parte das otimizações e melhorias propostas em termos de desempenho em operações de *checkpoint*, focando especialmente na redução do tamanho dos arquivos gerados pelas operações de *checkpoint*.

Essa biblioteca permite a sua utilização de duas maneiras distintas. A primeira é realizada de maneira totalmente transparente ao programador, ou seja, as operações de *checkpoint* são realizadas nos momentos necessários, sem o conhecimento do programador, de maneira automática. Outra abordagem é através da utilização de diretivas de programação, introduzidas diretamente na aplicação, pelo programador. Nessa segunda abordagem, o programador pode, se desejável, introduzir diretivas de programação dentro do código da aplicação, habilitando, dessa forma, a realização das operações de *checkpoint* nos momentos em que forem consideradas mais importantes (PLANK et al., 1995).

Apesar da implementação do mecanismo de *checkpoint* estar contido em uma biblioteca extra, para a sua utilização é necessário uma modificação mínima na aplicação em que se deseja habilitar recursos de *checkpoint*. Para habilitar a utilização de checkpoints, a aplicação deve ser modificada, renomeando a rotina principal do programa. A alteração consiste, basicamente em converter o nome do procedimento inicial escrito em C de `main()` para `ckpt_target()`. Essa modificação permite que o libckpt possa recuperar o controle da execução da aplicação desde o seu início. Caso a aplicação seja escrita em FORTRAN, o módulo principal `PROGRAM` deve ser renomeado para `SUBROUTINE`

`ckpt_target()`. Com essa alteração, a aplicação deve obrigatoriamente ser recompilada e, de maneira estática, *linkada* com a biblioteca libckpt (PLANK et al., 1995).

A realização de *checkpoints* transparentes através da biblioteca libckpt é feita de maneira incremental (ELNOZAHY et al., 1992; FELDMAN; BROWN, 1989; WILSON; MOHER, 1989). Isso é feito através de chamadas de sistema `mprotect` para manter o controle sobre as páginas de memória modificadas dentro de um determinado endereço de memória. Essas páginas modificadas são então marcadas como *sujas* e, no momento em que uma operação de *checkpoint* é realizada, somente as páginas marcadas nesse estado são escritas dentro do arquivo de *checkpoint*. Isso faz com que o tamanho do arquivo de *checkpoint* seja reduzido, além de melhorar o desempenho, em relação ao tempo de escrita do arquivo. Apesar dessa vantagem, a utilização de *checkpoints* incrementais faz com que arquivos de checkpoints antigos não possam ser descartados, pois as informações e dados sensíveis para a recuperação da aplicação podem estar distribuídos em diversos arquivos de *checkpoint*. Para mitigar esse problema, a biblioteca libckpt oferece um programa utilitário chamado `ckpt_coa` que pode ser utilizado para concatenar diversos arquivos incrementais de *checkpoint* em apenas um arquivo (PLANK et al., 1995).

A utilização de *checkpoint* incremental, apesar de benéfica, ainda assim faz com que a execução do *checkpoint* seja sequencial. Para viabilizar isso, a biblioteca libckpt oferece a possibilidade de realizar operações de *checkpoint* de maneira assíncrona: no momento em que se torna necessária a execução de um *checkpoint*, um processo filho é criado através da utilização de uma chamada de sistema `fork()` (LéON et al., 1993; PAN; LINTON, 1989). Esse novo processo criado é então responsável por realizar o *checkpoint* em disco (PLANK et al., 1995).

As abordagens de *checkpoint* apresentadas, bem como as suas otimizações para ganhos de desempenho nos tempos de criação e persistência de informações mantêm a transparência na execução através de técnicas ou recursos que não são tipicamente visíveis para as aplicações: *signal handlers*, criação de processos filhos e utilização direta de páginas de memória. Como abordagem para melhorar ainda mais o desempenho da biblioteca, é apresentado em (PLANK et al., 1995) duas abordagens para realização de *checkpoints* direcionados através da ação direta do usuário: exclusão de memória e *checkpoint* síncrono.

Como visto, o procedimento de *checkpoint* incremental utilizado pela biblioteca libckpt utiliza as páginas de memória marcadas como *sujas* para serem persistidas nas informações de *checkpoint*. Entretanto, mesmo as páginas de memória marcadas como sujas podem conter informações que não sejam relevantes ou importantes para serem persistidas, nem mesmo necessárias durante o processo de recuperação do estado. Para possibilitar uma maior flexibilidade em termos dos conteúdos que são contemplados nos arquivos de *checkpoints*, a biblioteca libckpt permite que o programador, de maneira explícita, determine possíveis segmentos de memória que não são importantes e que não precisam, necessariamente, serem persistidos durante a execução do *checkpoint*. A utilização desse recurso pode reduzir drasticamente o tamanho do arquivo final gerado, porém deve ser utilizado com absoluto cuidado posto que a exclusão errada de determinados segmentos de memória pode causar a execução de *checkpoints* inconsistentes, fazendo que a aplicação não seja apta a recuperar seu estado consistente (PLANK et al., 1995).

Outra opção para realização de *checkpoint* é abordagem síncrona. *Checkpoint* síncrono permite que o programador da aplicação introduza diretivas de código diretamente no fonte da aplicação, indicando os pontos em que a realização de *checkpoints* é mais interessante. Essa abordagem é chamada de síncrona pois não é realizada pelas interrupções de tempo normal do libckpt e sim por diretivas específicas. Entretanto, a utilização muito

frequente das chamadas para realização de *checkpoints* pode causar perdas significativas de desempenho. Para mitigar esse problema que pode ser gerado através da frequente chamada de *checkpoints*, libckpt permite estabelecer intervalos mínimos que devem ser respeitados entre execuções de *checkpoints* (PLANK et al., 1995).

3.3.2 Libckp

Segundo (ROMAN, 2002), libckp é uma implementação para *checkpoint/restart* implementada através da utilização de bibliotecas extras projetada especificamente para tolerância a falhas. Libckp, quando comparado com as demais implementações, apresenta como diferencial o tratamento dado à manipulação de arquivos. Em (WANG et al., 1995), o autor cita que o libckp trata o conteúdo dos arquivos abertos e manipulados pela aplicação no momento do *checkpoint* como sendo parte do estado do processo.

A biblioteca libckp faz cópias dos arquivos abertos e também cópias de possíveis arquivos removidos através da chamada de sistema `unlink()`. Ainda de acordo com (WANG et al., 1995), libckp disponibiliza mecanismos para a realização de chamadas, similares a `setjmp` e `longjmp` que permitem à aplicação iniciar, de maneira explícita, o inicio de uma operação de *checkpoint* ou realizar uma operação de *rollback*, retornando para um estado previamente armazenado através de um *checkpoint*.

Objetivando detalhar, a função `setjmp()` da API $C, C++$ salva no `envbuf`[1]as informações de sistema que estejam na pilha, para posteriormente serem utilizadas pelas chamadas `longjmp()`. A função, tipicamente, tem a assinatura apresentada na listagem:

Listing 3.1: Função setjmp

```
#include <csetjmp>
int setjmp( jmp_buf envbuf );
```

Em contrapartida, a função `longjmp()` faz com que a execução de um determinado programa seja iniciada a partir do ponto onde a última chamada para a função `setjmp()` tenha sido realizada. Essa função tem a assinatura mostrada abaixo:

Listing 3.2: Função longjmp

```
#include <csetjmp>
void longjmp( jmp_buf envbuf, int status );
```

3.3.3 Libtckpt

Dieter e Lumpp apresentam em (DIETER et al., 2001) uma extensão do conceito apresentado em Libckpt para lidar com processos *multi-threads* destinado a permitir a execução de *checkpoints* em *threads* POSIX, chamado libtckpt. A solução em questão adiciona à aplicação uma nova *thread* que será responsável pela execução dos *checkpoints*. Durante as execuções de *checkpoint* e *restart*, essa *thread* adicionada é utilizada para sincronizar as demais *threads* envolvidas na aplicação e executar as eventuais chamadas de *callbacks* do usuário. As eventuais chamadas de *callback* podem ser instaladas pelos usuários da aplicação em três momentos distintos:

1. Antes do *checkpoint* ser executado;

[1] `envbuf` representa o espaço de memória existente no ambiente de execução, destinado a armazenar informações temporárias.

2. Após o *checkpoint* ser finalizado;

3. Após o *restart* ser realizado.

Libtckpt garante a consistência das informações pois todas as *threads* relacionadas com a execução da aplicação são suspensas antes do início das atividades de *checkpoint*. O estado das *threads* é obtido através do envio de um sinal `POSIX` para cada uma das *threads* envolvidas, e o *checkpoint* propriamente dito também ocorre através de sinais do mesmo tipo. Uma limitação do modelo é o fato de que o libtckpt não é capaz de processar sinais pendentes enviados se a *thread* em questão estiver bloqueada por um *mutex* ou por semáforos. Para contornar essa limitação, a biblioteca libtckpt precisa, antes de iniciar o processo de *checkpoint*, desbloquear todas as *threads* causando um sincronismo no objeto, para que então, cada *thread* possa receber e lidar com os sinais de *checkpoint* de maneira adequada. (DIETER et al., 2001) cita que existe um custo, em tempo de execução, envolvido com a atividade de manter a sincronização dos objetos mantidos por cada *thread*.

Para a utilização do mecanismo libtckpt proposto por Dieter e Lump, algumas modificações na aplicação são necessárias. O código da aplicação onde deseja-se realizar a persistência de informações deve ser modificado para incluir um *header* especial. Além da utilização de um cabeçalho especial, a aplicação precisa ser modificada para realizar a chamada de uma função especial da biblioteca libtckpt para que a inicialização dos componentes de *checkpoint* e *restart* seja realizada.

3.3.4 Score

Score é uma abordagem proposta em (TAKAHASHI et al., 2000), que possibilita a realização de operações de *checkpoint* em aplicações paralelas. O modelo de solução disponível no Score implementa as funcionalidades de *checkpoint* e *restart* nas camadas mais baixas de comunicação e nas bibliotecas de *runtime*. Essa abordagem em baixas camadas permite que aplicações paralelas possam realizar *checkpoints* sem a necessidade de sofrerem modificações no código fonte da aplicação.

Em (TAKAHASHI et al., 2000), o autor cita que além da inexistência de modificações no código fonte de aplicação, para que a utilização da biblioteca Score seja possível, apenas algumas poucas modificações são necessárias nas camadas de comunicação de alto-nível, como por exemplo, MPI. Isso significa que, para uma aplicação paralela fazer uso dos benefícios oferecidos pelo Score, apenas algumas modificações na camada de comunicação são necessárias.

Outra característica do Score apresentada em (ROMAN, 2002) é a capacidade de armazenar informações de paridade em nodos remotos. Essa informação de paridade pode ser utilizada para reconstruir um arquivo de *checkpoint* que possa estar armazenado em um nodo que apresentou falhas. Isso permite que o estado consistente de uma aplicação pode ser recuperado mesmo quando um arquivo de *checkpoint* seja perdido.

3.3.5 CoCheck

CoCheck é uma solução que permite a realização de operações de *checkpoint* em ambientes com aplicações paralelas distribuídas. Segundo (STELLNER, 1996), os ambientes de programação mais populares, como NXLib, P4, PVM ou MPI não disponibilizam em suas soluções mecanismos para a realização de *checkpoints*. A solução foi inicialmente projetada para servir como o passo inicial para permitir a migração de tarefas entre

processos. A idéia no CoCheck é realizar as operações de *checkpoint* para que as informações projetadas possam ser armazenadas e transmitidas para outros processos de modo que estes possam reiniciar as atividades no ponto em que foram previamente interrompidas.

Essa solução permite habilitar a realização de *checkpoints* em aplicações que fazem uso de troca de mensagens utilizando a MPI, objetivando facilitar eventuais operações de migração de tarefas. Em (STELLNER, 1996), o autor menciona que CoCheck tem como principal objetivo, além de permitir a realização de *checkpoints* em ambientes MPI, atingir a maior portabilidade possível. Pelo fato de também ser de interesse da solução permitir a realização de operações de *checkpoint* em outras bibliotecas, além de MPI, mostra-se interessante o projeto de uma solução que seja independente das bibliotecas específicas de MPI. Para que esses objetivos fossem possíveis de serem atingidos, a solução CoCheck foi implementada como uma camada adicional, que é executada sobre a biblioteca de comunicação (que pode ser, por exemplo, MPI).

O CoCheck é implementado tendo como base a premissa de transparência na execução de *checkpoints* (STELLNER, 1996). Isso significa que as operações de *checkpoint* devem ser executadas de maneira transparente em relação a aplicação e à camada para troca de mensagens. Dessa forma, as ações de *checkpoint* são realizadas sem que a execução da aplicação seja prejudicada.

Um dos maiores problemas endereçados pela solução em questão é o fato de que, no momento em que o *checkpoint* é executado, podem existir mensagens em trânsito, em diversos lugares como na rede física, *buffers* dos sistemas operacionais ou até mesmo nas filas das bibliotecas de troca de mensagens utilizadas. Para que o *checkpoint* possa ser corretamente executado, é importante que ele seja realizada no momento em que o sistema se apresente em um estado global consistente (STELLNER, 1996).

Um estado global consistente pode ser definido como sendo o estado em que as informações e dados estejam organizados de maneira consistente permitindo seu armazenamento e conseqüente recuperação. Aplicações paralelas podem ser definidas como sendo um conjunto de processos que cooperam entre si para a realização de uma determinada tarefa através da troca de mensagens. Nessa abordagem, cada processo, se analisado individualmente, pode ser caracterizado como sendo uma seqüencia finita e ordenada de eventos. A organização consistente desses eventos é que garante a consistência global ou não da aplicação (STELLNER, 1996).

Para exemplificar o cenário descrito acima, a figura 3.2 representa um ambiente computacional com três processos. Cada vértice da figura representa um evento e as setas indicam as mensagens trocadas durante a execução. As linhas pontilhadas S, S' e S" representam os estados globais da aplicação, isto é, a maneira como a aplicação pode ser vista sob um ponto de vista externo ao modelo. Com base nesse conceito, os cenários de estados globais onde seja possível a interrupção da execução e a correta recuperação das atividades e retomada da execução de maneira direta são chamados de estados globais consistentes.

Para garantir que o *checkpoint* seja realizado no momento adequado, onde respeita-se um estado global consistente, o CoCheck faz uso de mensagens especiais, chamadas de RM - *ready messages*. Essas mensagens especiais são utilizadas para garantir que, no momento de execução do *checkpoint*, não existam outras mensagens em rede ou *buffer* que podem prejudicar o estado global consistente, tornando inviável a realização do *checkpoint*. Cada processo participante do ambiente computacional envia a RM para os processos com os quais se comunica e estes, ao receber a notificação, repassam para os de-

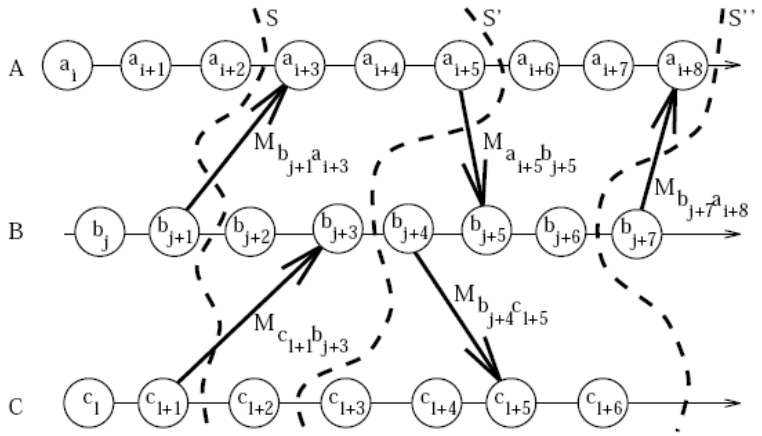


Figura 3.2: Caracterização de um Estado Global

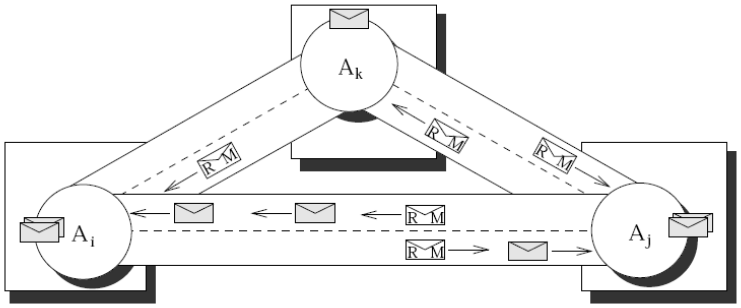


Figura 3.3: Limpando Canais de Comunicação

mais. Receber uma RM de um determinado processo significa que o processo que enviou a RM não possui mais mensagens de comunicação a ser repassada. Quando o momento em que todos os processos receberam um RM é alcançado, o estado global consistente é obtido e a operação de *checkpoint* pode ser realizada com segurança, de modo que as informações sejam passíveis de recuperação no futuro (STELLNER, 1996).

A figura 3.3 apresenta o efeito prático das trocas de mensagens RM entre os processos participantes (STELLNER, 1996). Nela é possível identificar que todas as mensagens que estão em trânsito antes do envido RM são entregues e adicionadas ao *buffer* do processo destino. Nesse mesmo fluxo, quando um determinado processo já coletou as RMs de todos os demais é possível assumir que não existem mais mensagens a serem transmitidas e as que estavam em trânsito já estão armazenadas de maneira segura em algum dos *buffers* para acesso posterior. Quando esse cenário é atingido, é possível que as operações de *checkpoint* sejam realizadas de maneira consistente, possibilitando a recuperação correta das informações, bem como habilitando que as tarefas sejam migradas entre os processos com segurança.

3.3.6 Esky

Esky é uma implementação de solução para *checkpoint/restart* focada em processos Unix, totalmente implementada em nível de usuário, sem a necessidade de utilização de *patches* ou módulos. A última versão estável da solução funciona em ambientes Linux

(2.2.x) e Solaris (2.6), independente do tipo de CPU utilizada.

O projeto inicial tinha como premissas permitir a realização de *checkpoint* para a plataforma Solaris, de maneira transparente, sem a necessidade de modificações invasivas no kernel do sistema operacional nem mesmo no código fonte da aplicação. O processo de *checkpoint* do esky é controlado através de um monitor, que intercepta as chamadas para funções realizadas pelo processo alvo. Isso é realizado através da utilização da variável de ambiente `LD_PRELOAD`. Apesar de não serem necessárias modificações no kernel do sistema operacional nem no código fonte da aplicação, a aplicação alvo deve ser *relinkada* de maneira dinâmica (GIBSON, 2009).

Entre as limitações do Esky, (GIBSON, 2009) cita que não é possível realizar *checkpoint* para programas paralelos ou *multi-threads*. Além disso, o Esky assume que o conteúdo de arquivos abertos considerados no *checkpoint* não será modificado até que o *restart* seja efetuado. Caso o conteúdo dos arquivos seja modificado no espaço de tempo entre um *checkpoint* e o *restart*, as informações serão desconsideradas.

3.3.7 Dynamit

Dynamit é uma solução que provê o balanceamento de carga dinâmico, para sistemas PVM [2]. Por ser construída com foco em sistemas PVM, a solução é também chamada *Dynamic PVM* freqüentemente representada pelo acrônimo DPVM. Nessa solução, as tarefas são migradas, de maneira transparente, entre os nodos para manter o melhor balanceamento possível. Para permitir a migração de tarefas, o DPVM possui um mecanismo de *checkpoint* implementado (OVEREINDER et al., 1996).

O DPVM consiste, basicamente, em três componentes principais:

- Escalonador, que é responsável por distribuir as tarefas de maneira equilibrada e consistente;
- Monitor, que é responsável por monitorar a carga de trabalho dos recursos do *cluster*;
- *Checkpointer*, responsável por persistir as informações produzidas e permitir que o mecanismo de migração de tarefas seja possível.

A versão atual do DPVM suporta apenas PVM, mas os autores do modelo citam em (OVEREINDER et al., 1996) que pesquisas estão sendo realizadas de modo a ampliar a cobertura para suportar MPI também. O mecanismo de *checkpoint* implementado pelo DPVM oferece, conforme citado em (OVEREINDER et al., 1996) algumas vantagens como:

- Tolerância a falhas, pois permite que *checkpoints* regulares possam ser realizados de maneira frequente;
- Possibilita a migração de processos;
- Permite a realização de *rollback* de transações.

[2] PVM é um pacote de software que permite que uma rede heterogênea de computadores possa ser programada como sendo uma única máquina paralela virtual.

Como descrito em (OVEREINDER et al., 1996), quando a operação de *checkpoint* é realizada, o espaço de endereços, juntamente com as bibliotecas compartilhadas do processo que está sendo persistido são mapeados e escritos em um arquivo em disco. Esse processo é realizado através de sinais SIGUSR1[3].

3.4 Implementações em nível de Sistema Operacional

De acordo com (LAADAN et al., 2005), abordagens de *checkpoint* realizadas diretamente através de operações do sistema operacional, oferecem uma grande transparência à aplicação, pois nenhuma modificação explícita precisa ser feita, nem mesmo recompilação da aplicação. Entretanto, são necessárias modificações extremamente invasivas no sistema operacional.

As seções abaixo apresentam algumas das soluções desenvolvidas que oferecem meios para a realização de *checkpoint* através do sistema operacional.

3.4.1 VMADump

A utilização do *Virtual Memory Area Dumper* possibilita capturar a execução de um determinado processo e copiá-lo para um nodo remoto, permitindo a continuidade de execução do mesmo. VMADump permite que o espaço de memória de um processo seja armazenado e recuperado, de maneira consistente, permitindo a interrupção e retomada de execuções (HENDRIKS, 2009).

A maioria dos programas em execução no sistema são *linkados* de maneira dinâmica (HENDRIKS, 2009). Em tempo de execução, essas aplicações utilizam `mmap` para obter cópias das várias bibliotecas, nos seus espaços de memória específico. Como as bibliotecas são sempre copiadas por inteiro para o espaço de memória, mesmo que apenas uma pequena parte dela seja de fato utilizada pela aplicação, o processo se torna custoso.

Para exemplificar, a listagem abaixo apresenta o espaço de memória do programa `sleep`, obtido de `/proc/pid/maps`:

Listing 3.3: Dump de memória do *sleep*

```
...-08049000 r-xp 00000000 03:01 288816 /bin/sleep
...-0804a000 rw-p 00000000 03:01 288816 /bin/sleep
...-40012000 r-xp 00000000 03:01 911381 /lib/ld-2.1.2.so
...-40013000 rw-p 00012000 03:01 911381 /lib/ld-2.1.2.so
...-40102000 r-xp 00000000 03:01 911434 /lib/libc-2.1.2.so
...-40106000 rw-p 000ea000 03:01 911434 /lib/libc-2.1.2.so
...-4010a000 rw-p 00000000 00:00 0
...-c0000000 rwxp fffff000 00:00 0
```

Para essa aplicação simples, são necessários $1.089.536$ *bytes*, sendo que tudo, exceto 32K, são informações referentes a bibliotecas compartilhadas.

É justamente essa grande quantidade de informações referentes a bibliotecas compartilhadas que VMADump procura evitar de copiar. (HENDRIKS, 2009) cita que a cópia de informações de bibliotecas compartilhadas pode ser evitada no momento de migração de processos para máquinas remotas se for possível garantir que as mesmas bibliotecas existem na máquina destino. VMADump armazena referências às regiões relacionadas a bibliotecas compartilhadas ao invés de armazenar a própria informação, fazendo que o tamanho da informação seja consideravelmente menor.

[3]SIGUSR1 e SIGUSR2 são sinais enviados para um determinado processo para indicar a existência de condições pré-definidas pelos usuários.

Em (HENDRIKS, 2009), o autor aponta como limitação do VMADump o fato de ser possível realizar a serialização de apenas uma *thread* em aplicações *multi-thread*. Além disso, VMADump não se preocupa com as informações referentes a descritores de arquivos, de modo que não é capaz de persistir e, por consequência, recuperar esse tipo de informação.

3.4.2 EPCKPT

EPCKPT é um mecanismo para realização de *checkpoint/restart* desenvolvido pela Universidade Federal do Rio de Janeiro. A solução foi criada tendo em mente a migração de processos, o que faz com que a preocupação em termos do tamanho das informações produzidas seja constante. Essa preocupação com o tamanho das informações de *checkpoint* é relevante quando se pensa na transferência dos dados para possibilitar a migração dos processos (PINHEIRO, 1998).

A solução é bastante similar ao esquema desenvolvido pelo VMADump. Assim como o VMADump, a solução desenvolvida pela Universidade Federal do Rio de Janeiro também é implementada no segmento estático do *kernel* do sistema operacional. Entretanto, (PINHEIRO, 1998) destaca que o EPCKPT oferece uma transparência maior nas execuções dos *checkpoints*, se comparado ao VMADump. Essa maior transparência na execução está relacionada com a utilização do PID original do processo, ao invés de utilizar uma macro, como o modelo seguido pelo VMADump (SANCHO et al., 2005). o EPCKPT, de acordo com (PINHEIRO, 1998) fornece uma série de melhorias se comparado às demais soluções para realização de *checkpoint*. Algumas dessas características são levantadas pelos autores:

- Permite a realização de *checkpoints* em diversos tipos de aplicações. Por ser uma solução desenvolvida dentro do *kernel* linux, é possível ter acesso a informações que permitem capturar informações importantes para armazenamento dos dados e futura recuperação;

- Tamanho do *checkpoint* é mínimo. Como o EPCKPT tem como principal preocupação a migração de processos, existe uma preocupação constante em relação ao tamanho físico das informações produzidas, para que seja mínimo o esforço necessário para a migração;

- *Checkpoint* transparentes de aplicações. Por ser implementado em nível de sistema operacional, toda aplicação pode ser persistida sem a necessidade de modificação, de maneira totalmente transparentes;

- *Checkpoint* de aplicações paralelas. A implementação em nível de sistema operacional permite que se tenha acesso a informações que tenham chamado operações de `fork()`, `exec()` e `mmap()`;

O início de uma operação de *checkpoint* através do EPCKPT pode ser realizada de duas maneiras: através de envio de sinais ao *kernel* ou acionado diretamente pelo usuário. Para que a segunda abordagem fosse possível, a solução desenvolvida por (PINHEIRO, 1998) disponibiliza uma série de ferramentas que possibilitam a chamada direta, por parte do usuário, das ações para realização de *checkpoints*.

3.4.3 CRAK

CRAK é uma solução de mecanismo para *checkpoint/restart* inicialmente proposta e apresentada em (ZHONG; NIEH, 2002). Essa solução foi projetada para ser adaptada como um módulo a ser adicionado ao *kernel* do sistema operacional Linux. A premissa inicial que os autores tentaram seguir com a implementação do CRAK é a de manter o índice de modificação, tanto em termos de sistema operacional no menor índice possível, e inexistente em termos de aplicação (ZHONG; NIEH, 2002). Isso significa que para a utilização do CRAK não é necessário nenhuma alteração no código da aplicação que está sendo utilizada; entretanto, o sistema operacional deve sofrer uma pequena alteração para que o módulo responsável pela realização das operações de *checkpoint* seja adicionado e instalado.

As operações realizadas pelo mecanismo CRAK de *checkpoint* podem ser divididas, de acordo com (ROMAN, 2002) em duas macro-atividades. A primeira, realizada no espaço de dados do usuário, é responsável por identificar o conjunto de processos em que se deseja aplicar as operações de *checkpoint*, isto é, realizar o levantamento de quais aplicações serão passíveis de terem seu estado persistido para uma posterior recuperação. CRAK permite que essa atividade de identificação de processos e atividades a serem persistidas seja identificada de três maneiras distintas:

1. Realização de *checkpoint* sobre apenas um processo simples;
2. Realização de *checkpoint* sobre os filhos de um determinado processo;
3. Realização de *checkpoint* sobre um processo e todos os seus filhos;

Após identificados os processos a serem persistidos durante a execução do *checkpoint*, o CRAK envia um sinal `SIGSTOP`, indicando que os processos devem ser temporariamente interrompidos. Dessa forma, com todos os processos momentaneamente suspensos, o ambiente computacional se apresenta em uma forma consistente, permitindo que as operações de *checkpoint* sejam corretamente executadas.

Nesse momento, quando os processos estão suspensos e aptos a terem suas informações persistidas, entra a segunda etapa do processo, realizada no *kernel* do sistema operacional. As informações são então persistidas em meio físico e o CRAK envia então um sinal `SIGCONT` indicando aos processos que as suas atividades normais de execução podem ser retomadas ou, opcionalmente, finalizadas se for o caso.

As informações persistidas pelo CRAK durante a execução de *checkpoints* contemplam credenciais dos processos que foram persistidos, informações sobre memória virtual, sinais ativos e sinais pendentes, o diretório corrente de trabalho além de informações sobre arquivos abertos ou em uso (ROMAN, 2002). Em (ZHONG; NIEH, 2002), é mencionado que durante a execução de um *checkpoint*, caso existam arquivos regulares sendo manipulados, as informações referentes ao nome do arquivo, modo de acesso, *flags* diversas e a posição onde o arquivo estava sendo modificado (ou acessado) é armazenada. Essas informações são posteriormente utilizadas, no processo de *restart*, permitindo que o arquivo seja reaberto e posicionado no exato ponto onde estava sendo previamente modificado.

Em (ZHONG; NIEH, 2002; ROMAN, 2002) os autores levantam como característica relevante do CRAK a possibilidade de realização de operações de *checkpoint* em múltiplos processos. Entretanto, (ROMAN, 2002) afirma que o mesmo não é possível para processos *multi-threads*, já que o CRAK não está preparado para compreender acesso a

áreas de memória compartilhada. Isso significa que através da utilização do CRAK é possível realizar operações de *checkpoint* e persistir dados referentes a aplicações com múltiplos processos, porém o mesmo não é válido para aplicações que fazem uso de compartilhamento de memória para resolução de algoritmos.

Apesar do CRAK implementar uma série de requisitos interessantes e desejáveis para um mecanismo de *checkpoint*, ele não pode ser considerado como uma solução genérica. Em (ROMAN, 2002) o autor cita justamente que essas limitações em relação à realização de *checkpoints* para processos *multi-threads* prejudica a utilização da solução de maneira ampla. Outra limitação do CRAK apontada em (ROMAN, 2002) é o fato de não permitir o registro direto, pelos usuários, de *handlers* para a realização de *checkpoints*. Em (ZHONG; NIEH, 2002) o autor também menciona a falta de possibilidade de interromper ou evitar a execução de *checkpoints*. Isso significa que quando CRAK é utilizado, não existem mecanismos para suspender ou evitar a realização de *checkpoints*.

3.4.4 Zap

ZAP foi inicialmente apresentado em (OSMAN et al., 2002) e pode ser definido como um sistema que permite a migração, de maneira transparente, de aplicações em um ambiente de rede. ZAP foi projetado como uma solução que adiciona uma fina camada de virtualização sobre o sistema operacional, tendo como principal diferencial a criação de PODs. POD é um grupo de processos organizados de maneira consistente, proporcionando uma camada virtualizada do ambiente computacional e do sistema envolvido (OSMAN et al., 2002).

Essa organização feita através de POD permite o desacoplamento da aplicação em relação ao sistema operacional, o que permite que, sendo atingido um estado consistente de execução, o processo possa ter suas informações serializadas - através de um *checkpoint* - e, se for de interesse, seja migrado para outro local. Esse desacoplamento permite que, após realizado o *checkpoint*, as informações possam ser transferidas entre diferentes máquinas, permitindo que a execução do processo seja retomada sem a perda de resultados e evitando que resíduos de informações fiquem perdidos no ambiente original de execução (OSMAN et al., 2002).

A idéia principal implementada pelo Zap é a de eliminar qualquer vínculo entre a aplicação a ser executada e o sistema operacional. Todas as informações relativas à execução, como por exemplo, ponteiros para arquivos, *sockets* e similares são contidos dentro de uma camada de abstração. No momento de execução do *checkpoint*, todas as informações que estão armazenadas nesse *namespace* especial criado pelo Zap são armazenadas e transmitidas para que possam, futuramente, serem utilizadas para a retomada de um estado consistente. Zap pode ser entendido como uma solução complementar ao CRAK: complementar pelo fato de que o Zap adiciona uma camada de virtualização sob o sistema operacional, ao contrário do CRAK (visto neste mesmo capítulo). O Zap, se entendido como uma ferramenta para migração de processos, utiliza o CRAK como mecanismo para realização de *checkpoint*.

Em (OSMAN et al., 2002) aponta como principais características do Zap a transparência em relação às aplicações. Pelo fato da solução ser implementada como uma camada de virtualização sob o sistema operacional Linux, não existe a necessidade de modificações invasivas no sistema operacional, nem mesmo modificações específicas no código da aplicação que se deseja persistir e migrar.

3.4.5 BLCR

Berkeley Lab *Checkpoint/Restart* é uma solução que faz parte da *Scalable Systems Software Suite*, desenvolvida pelo *Future Technologies Group* do *Lawrence Berkeley National Lab*, financiado pelo departamento de energia dos Estados Unidos. É uma solução *open-source* implementada sob a licença GPL (HARGROVE; DUELL, 2006). O BLCR permite a realização de *checkpoints* e foi projetado, principalmente, para aplicações HPC. É uma solução criada para atuar em nível de sistema operacional, e foi desenvolvida como um módulo que pode ser adicionado ao *kernel* Linux. Segundo (HARGROVE; DUELL, 2006), a solução foi inicialmente projetada para ser executada no Linux 2.4.x e Linux 2.6.x em arquiteturas x86 e x86-64.

De acordo com (PENG; KIAN, 2009), o BLCR é um módulo para *Kernel* Linux que permite armazenar o estado de um processo em um disco físico, possibilitando a sua futura recuperação em memória. O arquivo criado em disco, chamado de arquivo de contexto, contem todas as informações necessárias para retomar a execução normal de um processo que tenha sido interrompido. Em (PENG; KIAN, 2009; HARGROVE; DUELL, 2006) os autores citam que um arquivo de contexto pode ser criado a qualquer momento durante a execução de um processo. Com o arquivo de contexto corretamente armazenado, a execução do processo pode ser retomada a qualquer momento no futuro, até mesmo em recursos ou máquinas diferentes.

O BLCR foi criado tomando como base as funcionalidades existentes na solução VMADump, descrita nesse mesmo capítulo, sendo que adaptações específicas foram realizadas pelos autores a fim de adequar o BLCR às necessidades específicas do laboratório (PENG; KIAN, 2009). Segundo os mesmos autores, a solução é mantida como um módulo a parte, distinto do VMADump por questões de manutenção, já que os binários são diferentes.

Em (HARGROVE; DUELL, 2006), o autor apresenta algumas limitações da solução:

- BLCR não suporta a realização de *checkpoint* para grupos de processos. Isso significa que apenas processos individuais podem ser contemplados em operações de *checkpoint*;
- Para que a operação de *restart* seja realizada com sucesso o PID do processo original que está sendo recriado não pode estar em uso;
- O processo de *restart* somente será realizado com sucesso se os originais utilizados no momento da criação do arquivo de contexto bem como as bibliotecas compartilhadas utilizadas pelo processo estiverem disponíveis e na mesma versão utilizada no momento do *checkpoint*;

3.4.6 UCLiK

UCLiK (FOSTER, 2003), abreviação para *Unconstrained Checkpointing in the Linux Kernel* é um módulo de *kernel* que permite aos usuários do sistema operacional a realização de operações de *checkpoint* e *restart* de processos de maneira transparente. Foi inicialmente projetado para o *kernel* do Linux na versão 2.4.19, mas disponibiliza atualizações para versões recentes do *kernel* LINUX.

UCLiK foi construído tendo como base a solução CRAK também apresentada nesse capítulo. A principal diferença entre CRAK e UCLiK é que este último endereça melhorias em relação a manipulação de arquivos. Segundo (SANCHO et al., 2005), quando do

momento de *restart* de um processo, UCLiK é capaz de recuperar o PID original do processo, bem como todo o conteúdo do mesmo, além de identificar os arquivos que foram excluídos durante o processo.

3.4.7 CHPOX

CHPOX foi inicialmente apresentado em (SUDAKOV et al., 2007) oferece uma solução transparente para realização de *checkpoints* e a conseqüentes *restarts* de processos em *clusters* Linux. O projeto inicial do CHPOX tinha como objetivo permitir a recuperação de tarefas de longa duração, como por exemplo simulações numéricas, em caso de quebras de sistema, quedas de energia e outros problemas semelhantes.

A solução foi criada tendo como base para desenvolvimento o EPCKPT, com o diferencial de ser implementado como um módulo do *kernel* do linux. Outra diferença em relação ao EPCKPT é o fato de que o CHPOX armazena o estado dos processos envolvidos de maneira local. Isso é feito, através da criação de uma nova entrada no sistema de arquivos do sistema operacional e de um evento no *kernel* - `SIGSYS` (SUDAKOV et al., 2007).

Para que o mecanismo de *checkpoint/restart* do CHPOX possa ser utilizado, é necessário que a aplicação em questão se registre através do envio do seu PID para que seja armazenado na entrada do sistema de arquivos, mencionada acima. Com o identificador do processo em que deseja-se aplicar o *checkpoint* armazenado, a execução das operações de *checkpoints* são iniciadas através do envio de sinais `SIGSYS` para o processo.

A solução CHPOX é *SMP-Safe* e, para sua utilização, não é necessário que a aplicação seja recompilada nem mesmo *re-linkada*. Segundo (SUDAKOV et al., 2007), CHPOX oferece suporte a utilização de dados em memória virtual, assim como é capaz de persistir corretamente informações referentes a arquivos abertos. Outra característica do CHPOX citada em (SUDAKOV et al., 2007) é a possibilidade de persistir, nas informações de *checkpoint*, dados sobre os processos filhos.

O multiprocessamento simétrico é uma tecnologia que permite a um determinado sistema operacional distribuir tarefas entre dois ou mais processadores, permitindo que vários processadores compartilhem o processamento de instruções requisitadas pelo sistema. O multiprocessamento simétrico oferece um aumento linear na capacidade de processamento a cada processador adicionado. Não há necessariamente um hardware que controle este recurso, cabe ao próprio sistema operacional suportá-lo.

3.4.8 PsncR/C

PsncR/C é um mecanismo para *checkpoint/restart* proposto em (MEYER, 2003), direcionado para plataformas SUN. É implementado através de *threads* diretamente no *kernel* do sistema operacional, sendo disponível como um módulo adicional ao *kernel*. Durante a execução do *checkpoint*, todas informações referentes ao estado dos processos envolvidos serão persistidos em disco.

De acordo com (MEYER, 2003), uma nova entrada no sistema de arquivos `/proc` é criada e todas as operações de *checkpoint* são realizadas através da interface `iotcl`. Por não possuir preocupações específicas relacionadas com a otimização de dados para reduzir o tamanho físico em disco das informações de *checkpoint*, todas as bibliotecas compartilhadas, códigos utilizados e arquivos abertos são invariavelmente incluídos nos *checkpoints*.

3.5 Persistência Ortogonal

Um sistema persistente, segundo (ATKINSON; MORRISON, 1995), lida com as informações que estão armazenadas em meio físico permanente, como sendo uma extensão estável da memória volátil. Nessa visão, os objetos podem ser dinamicamente manipulados, modificados e alocados em memória, mas seus respectivos dados são persistidos entre as execuções de um determinado programa. Isso significa que, em um sistema persistente, mesmo as informações armazenadas somente em memória são mantidas entre as execuções de uma determinada aplicação.

Os princípios de transparência e ortogonalidade são amplamente discutidos e considerados como fundamentais no projeto de uma linguagem de programação persistente (ATKINSON; MORRISON, 1995). Com esses princípios básicos observados, a abstração de persistência pode ser disponibilizada, em seu total potencial.

O conceito de transparência, nesse contexto, significa que, sob a perspectiva do programador do sistema, o acesso aos objetos persistentes não requer a escrita de comandos ou instruções específicas. Isso faz com que a transferência dos objetos do meio de persistência estável para a memória seja completamente transparente, em termos de desenvolvimento (ATKINSON; MORRISON, 1995). Com isso, aplicações que manipulam objetos potencialmente persistentes são muito semelhantes com aplicações que somente manipulam objetos transientes, em memória. Em termos de funcionalidade, o conceito de transparência permite que, ao invés de instruções explícitas de leitura e escrita, o compilador ou sistema de *run-time* envolvido persiste os dados, de maneira automática, em *cache*, conforme demanda da aplicação.

Por ortogonalidade devemos entender como a possibilidade de tornar a linguagem de programação persistente, com o mínimo de modificações ou alterações na sintaxe já existente, mantendo a semântica já definida (MOSS; HOSKING, 1996). Em um sistema persistente ortogonal, essa premissa é observada e todo e qualquer objeto criado passa a ser persistente, de maneira automática, a partir do momento em que é referenciado por um objeto já persistente (ATKINSON; MORRISON, 1995). Com esse conceito de ortogonalidade, os programadores precisam de poucos ajustes em termos de conhecimento da linguagem para poderem fazer uso dos benefícios da persistência.

Em (ATKINSON; MORRISON, 1995), os autores citam que um mecanismo comum para habilitar o conceito de persistência ortogonal em uma linguagem de programação, é tratando o mecanismo de persistência estável como uma extensão das informações que são alocadas dinamicamente em *heap*. Isso permite que os dados transientes e os dados persistentes sejam tratados de maneira uniforme e transparente.

O conceito de persistência ortogonal foi desenvolvido e melhorado ao longo dos anos de pesquisa, na comunidade acadêmica. Esse conceito estabelece três principios básicos que caracterizam a ortogonalidade, que são independentes da linguagem de programação em que sejam implementados:

- *Ortogonalidade de tipo*: a persistência deve ser possível e estar disponível para todo e qualquer dado, independente do seu tipo.

- *Persistência Transitiva* [4]: o ciclo de vida de todos os objetos é determinado pelo alcance, a partir de um determinado objeto raiz.

- *Independência de Persistência*: a persistência dos dados deve ser independente de como o programa manipula essas informações.

A aplicação desses três princípios produz um mecanismo de persistência ortogonal. Segundo (DEARLE et al., 2009) a violação de qualquer uma dessas premissas acarreta o aumento da complexidade, que um sistema persistente procura sempre evitar.

Diversas linguagens de programação oferecem implementações ou versões que disponibilizam o conceito de persistência ortogonal. Abaixo são apresentadas algumas dessas linguagens, junto com uma breve descrição.

3.5.1 PS-Algol

Foi a primeira linguagem de programação a disponibilizar o conceito de persistência ortogonal (DEARLE et al., 2009). PS-Algol é uma solução que adiciona um pequeno número de funções específicas na linguagem S-Algol, da qual deriva.

O exemplo de código apresentado abaixo, retirado de (ATKINSON et al., 1990), apresenta o funcionamento da linguagem PS-Algol, em termos de sua persistência ortogonal:

Listing 3.4: Persistência ortogonal com PS-Algol

```
structure person(string name, phone; pntr addr)
structure address(int no ; string street, town)
let db = open.database("addr.db", "write")

if db is error.record
   do { write "Can't open database"; abort }

let table = s.lookup("addr.table", db)
let p = person("al", 3250,
            address(76, "North St", "St Andrews"))

s.enter("al", table, p)
commit
```

Nesse exemplo, a base de dados `addr.db` é aberta e um *objeto* `person` é criado. Note que também é criado um *objeto* `address` que é associado ao objeto `person` previamente criado. No momento em que a instrução de `commit`, é chamada, de maneira transparente e ortogonal, ambos objetos são armazenados para posterior consulta.

3.5.2 Napier88

Napier88 é uma implementação que tenta explorar os limites da persistência ortogonal através da incorporação de todo o ambiente de suporte da linguagem de programação (ATKINSON et al., 1990). A linguagem Napier88 foi inicialmente concebida por Ron Morrison e Malcolm Atkinson, com projeto e primeira implementação feita por Fred Brown, Richard Connor, Alan Dearle e Ron Morrison (ATKINSON et al., 1990).

O modelo proposto pelo Napier88 consiste de uma linguagem de programação e seu ambiente persistente. A implicação do conceito de persistência ortogonal implica no fato de não ser necessário converter ou mover, de maneira explícita, os dados de memória para um meio físico persistente, seja para dados temporários ou permanentes (ATKINSON et al., 1990). O modelo de persistência do Napier88 é obtido através da utilização de um objeto raiz persistente, onde todos os objetos derivados deste assumem a propriedade de persistência também (BROWN, 1988).

O sistema Napier88 foi projetado como uma arquitetura em camadas, composta basicamente por três níveis: o compilador, a máquina abstrata de persistência (PAM, do inglês

[4] A persistência transitiva é um conceito que deriva do *Principle of Persistence Identification*, definido em (ATKINSON; MORRISON, 1995).

Persistent Abstract Machine) e o componente para armazenamento persistente (BROWN, 1988). Todas essas camadas definidas pela arquitetura são virtuais, no sentido de não ser específico o modo como sejam implementadas. Isso significa que as camadas do Napier88 podem ser implementadas de maneira totalmente independente ou agrupada, de acordo com os requisitos de desempenho que estejam associados ao modelo (BROWN, 1988; ATKINSON et al., 1990, 1996).

3.5.3 Arjuna

O foco do Arjuna é oferecer um conjunto de ferramentas para possibilitar a criação de aplicações distribuídas tolerantes a falhas. De acordo com (PARRINGTON et al., 1995; SHRIVASTAVA et al., 1991), isso é obtido através da utilização de objetos persistentes através do conceito de ortogonalidade, possibilitando a realização de transações atômicas aninhadas. Por transações atômicas o autor se refere à propriedade de atomicidade, associada às transações.

Para tornar possível a persistência e recuperação dos objetos, um *snapshot* é tirado antes das modificações serem realizadas dentro do escopo da transação atômica, conforme descrito em (PARRINGTON et al., 1995). Com esse mecanismo, antes de iniciar a execução de qualquer trecho crítico (transação atômica) o estado geral de execução é armazenado, sendo substituído ao final da transação caso esta tenha sido efetivada com sucesso. Em caso de ocorrência de falhas, o estado armazenado antes da transação é recuperado e utilizado como base para o re-início das atividades de processamento.

Uma aplicação Arjuna consiste de uma ou mais transações atômicas criadas para controlar uma coleção de operações realizadas em objetos da linguagem. Esses objetos são, por definição, uma instância de um tipo abstrato de dados (uma classe C++) que pode ser persistido de modo a continuar existindo mesmo após o término de execução da aplicação. Em (SHRIVASTAVA et al., 1991) o autor menciona que a partir da garantia de que toda manipulação de objetos seja realizada por meio de transações atômicas, é possível afirmar que a integridade dos objetos e das informações será garantida em caso de falhas ocorridas.

3.5.4 Persistent Java

PJama (ATKINSON et al., 1996) é uma das implementações de ortogonalidade da plataforma Java. Nessa implementação, o programador da aplicação utiliza uma API específica para realizar o mapeamento dos objetos com `strings` em um mapa persistente. Dessa forma, todos os objetos transientes que estejam associados com um registro nesse mapa, passam a ser persistidos.

A implementação do PJama, segundo (ATKINSON et al., 1996), não traz alterações à linguagem de programação em si, nem mesmo necessita de modificações no compilador e nas bibliotecas padrão. Entretanto, alterações na máquina virtual são necessárias para permitir a manipulação dos objetos da memória para o ambiente de persistência, e vice-versa.

Máquinas virtuais podem ser definidas como um sistema computacional que é implementado em termos de software, usualmente sendo executado dentro de um computador real. Nos termos que serão utilizados aqui, máquinas virtuais devem ser entendidas como as soluções responsáveis por encapsular a execução de uma aplicação (um programa Java ou C#, por exemplo).

A máquina virtual do Java (JVM) é um conjunto de programas e estruturas de dados que usam um modelo de máquina virtual para a execução de outros programas ou

scripts, desenvolvidos em uma linguagem intermediária chamada de *bytecode*. Como outro exemplo, o CLR é a máquina virtual oferecida pela plataforma .NET. É semelhante à implementação da JVM, sendo responsável pela execução de aplicações em linguagem intermediária CIL, também chamada de MSIL.

3.5.5 Outras Soluções para Persistência Ortogonal

Além das diversas linguagens de programação que oferecem suporte à persistência ortogonal, sistemas gerenciadores de banco de dados orientados a objetos também podem ser listados aqui. Como exemplo, pode-se citar DB4O, que pode ser utilizado nas plataformas Java e .NET. DB4O não requer o mapeamento explícito entre os dados persistentes e transientes, de modo que a persistência das informações fica transparente ao usuário (CORPORATION, 2009), como mostrado no trecho de código abaixo:

Listing 3.5: Persistência ortogonal utilizando DB4O

```
ObjectContainer db=Db4o.openFile(Util.DB4OFILENAME);
try {
   Person al = new Person("al", 49);
   db.set(al);
}
finally {
   db.close();
}
```

Nesse exemplo, um objeto `Person` é persistido, de maneira transparente. Note que essa transparência faz com que os objetos armazenados na base de dados sejam *POJOs* sem interface extras, extensões de classes ou anotações.

Outra solução, apresentada em 2002 é *Java Data Objects* (PROCESS, 2009). JDO oferece um mecanismo para persistência de objetos java, sem a necessidade de utilizar linguagens de acesso a dados, como SQL. Utilizando JDO, os objetos podem ser armazenados em bancos de dados relacionais, banco de dados orientado a objetos, arquivos XML ou outras tecnologias semelhantes.

A API de persistência do Java (MICROSYSTEMS, 2009) foi projetada para atuar dentro e fora do *container* J2EE, criando um modelo persistente para objetos Java. Essa API também tinha como objetivo reduzir a complexidade do JDO, retirando a necessidade do mapeamento das informações em arquivos de XML de configuração.

A Microsoft, reconhecendo a existência de problemas para criação e utilização de consultas em `strings` dentro do código das aplicações, criou o *Language-Integrated Query*, LINQ (KULKARNI et al., 2009). LINQ tem a abordagem de tratar a consulta em dados relacionais como sendo apenas uma das opções para busca. Isso significa que, através de LINQ, realizar buscas em base de dados relacionais, base de dados orientadas a objetos ou coleções de objetos mantidas em memória são equivalentes. Dessa forma, é possível escrever aplicações que fazem buscas em memória ou em base de dados relacionais, da mesma forma. O trecho de código abaixo apresenta um exemplo de busca de *Pessoas* em uma determinada coleção:

Listing 3.6: Busca através de LINQ

```
static void doquery( Person[] people )
{
   IEnumerable<Address> result = from p in people
                where p.age == 49
                select p.address;
```

```
   foreach (Address item in result)
      Console.WriteLine(item.getTown());
}
```

3.6 Outras Implementações

Existe uma quantidade significante de soluções para *checkpoint* de aplicações de longa duração. Além das implementações mais populares, citadas e detalhadas nesse capítulo, uma grande quantidade de soluções, de menor expressão ou utilização, foram deixadas de fora dessa análise. Dentro dessa linha, é possível destacar a participação de grandes empresas privadas que participam ativamente do desenvolvimento de soluções para *checkpoint*. A SGI disponibiliza uma implementação de mecanismo de *checkpoint/restart* para o sistema operacional IRIX. Essa solução é de código fechado e é disponibilizada como parte do *kernel* do IRIX. A Cray implementou um mecanismo de *checkpoint/restart* para os sistemas operacionais Unicos e Unicos/mk, também de código fechado. Acredita-se que essas soluções sejam os produtos mais robustos que existem atualmente. Entretanto, não é possível apresentar maiores detalhes das soluções pois não existem documentação ou informações públicas disponíveis, descrevendo detalhes sobre a implementação.

O centro de supercomputação de Pittsburgh projetou e implementou uma solução para *checkpoint* e *restart* em nível de usuário para sistemas TCS. Esse sistema de checkpoint permite a recuperação automática de *jobs* em caso de falhas nas máquinas ou em períodos de manutenção de equipamentos. Segundo (STONE et al., 2009) a solução pode ser utilizada para maximizar a utilização dos equipamentos através do armazenamento períodico das informações intermediárias produzidas. Foi projetado para ser acessível para aplicações C, C++ e Fortran. A solução criada pelos laboratórios do centro de supercomputação de Pittsburgh foi inicialmente projetado para suprir três requisitos básicos: permitir a recuperação de *jobs* em casos de falhas, ser escalável e exercer efeito mínimo no tempo de execução do *job* (STONE et al., 2009).

Carothers e Szymanski descrevem em (CAROTHERS; SZYMANSKI, 2002) um mecanismo de *checkpoint* implementado em nível de sistema para programas *multi-threads* executados no sistema operacional Linux. Nessa proposta, uma nova chamada de sistema operacional é introduzida, sendo responsável por realizar uma cópia da aplicação, utilizando um mecanismo *copy-on-write* semelhante aos utilizados pela chamada *fork*. O novo processo criado pela chamada mencionada permanece em memória, não sendo realizada a persistência em disco. Apesar de ter a vantagem de ser bastante rápida, oferece a limitação de não tolerar falhas como *reboot* ou quedas de energia, visto que as informações são mantidas totalmente em memória.

StarFish é um ambiente para execução de aplicações MPI-2 (dinâmicas e estáticas) em *clusters* de *workstations*. Trata-se de um ambiente eficiente, tolerante a falhas, com alta disponibilidade que permite comunicação em grupo de maneira flexível e portável (AGBARIA; FRIEDMAN, 1999). Esse ambiente também oferece um mecanismo de *checkpoint/restart*, com a limitação de estar disponível apenas para aplicações escritas em Ocaml. O mecanismo também necessita intervenções do usuário para sincronização dos processos, o que faz com que os resultados de *checkpoint* produzidos não sejam totalmente consistentes.

Existe também um grande número de soluções que propõem a realização de *checkpoints* e migração de processos através de abordagens orientadas a objetos. Dentro dessa classe de solução podemos citar Abacus (AMIRI et al., 1999), Emerald (O'CONNOR,

2009), Globus (FOSTER; KESSELMAN, 1996), Legion (GRIMSHAW et al., 1997) e Rover (JOSEPH et al., 1997). Esses sistemas são usualmente definidos e projetados como *middlewares* ou até mesmo como linguagens de programação específicas, o que requer ações explícitas do programador para controlar e realizar as operações de *checkpoint* e migração.

Kasbekar et al (KASBEKAR et al., 1999) apresentam um mecanismo para realizar *checkpoints* seletivos em aplicações orientadas a objetos que possuam *threads* independentes. No modelo proposto pelos autores, um sub-sistema é criado e nele é mantido, de maneira centralizada, um grafo de dependência que oferece informações referentes às *threads* envolvidas e os objetos que são acessados por cada uma das *threads*. Esse grafo é percorrido, no momento em que a ação de *checkpoint* é realizada, para identificar qual o conjunto de *threads*, e seus respectivos objetos, que devem ser incluídos no *checkpoint*. O modelo lida com falhas de maneira individual a cada *thread*; quando uma *thread* falha, somente ela e os objetos envolvidos no seu processamento são consideradas para o processo de *roll-back*, sendo que as demais *threads* continuam o processamento normal de execução. Essa abordagem faz com que cada *thread* tenha que atualizar constantemente seu estado de execução no grafo.

3.7 Considerações Finais

Este capítulo introduziu e detalhou alguns dos principais mecanismos de *checkpoint* disponíveis. Foram apresentadas as soluções existentes para as principais plataformas de computação voluntária. Nesse tipo de ambiente, os mecanismos de *checkpoint* são fundamentais para permitir o armazenamento e recuperação de resultados intermediários produzidos pelos recursos voluntários. Também foram apresentadas soluções de *checkpoint* que se apresentam como soluções para tolerância a falhas. Dentre essas propostas, foram detalhadas as soluções que fazem uso de bibliotecas extras acopladas à aplicação-alvo, implementações que utilizam recursos do sistema operacional, além de outras soluções para *checkpoint* que propõem modificações nas máquinas virtuais.

O mecanismo de *checkpoint* que é detalhado nessa dissertação, oferece uma proposta de solução que pode ser comparada a alguns dos mecanismos aqui apresentados. O modelo proposto faz uso de pequenas modificações na aplicação em que se deseja executar no ambiente voluntário. Apesar de ser necessária essa pequena modificação, não se faz necessário realizar nenhuma modificação invasiva em termos de sistema operacional, nem mesmo necessidades especiais para compilar a aplicação. Ao classificarmos o modelo proposto por essa dissertação, é correto defini-lo como uma solução realizada em nível de usuário, sem modificações ou alterações no sistema operacional ou máquina virtual.

4 DISKLESS CHECKPOINT

Mecanismos de *Checkpoint* operam através do armazenamento das informações necessárias para a retomada de execução de um processo ou programa. Abordagens de *checkpoint* baseadas em disco armazenam esses dados em meios de armazenamento estáveis, como discos. Essa operação de armazenamento se apresenta, na maioria das vezes, como um gargalo de execução para os ambientes distribuídos, em termos de performance. Isso é ainda mais grave em ambientes distribuídos que possuem mais processadores do que disco, pois a disputa pela escrita, por parte dos processadores, é mais intensa, gerando um aumento ainda maior na performance para execução do *checkpoint* (SILVA; SILVA, 1998a; PLANK et al., 1998a; BOWMANM, 2006).

Esse custo elevado, associado com os gargalos de desempenho para execução acarreta um elevado custo para essa abordagem, o que limita a quantidade de *checkpoints* que podem ser executados. Além disso, o armazenamento das informações em disco acarreta um *overhead* de rede e disco que fazem com que o processamento normal da aplicação de interesse aumente em termos de tempo (SILVA; SILVA, 1998a; PLANK et al., 1998a; BOWMANM, 2006).

O presente capítulo mostra uma abordagem para realização de *checkpoints* que utiliza recursos voláteis, como memória, para armazenamento das informações intermediárias produzidas. Além de apresentar, conceitualmente, a abordagem de *diskless checkpoint*, são mostradas as principais abordagens utilizadas dentro desse conceito.

4.1 Fundamentação Teórica

O objetivo principal do *diskless checkpoint* é tornar o processamento mais rápido, eliminar o mencionado *overhead* de rede e reduzir o tempo para armazenamento das informações através da utilização, de maneira eficiente, da memória disponível (SILVA; SILVA, 1998a; PLANK et al., 1998a; BOWMANM, 2006). *Diskless Checkpoint* pode ser definido como uma técnica para armazenar o estado de um processo de longa duração em um ambiente distribuído sem o armazenamento de informações em meios estáveis (DONGARRA et al., 2008). Com essa abordagem, cada processador envolvido no processamento armazena uma cópia do seu estado localmente, na maioria das vezes, em memória. Dessa forma, em caso de ocorrência de falhas, cada processador é capaz de utilizar as informações armazenadas e recuperar a sua execução a partir do último ponto estável armazenado. Através da não utilização de meios estáveis, como disco rígido, essa abordagem de *checkpoint* remove o principal *overhead* dos mecanismos de *checkpoint* tradicionais em sistemas distribuídos (PLANK et al., 1998a).

Diskless Checkpointing é uma abordagem atrativa que oferece meios confiáveis e de alto-desempenho para armazenamento de informações (LU, 2005). Nesse modelo,

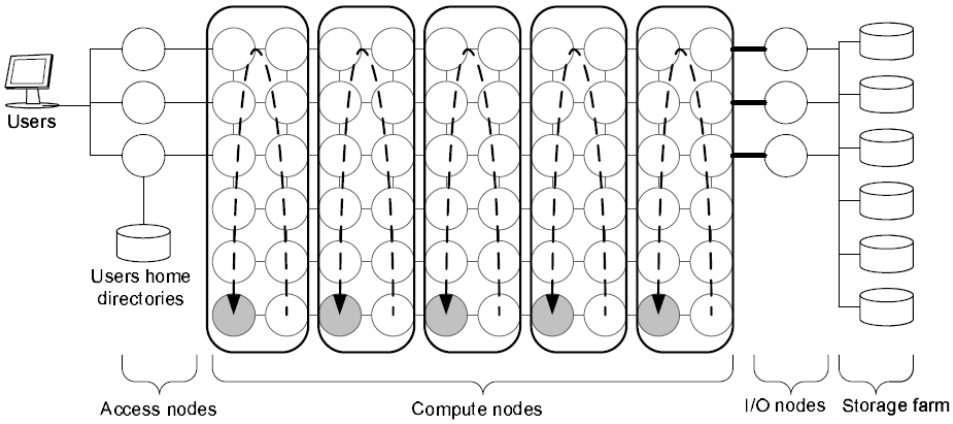


Figura 4.1: Estrutura de *Diskless Checkpoint* para grandes sistemas

o *checkpoint* é inicialmente escrito em memória, que é muito mais rápido que a escrita em disco. Para garantir a integridade dos dados armazenados, códigos de redudância (*bits* de paridade ou códigos *Reed-Solomon* são calculados e armazenados em processadores reservas). Esses processadores extra também assumem o papel de nodos computacionais se o processador que acusou uma determinada falha não consiga recuperar seu estado estável para reiniciar o processamento normal em um curto período de tempo - essa situação acontece tipicamente em casos de falhas de *hardware* (PATTERSON et al., 1988). Sistemas de grande porte que utilizam o mecanismo de *checkpoint* em memória, tendem a se organizar de modo particionado e em pequenos grupos. Cada pequeno grupo formado é então capaz de cuidar dos seus cálculos para redundância e de suas recuperações em caso de falhas. A figura 4.1 representa graficamente a organização de um sistema de grande porte que utilize *diskless checkpoint* como mecanismo para persistência das informações em caso de falhas. Nesse exemplo, os nodos computacionais são particionados em cinco grupos distintos onde, para cada grupo, existe a presença de um nodo extra. Os dados de *checkpoint* produzidos são escritos em memória local e as informações de redundância são calculadas e armazenadas no nodo extra.

Para tornar a abordagem de *diskless checkpoint* a mais eficiente possível, ela pode ser implementada em nível de aplicação, ao invés de ser utilizada em nível de sistema (JAMES S. PLANK YOUNGBAE KIM, 1997). De acordo com (DONGARRA et al., 2008), entre as vantagens de utilizar um mecanismo de *checkpoint* em nível de aplicação, pode-se destacar:

- *Checkpointing* em nível de aplicação pode ser configurado e disponibilizado em pontos específicos de sincronização no programa, atingindo, de maneira automática, *checkpoints* consistentes;
- O tamanho do *checkpoint* (em termos de memória) pode ser minimizado pois os desenvolvedores da aplicação podem restringir os dados a serem persistidos de acordo com as necessidades específicas;
- *Checkpoints* realizados em nível de aplicação podem ser implementados de modo que a operação de recuperação do estado consistente pode ser realizado em ambientes heterogêneos.

A idéia de substituir a mídia magnética por memória como mecanismo preferido para armazenamento tem suas origens na década de 80. Mas somente tornou-se efetivamente

utilizada nos dias atuais graças à diminuição considerável dos preços desse tipo de recurso, bem como com a ampla adoção dessa tecnologia em dispositivos móveis como câmeras digitais, PDAs, *pendrives* e cartões de memória *flash*.

Operações de IO sempre se apresentaram como fatores críticos, fazendo com que pesquisadores, desde a década de 80, pensem em soluções para armazenar todas as informações das bases de dados em memória (GARCIA-MOLINA; SALEM, 1992). Entretanto, a utilização de discos rígidos não pode ser descartada e os dados ou as informações de *logs* transacionais devem ser persistidos em meio físico. Outro fator que se apresenta como limitante para a ampla utilização de memória volátil como meio de armazenamento era o preço da mesma: apenas algumas poucas aplicações de telecomunicações e outras de tempo real ofereciam uma quantidade suficiente para o armazenamento de todos os dados e informações.

No contexto de sistemas operacionais, muitos trabalhos foram desenvolvidos em sistemas *in-memory*. *Non-volatile* RAM (NVRAM) tem sido empregada em diversas soluções para mitigar o tráfego de escrita gerado pelas aplicações, bem como para oferecer maior confiabilidade e segurança (BAKER et al., 1992; WU; ZWAENEPOEL, 1994; CHEN et al., 1996; WANG et al., 2002).

Dispositivos móveis também fazem uso de sistemas gerenciadores de arquivos em disco. Como exemplo, memória *flash* é um tipo de NVRAM que é amplamente utilizado em *hand-helds*. A latência de acesso nesse tipo de dispositivo é mais lento que DRAM, especialmente para operações de escrita. Isso se deve pelo fato de que o conteúdo precisa ser apagado antes que novas informações possam ser escritas. Esse processo de limpeza deve ser executado de maneira *block-wise* e leva, em média, aproximadamente 0.6 a 0.8 segundos por bloco. Como resultado, esse tipo de sistema baseado em memória *flash* deve fazer uso de algoritmos e técnicas para substituição de blocos. Como exemplos de sistemas desse tipo, as soluções (WU; ZWAENEPOEL, 1994; KAWAGUCHI et al., 1995) podem ser citadas.

Semelhante aos sistemas de arquivos em memória, é a utilização de discos RAM, que é suportada por diversos sistemas operacionais (TANENBAUM, 2007). Um disco RAM pode ser definido como uma porção pré-alocada da memória principal que simula um disco físico tradicional. Essa solução é usualmente implementada através de um programa residente em memória ou um *driver* de *kernel* do sistema operacional. Discos RAM são normalmente utilizados para armazenar dados intermediários que serão descartados em curtos espaços de tempo, como, por exemplo, arquivos temporários.

Tendo esse conceito definido, *Diskless Checkpointing* pode ser pensado como uma organização RAID de um grupo de discos RAM. Dentre os vários trabalhos desenvolvidos dentro dessa abordagem, Veer et al, em (SILVA et al., 1994) implementou uma solução de *checkpoint* espelhado em redes *Transputer*, onde as informações de *checkpoint* são armazenadas em nodos vizinhos, que se apresentam como recursos parceiros para armazenamento das informações, como forma de *backup*. Plank (PLANK; LI, 1994a) propôs a utilização de códigos de redundância para melhorar a recuperação de nodos de execução que venham a sofrer falhas. Chiueh (CHIUEH; DENG, 1996a) realizou testes em ambientes de *checkpoint* espelhados e com informações de paridade em máquinas massivamente paralelas e obteve resultados que mostraram que o espelhamento torna a solução até dez vezes mais rápida, com um custo de cerca de duas vezes mais memória. Plank, em (PLANK et al., 1998a), estende a abordagem de paridade para utilizar *Reed-Solomon* para suportar falhar arbitrárias.

Em tempos recentes, a utilização de mecanismos para *checkpoint* em memória volta-

ram a despertar interesse em pesquisadores ao redor do mundo. Em (CHEN et al., 1994), os autores utilizaram *checkpoint* com espelhamento em conjunto com uma biblioteca MPI tolerante a falhas para oferecer maior confiabilidade na execução de programas paralelos. Zheng (ZHENG et al., 2004) modificou o conceito de espelhamento através do armazenamento das informações de *checkpoint* em dois nodos diferentes, com dedicação exclusiva, fazendo que a resistência a falhas múltiplas fosse aumentado. Em (SUNDERAM et al., 2004), Chen também estudou uma nova classe de códigos de redundância, onde as informações são tratadas como um *stream* de *bits* sem considerar informações semânticas. Na abordagem de Chen, os códigos de redundância são definidos sobre números reais ou números complexos.

É importante destacar que as abordagens de *diskless checkpoint* não se apresentam como uma solução que substitui os mecanismos de *checkpoint* tradicionais: ela serve como um complemento. A implementação de uma solução de *checkpoint* em memória, segundo (PLANK et al., 1998a) é útil para armazenamento de informações intermediárias produzidas ou dados temporários. Essa abordagem oferece uma maneira rápida de armazenamento, para quando as informações a serem armazenadas mudam com freqüência.

O presente capítulo apresenta os principais esquemas de *checkpoint* em memória que podem ser utilizados sem nenhuma modificação de *hardware*: *neighbor-based*, *parity-based*, *checksum-Based*, *weighted-checksum-based*, *parity-based* e *encoding*.

4.2 Neighbor-Based Checkpointing

Em mecanismos de *checkpoint* baseados em vizinhos (do inglês *neighbor-based*), cada processador do modelo de computação participa de um ambiente onde um processador é definido e eleito como vizinho. Nessa abordagem, além de manter as informações em sua memória, cada processador armazena uma cópia das informações de seu *checkpoint* no processador vizinho. Dessa forma, quando falhas acontecem para um determinado processador, as informações de *checkpoint*, que serão utilizadas para retomar a execução, podem ser buscadas na memória do processador vizinho.

O *overhead* de desempenho da execução de *diskless checkpoint* é usualmente muito baixo. O *checkpoint* é armazenado somente em dois lugares: no processador original, onde o processamento está sendo realizado e no processador vizinho, que serve como um local de armazenamento de *backup*. A recuperação das informações envolve somente os processadores envolvidos na falha e seus vizinhos. Não existem comunicações globais entre todos os processadores do ambiente, nem mesmo são necessários cálculos para codificação e decodificação das informações armazenadas no *checkpoint*.

Baseado em como o processador vizinho é eleito, a abordagem de *neighbor-based checkpointing* pode se estabelecer de acordo com três esquemas distintos: *mirroring*, *Ring Neighbor* e *Pair Neighbor* (DONGARRA et al., 2008).

4.2.1 Mirroring

O esquema de espelhamento, do mecanismo de *checkpoint* baseado em utilização de vizinhos, foi originalmente proposto em (PLANK et al., 1998a). Nesse esquema, supondo a existência de n processadores disponíveis e dedicados para computação, outros n processadores devem ser alocados e dedicados exclusivamente para participarem como vizinhos. Com essa abordagem, o *i-ésimo* processador armazena uma cópia das suas informações de *checkpoint* no *i-ésimo* processador de resguarde que foi alocado para atuar como vizinho.

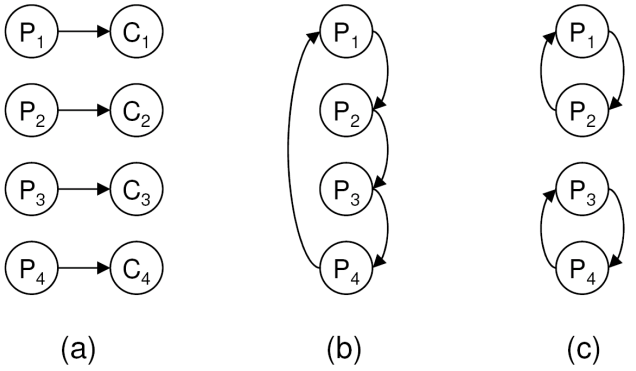


Figura 4.2: Esquemas de *checkpoint* baseado em vizinhos

A figura 4.2(a) apresenta um modelo de grafo que demonstra o esquema de espelhamento: cada processador P_1 possui um processador vizinho C_1 dedicado exclusivamente para manter as informações de *checkpoint* produzidas.

O modelo apresentado está projetado para suportar até *n* falhas de processadores (onde *n* é o número de processadores disponíveis no modelo). Entretanto, a solução não consegue ser tolerante a falhas em casos onde ocorra um erro simultâneo no processador e seu vizinho. Se as premissas que estabelecem que as falhas de cada processador são independentes (isto é, não ocorrem simultaneamente no processador e no seu vizinho eleito) e que as falhas são distribuídas de maneira idêntica forem verdadeiras, a probabilidade de que o modelo de espelhamento se comporte dentro do esperado para falhas em *k* processadores pode ser estabelecida pela seguinte equação:

$$\frac{C_n^k 2^k}{C_{2n}^k}$$

Quando o número de falhas *k* for significativamente menor que o número de processadores *n*, a probabilidade de que o modelo se comporte bem a *k* falhas é muito próximo de 1. A principal desvantagem que se observa nesse modelo de espelhamento é a necessidade da existência de *n* processadores adicionais que devem ser dedicados exclusivamente para o armazenamento das cópias de *checkpoint*, tornando-os, portanto, recursos que não podem ser utilizados para computação e produção de resultados.

4.2.2 Ring Neighbor

Em (SILVA; SILVA, 1998b), o esquema de vizinhança em anel foi discutido por Silva *et al*. Nesse modelo, não existe a necessidade de uso de processadores adicionais para a manutenção de cópias de segurança dos *checkpoints* produzidos. Os processadores participantes do modelo de computação são armazenados em forma de um anel virtual, como demonstra a figura 4.2(b).

Cada processador envia uma cópia das suas informações de *checkpoint* para o processador que o segue no anel virtual. Dessa forma, cada processador participante tem, em memória, informações referentes a dois *checkpoints*: o seu *checkpoint* local, com as informações produzidas localmente, além das informações de *checkpoint* produzidas pelo seu vizinho no anel.

Esse modelo de vizinhança em anel é capaz de tolerar uma variação de 1 até $\lfloor \frac{n}{2} \rfloor$ falhas de processadores, em um ambiente composto de *n* processadores, dependendo de

como é a distribuição das falhas dentro do anel. Se comparado com o modelo de espelhamento, a vantagem de organizar a solução em anel é que não se faz necessária a utilização de processadores redundantes para armazenamento das cópias de segurança. Entretanto, como desvantagem, podemos citar que no modelo em anel, cada processador participante precisa armazenar duas cópias de *checkpoints* em memória, fato que pode prejudicar, mesmo que pouco, a capacidade de processamento e armazenamento dos processadores envolvidos.

4.2.3 Pair Neighbor

Outra abordagem para os mecanismos de *checkpoint* baseado na utilização de vizinhos é o da vizinhança pareada. Nesse modelo, os processadores participantes do modelo computacional devem ser organizados em duplas (para esse modelo, deve-se assumir que exista um número par de processadores participantes). Dessa forma, os dois processadores do par estabelecido são vizinhos entre si, e cada processador envia uma cópia das suas informações de *checkpoint* para o outro, como mostra a figura 4.2(c).

Da mesma forma que a abordagem em anel, nesse modelo não existe a necessidade da utilização de processadores redundantes e cada processador possui, em memória, informações referentes ao seu *checkpoint* local e também informações de *checkpoint* do processador pareado. Entretanto, se comparado com o esquema de vizinhança em anel, o grau de tolerância a eventuais falhas desse modelo é melhor. Assim como o modelo de espelhamento, se assumirmos que a falha ocorra de maneira independente e identicamente distribuída, isto é, não ocorra para ambos processadores de um mesmo par, a probabilidade de que o modelo resista a *k* falhas em um modelo computacional com *n* processadores é:

$$\frac{C_{\frac{n}{2}}^{k} 2^k}{C_{2n}^{k}}$$

4.3 Parity-Based Checkpointing

A execução de *checkpoints* através de mecanismos de paridade consiste basicamente de duas etapas: cada processador executa um *checkpoint* das informações em memória local e, em uma segunda etapa, realiza a codificação dessas informações armazenando-as em um processador dedicado para armazenar as informações de *encoding* de todos os processadores participantes do ambiente de processamento (HUNG, 1998).

Em caso de ocorrência de falhas em algum dos processadores envolvidos, todos os processadores do ambiente computacional em questão, que não estiverem envolvidos na falha, executam uma ação de *rollback* utilizando como estado consistente a informação de *checkpoint* existente em suas memórias locais. A partir dessa recuperação, um processador P_i qualquer é utilizado como substituto do processador que apresentou falhas e este executa uma ação de *rollback* utilizando a informação calculada a partir dos dados dos processadores funcionais e da informação de *encoding* do processador dedicado.

Essa abordagem para execução de *checkpoint* pode ser feita através de *checkpoint* local, *checkpoint* através de *encoding* ou da combinação das duas abordagens. As seções seguintes detalham essas possibilidades.

4.3.1 Local Checkpointing

Como o nome sugere, essa abordagem de *checkpoint* local determina que os processadores envolvidos devam realizar o armazenamento de suas informações específicas de *checkpoint* em memória local. Além disso, por se tratar de uma abordagem *diskless*, não existe a necessidade de armazenamento em disco. Essa abordagem pode ser implementada de três maneiras distintas: simples, incremental e *forked*.

4.3.1.1 Simple Checkpointing

A forma mais simples para realização de *checkpoint* é através da manutenção de uma cópia local em memória do espaço de endereços e dos registradores. Nessa forma mais simples de *diskless checkpoint*, se a execução de uma operação de *rollback* é necessária, as informações dos endereços e registradores, mantidas em memória, são recuperadas e aplicadas ao processador, possibilitando o retorno a um estado de execução consistente.

É importante que seja destacado que essa abordagem não oferece tolerância a falhas no próprio processador. Isso significa que o processador que apresentar qualquer falha de execução não será capaz de recuperar seu próprio estado a partir do último *checkpoint*, pois não terá mais as informações disponíveis em memória. Essa abordagem somente é válida para permitir que um processador execute as operações de *rollback* para as últimas informações consistentes armazenadas em caso de falhas de outros processadores.

O principal problema dessa abordagem é a quantidade de memória necessária para sua correta implementação. Pra que seja possível realizar a recuperação completa das informações e recuperar a correta execução no ambiente computacional, cada processador participante deve manter, em sua memória local, uma cópia de toda a aplicação (PLANK et al., 1998b; HUNG, 1998).

4.3.1.2 Incremental Checkpointing

Nssa abordagem, o mecanismo de proteção de páginas de memória do sistema operacional é utilizado para observar, em tempo de execução do *checkpoint*, quais páginas de memória sofreram modificações desde o último *checkpoint*. As informações identificadas através desse processo são, portanto, as informações que devem ser armazenadas. Para identificar os dados que foram modificados, existem duas classes de técnicas que podem ser utilizadas: baseada em páginas e baseadas em *hash*. Para a primeira técnica, é necessário disponibilidade de memória e suporte do sistema operacional para a manipulação dos *bits* de proteção a fim de identificar as páginas de memória modificadas. Na segunda técnica, a memória é particionada em blocos e uma função de *hash* é executada sobre cada um dos blocos. A partir das informações geradas pela execução da função, é possível encontrar os blocos modificados.

Entretanto, essa abordagem incremental apresenta um problema relacionado à quantidade de informações desatualizadas que deve ser mantida em memória. Em abordagens de *checkpoint* não-incrementais, apenas as informações de *checkpoint* mais recentes devem ser mantidas para possibilitar a recuperação das informações, ou seja, informações mais antigas podem ser descartadas. Em contraponto, em abordagens incrementais, as informações antigas de *checkpoint* não podem ser descartadas pois as informações das páginas de memória são distribuídas e espalhadas por diversos *checkpoints*. Dessa forma, o tamanho cumulativo das informações de *checkpoint* incremental tende a crescer a medida do tempo, já que vários valores atualizados podem ser armazenados para uma mesma página de memória (PLANK et al., 1995)

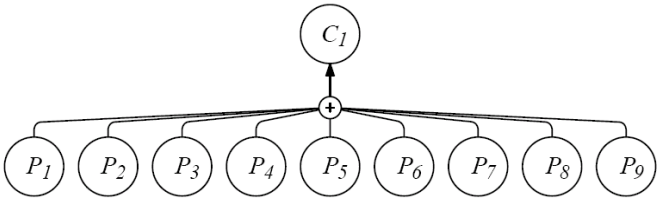


Figura 4.3: Esquema de Paridade RAID nível 5

4.3.1.3 Forked Checkpoiting

Essa implementação de *checkpoint* é realizada através de um mecanismo de clonagem de processo, como por exemplo, a chamada de sistema *fork* existente em sistemas Unix. O processo clonado, resultante da execução dessa chamada de sistema funcionará como o *diskless checkpoint*. Essa informação é, portanto, armazenada em memória e tem sua principal utilidade para que os processadores que não apresentarem falhas ou erros de execução possam realizar um *rollback* e retomar o seu estado consistente após a ocorrência de falhas em outros processadores (PLANK et al., 1998b; HUNG, 1998).

Esse modelo é funcional para que todos os processadores participantes do ambiente computacional possam realizar operações de *rollback* e estejam aptos a recuperar um estado consistente de execução. Entretanto, não se apresenta como uma solução tolerante a falhas para o próprio processador que apresenta o erro. Isto significa que não é uma solução tolerante a falhas para o próprio processador em estado consistente (PLANK et al., 1998b; HUNG, 1998).

4.3.2 Encoding Checkpointing

Através de mecanismos de *checkpointing* que façam uso de alguma abordagem de codificação, um número m de processadores extra são utilizados de maneira dedicada para a realização dessa codificação. Esses processadores são utilizados para realizar a codificação dos *checkpoints* para todos os processadores participantes do ambiente computacional além de serem úteis para recuperação dessas informações, que servem para que os processadores que venham a falhar possam ter seu estado consistente re-calculado e recuperado. Existe um grande número de abordagens de codificação para *checkpoint*, dentre elas podemos citar paridade (RAID nível 5) e *Reed-Solomon* que são apresentadas em detalhes abaixo.

4.3.2.1 Parity (RAID nível 5)

Sendo F o tempo médio que antecede uma falha em um equipamento qualquer, então o tempo médio para que uma falha qualquer ocorra em um ambiente computacional com n equipamentos é $\frac{F}{n}$. Para ambientes computacionais com valores de n pequenos e com equipamentos razoavelmente confiáveis, a utilização de apenas um processador dedicado para a execução de *checksums* é suficiente para oferecer uma boa tolerância a falhas (PATTERSON et al., 1988; GIBSON, 1992; CHEN et al., 1994). A essa configuração de ambiente (que pode ser representado graficamente através da figura 4.3) é dado o nome de RAID de nível 5 e a técnica de codificação utilizada é chamada de paridade de $n + 1$.

Nessa abordagem de codificação existe um processador dedicado para *checkpoint* $m = 1$ que é responsável por codificar a paridade para cada *checkpoint* da aplicação. Sendo b_i^j o *byte* que representa o j-ésimo *byte* do processador i, então o j-ésimo *byte* do

processador de *checkpoint* será:

$$b^j_{ckp} = b^j_1 \oplus b^j_2 \oplus \ldots \oplus b^j_n$$

Em caso de ocorrência de falhas, de acordo com (PLANK et al., 1998b), o estado consistente de execução pode ser recuperado da seguinte forma: um processador substituto qualquer deve ser eleito para assumir o lugar de processamento do que apresentou falha. Esse novo processador pode ser o processador responsável por *checkpoint*, um processador extra que estava alocado como reserva ou até mesmo o próprio processador que inicialmente havia falhado, caso seja uma ocorrência de falha transiente. Esse novo processador calcula então a informação de *checkpoint* do processador falho através da utilização das informações de paridade dos processadores que *não* sofreram falhas e do *encoding* existente no processador de *checkpoint*. Em representação matemática, suponha que o processador i sofra uma falha qualquer, o *checkpoint* pode então ser reconstruído com:

$$b^j_i = b^j_i \oplus \ldots \oplus b^j_{i-1} \oplus b^j_{i+1} \oplus \ldots \oplus b^j_n \oplus b^j_{ckp}$$

Quando o processador substituído finalizou o cálculo do *checkpoint* do processador que falhou, todos os processadores da aplicação estão aptos a realizar a operação de *roll-back* no último *checkpoint* e o processamento computacional pode seguir a partir das informações recuperadas.

4.3.2.2 Reed-Solomon

Em 1960, Irving Reed e Gus Solomon publicaram um artigo no *Journal of the Society for Industrial and Applied Mathematics*, (REED; SOLOMON, 1960), descrevendo um novo modelo de codificação para correção de códigos, chamado de *Reed-Solomon*. Essa solução tem uma grande utilidade, sendo utilizada em diversas aplicações atualmente, desde *disc players* até aplicações mais complexas. *Reed-Solomon*, doravante chamados R-S, são códigos não-binários cíclicos com símbolos, formado por seqüências de m *bits*, onde m é um número inteiro positivo qualquer, maior ou igual a 2.

Códigos $R - S(n, k)$ para símbolos m-bit existem para todo n e k onde

$$0 < k < n < 2^m + 2$$

onde k representa os símbolos de dados que estão sendo codificados e n é o número total de símbolos do bloco. Para a codificação mais convencional de $R - S(n, k)$, tem-se:

$$(n, k) = (2^m - 1, 2^m - 1 - 2t)$$

onde t é a capacidade de correção de erros do símbolo e $n - k = 2t$ representa o número de símbolos de paridade do modelo. Uma versão estendida do código R-S pode ser criada com, no máximo, $n = 2m$ ou $n = 2m + 1$.

Em códigos R-S, para seqüências de informações não-binárias, a distância entre duas palavras codificadas é definida como o número de símbolos onde as seqüências se diferem. Para códigos R-S, a distância mínima, de acordo com (GALLAGER, 1968) é:

$$d_{min} = n - k + 1$$

Essa codificação é capaz de proporcionar, de acordo com os autores de (SKLAR, 1988), a correção de qualquer combinação de t (ou menos) erros, onde t pode ser expresso por:

$$t = \lceil \frac{d_{min} - 1}{2} \rceil = \lceil \frac{n - k}{2} \rceil$$

A equação acima ilustra o fato de que, para códigos R-S, para a correção de t erros não são necessários mais do que $2t$ símbolos de paridade.

4.3.3 Integração de Local e Encoding Checkpointings

Um dos fatores mais importantes para mecanismos de *checkpoint* é o desempenho, em termos de tempo de execução (CHIUEH; DENG, 1996b; SILVA; SILVA, 1999). O *overhead*, nas abordagens de integração de local e *encoding checkpointings*, surge nas atividades de cálculo e envio das informações calculadas.

Em alguns mecanismos de codificação mencionados aqui, existe um gargalo pois os processadores destinados a execução de *checkpoints* são alvos de uma série de mensagens, enviadas por todos os processadores participantes do ambiente computacional. Além desse gargalo em termos da quantidade de mensagens recebidas, esses processadores são responsáveis por todo o cálculo de codificação. Para mitigar esses gargalos e melhorar um pouco o desempenho, uma solução é a utilização de um método FAN-IN[1]. Nessa abordagem, os processadores do ambiente computacional realizam os cálculos de *encoding* em $\log n$ passos e enviam o resultado final para os processadores de *checkpoint*.

Outra abordagem para mitigar o tempo de execução, melhorar o desempenho e reduzir a comunicação na rede é reduzir o tamanho das mensagens trocadas. Através da utilização de uma abordagem semelhante ao *checkpoint* incremental, apenas as páginas de memória modificadas desde o último *checkpoint* são enviadas para os processadores. Essas mudanças, chamadas de *diff*, são enviadas aos processadores de *checkpoint* que realiza uma operação de XOR com as informações já existentes. Ainda, o método pode ser melhorado através da compressão das informações de *diff* antes do envio da mensagem (HUNG, 1998).

4.4 Checksum-Based Checkpointing

O *checkpoint* baseado em *checksums* é uma versão modificada do mecanismo de *checkpoint* baseado em paridade (apresentado anteriormente, nesse mesmo capítulo) proposto e apresentado em (PLANK; LI, 1994b). Nesse modelo, ao invés da utilização de paridade, uma proposta de adição numérica de pontos-flutuantes é utilizada para codificar todas as informações dos dados do *checkpoint* local. Através do envio das informações codificadas aos processadores dedicados para *checkpoints*, essa proposta adiciona um *overhead* de memória muito menor ao modelo computacional, se comparado com o mecanismo de *checkpoint* baseado em vizinhos. Entretanto, as atividades de cálculo e envio das informações codificadas introduzem um *overhead* de desempenho maior, se comparado ao mesmo esquema. De acordo com o modo como o cálculo do *encoding* é realizado, o modelo de *checkpoint* baseado em *checksum* pode ser definido em duas categorias: esquema de *checksum* básico e esquema de *checksum* de uma dimensão.

[1]FAN-IN pode ser definido como o número de entradas padrão de uma determinada entrada lógica.

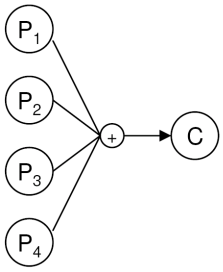


Figura 4.4: Esquema baseado em *checksum*

4.4.1 Esquema baseado em Soma de Verificação Básico

No esquema de soma de verificação básico, do inglês *basic checksum*, supondo a existência de um programa sendo executado em um ambiente computacional com N processadores disponíveis, é necessária a existência de um processador de ordem $N + 1$, denominado processador de *checksums*. Durante toda a atividade computacional do modelo em questão, informações de *checkpoint* consistentes são mantidas, em memória, pelos N processadores participantes. Além disso, a soma de verificação das informações de *checkpoint* para os N processadores é mantida no processador de *checksums*, como mostra a figura 4.4:

Assumindo como verdade as premissas que estabelecem que P_i é a informação de *checkpoint* do processador de ordem i e que C é a soma de verificação do *checkpoint* local no processador de *checksums*, se olharmos os dados de *checkpoint* produzidos como sendo um *array* de números reais, então a codificação do *checkpoint* estabelece a identidade

$$P_1 + \ldots + P_n = C$$

entre os dados do *checkpoint* do processador P_i e a soma de verificação C no processador de *checksums*. Em caso de ocorrência de uma falha em qualquer um dos processadores P_n, a identidade estabelecida acima se torna em uma equação com um termo desconhecido. Desse modo, através da resolução da equação, é possível reconstruir as informações faltantes.

Devido as operações aritméticas de ponto flutuante utilizadas nos mecanismos de *checkpoint* e *recovery*, podem ocorrer erros de arredondamento, onde o valor calculado aproximado difere do valor real matemático. Embora esse problema esteja presente, ele pode ser considerado praticamente nulo, de acordo com (PLANK; LI, 1994b), pois a codificação dos dados do *checkpoint* envolve apenas operações de adição e a decodificação envolve apenas adição e subtração.

O esquema baseado em soma de verificação básico consegue lidar com apenas uma falha. Caso duas ou mais falhas ocorram, a identidade acima passa a ter mais de um termo desconhecido, fazendo com que sua solução não seja possível, o que torna o sistema impossível de ser recuperado. Entretanto, esse princípio básico de recuperação através de *checksum* pode ser utilizado para construir mecanismos mais elaborados, como o esquema de *uma dimensão*, que permite que a solução sobreviva a múltiplas falhas.

4.4.2 Modelo de *checksum* de Uma Dimensão

Supondo que um determinado programa está sendo executado em um ambiente computacional com mn processadores disponíveis, o modelo sugere que o ambiente seja particionado em m grupos com n processadores em cada um, tendo um processador de *check-*

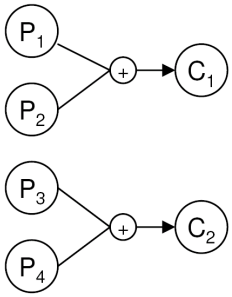


Figura 4.5: Esquema de Checksum de Uma Dimensão

sum dedicado para cada grupo. Nesse esquema, representado pela figura 4.5, para cada grupo, a soma de verificação deve ser calculada conforme descrito no esquema simples, apresentado acima.

A vantagem desse modelo, segundo (DONGARRA et al., 2008), é que os *checkpoints* são realizados em um subgrupo específico de processadores, de modo que a codificação das informações, para cada grupo, possa ser realizada em paralelo. Se comparado com o modelo básico de soma de verificação, o desempenho desse esquema é usualmente melhor.

Se assumirmos como verdadeiras as premissas que estabelecem que as falhas sejam independentes e identicamente distribuídas, a probabilidade de que o esquema de verificação de soma de uma dimensão permaneça funcional em caso de $k(k < m)$ falhas é

$$\frac{C_m^k (n+1)^k}{C_{m(n+1)}^k}$$

4.4.3 Modelo de *checksum* de Duas Dimensões

O modelo de *checksum* de duas dimensões, representado pela figura 4.6 é uma extensão do modelo de *checksum* de uma dimensão. Nessa abordagem, os processadores participantes do modelo são organizados logicamente em um *grid* de duas dimensões, com a presença de um processador para *checkpoint* em cada coluna e linha do grid. Cada um dos processadores de *checkpoint* é então responsável por realizar o *encoding* dos processadores da sua linha ou coluna. O modelo de paridade em duas dimensões necessita de $m \geq \sqrt{n}$ processadores de *checkpoint*, fazendo com que esse esquema esteja apto a suportar falhas de um processador P_n qualquer, para uma determinada linha ou coluna da matriz.

4.5 Checkpoint baseado em Weighted-Checksum

De acordo com (DONGARRA et al., 2008), o modelo de soma de verificação com utilização de pesos é uma extensão do esquema de *checkpoint* baseado em *checksum* apresentando anteriormente, com o diferencial de estar apto a responder, de maneira consistente, a falhas múltiplas que possam ocorrer em diversos padrões, utilizando para isso um número mínimo de processadores para redundância (processadores dedicados exclusivamente para armazenamento de informações de *checkpoint*). De acordo com (PLANK, 1997), essa abordagem também pode ser interpretada como uma implementação do esquema de codificação *Reed-Solomon* no campo de números reais.

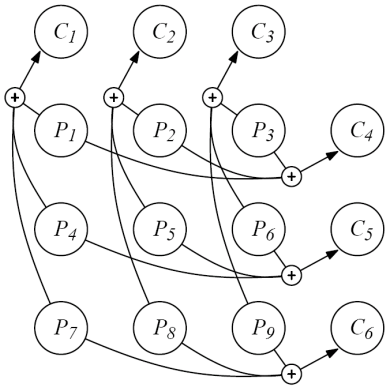


Figura 4.6: Esquema de Checksum de Duas Dimensões

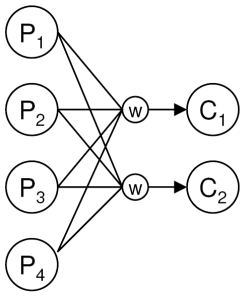


Figura 4.7: Modelo Básico de Weighted Checksum

Nessa abordagem, cada processador participante do ambiente de computação mantém, em memória local, uma cópia das informações de *checkpoint*. Além disso, são estabelecidas m equações de igualdade, através do armazenamento dos pesos associados ao *checkpoint* local para os m processadores. Desse modo, quando ocorrem f falhas, onde $f \leq m$, as m igualdades se transformam em m equações com f termos desconhecidos. Através da escolha correta dos pesos, é possível recuperar os dados perdidos nas f falhas através da resolução das m equações geradas.

4.5.1 The Basic Weighted Checksum Scheme

Para definir o modelo básico de *checksum* com pesos associados, supõe-se a existência de n processadores utilizados para a produção de resultados no ambiente computacional e assume-se que as informações de *checkpoint* produzidas pelo i-ésimo processador é P_i. Para ser possível reconstruir os dados perdidos nos processadores que acusaram falhas, é necessário que outros m processadores sejam dedicados para manter m somas de verificações com pesos associados (*weighted checksums*) dos dados de *checkpoint*, como apresentado na figura 4.7. Dessa forma, a soma de verificação com peso associado P_j no j-ésimo processador de *checksums* pode ser calculada através da fórmula:

$$\begin{cases} a_{11}P_1 + \ldots + a_{1n}P_n & = C_1 \\ & \vdots \\ a_{m1}P_1 + \ldots + a_{mn}P_n & = C_m \end{cases}$$

onde $a_ij, i = 1, 2, \ldots, mj = 1, 2, \ldots, n$ representa o peso que deve ser escolhido. Sendo definido que $A = (a_{ij}{}_{mn})$ é a matriz do modelo de *checksum* com pesos associados.

Supondo que k processadores participantes do modelo de computação e $m - h$ processadores destinados a operações de *checkpoint* entraram em um estado de falha, então $n - k$ processadores de computação e h processadores de *checkpoint* sobreviveram às falhas. Se avaliarmos as informações dos processadores que apresentaram falhas como termos desconhecidos, a partir da equação acima, teremos m equações com $m - (h - k)$ termos desconhecidos.

Se $k > h$, então teremos menos equações geradas do que termos desconhecidos, fazendo com que se tenha mais de uma solução possível o que, por consequencia, não permita a recuperação dos processadores com falhas. Entretanto, se $k < h$, teremos mais equações do que termos desconhecidos. Se a matriz A for corretamente definida, uma solução única para a equação acima pode ser encontrada e as informações perdidas pelos processadores que acusaram falhas podem ser recuperadas através da resolução das equações.

Em termos gerais, se tomarmos como verdadeiras as premissas:

- Os processadores $j_1, j_2, \ldots, j_k$ participante do modelo computacional falharam e os processadores $j_{k+1}, j_{k+2}, \ldots, j_n$ permanecem disponíveis, sem apresentarem falhas;
- Os processadores de *checkpoint* $i_1, i_2, \ldots, i_h$ permanecem disponíveis sem a ocorrência de falhas enquanto os processadores de *checkpoint* $i_{h+1}, i_{h+2}, \ldots, i_m$ caíram em decorrência de falhas.

A partir dessas premissas, a equação acima apresenta $P_{j1}, \ldots, P_{jk}$ e $C_{i_{h+1}}, \ldots, C_im$ como termos desconhecidos depois da ocorrência de falhas. Dessa forma, a equação pode ser re-estruturada, do seguinte modo:

$$\begin{cases} a_{i_1 j_1} P_{j_1} + \ldots + a_{i_1 j_k} P_{jk} & = C_{i_1} - \sum_{t=k+1}^{n} ai_1 j_t P_{j_t} \\ & \vdots \\ a_{i_h j_1} P_{j_1} + \ldots + a_{i_h j_k} P_{jk} & = C_{i_h} - \sum_{t=k+1}^{n} ai_h j_t P_{j_t} \end{cases}$$

e

$$\begin{cases} C_{i_{h+1}} & = a_{i_{h+1}1} P_1 + \ldots + a_{i_{h+1}n} P_n \\ & \vdots \\ C_{i_m} & = a_{i_m 1} P_1 + \ldots + a_{i_m n} P_n \end{cases}$$

Sendo A_r a matriz de coeficientes do sistema linear exposto pela equação acima. Se A_r possui um posto completo de colunas[2], então $P_{j1}, \ldots, P_{jk}$ pode ser resolvido através da equação acima e $C_{ih+1}, \ldots, C_{im}$ pode ser recuperado substituindo $P_j 1, \ldots, P_j k$.

O fato de estar ou não apto a recuperar os dados perdidos nos processadores que acusaram falhas depende diretamente da matriz A_r possuir ou não um posto completo de colunas. Entretanto, para a equação apresentada acima, A_r pode ser qualquer sub-matriz de A, dependendo de como os processadores que acusaram falhas estão distribuídos. Se qualquer sub-matriz quadrada de A e existem mais de m processadores com falhas, então é garantido que A_r terá um posto completo de coluna. Dessa forma, para estar apto a se recuperar de até m processadores com falhas, a matriz de *checkpoint* A deve satisfazer qualquer sub-matriz quadrada de A.

[2]O posto (ou Rank) de uma matriz é o número de linhas linearmente independentes igual ao número de colunas linearmente independentes. O posto de uma matriz mxn é estabelecido por, no máximo, $minm, n$. Da matriz que possui o posto maior possível é dito que esta tem um posto completo. Dessa forma, da matriz mxn que possui um posto igual a n é dito que esta possui um posto completo de colunas.

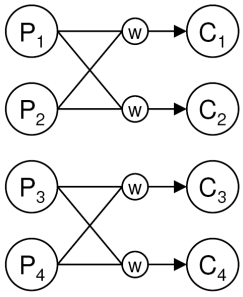


Figura 4.8: Esquema de Checksum com Pessos Associados de Uma Dimensão

4.5.2 Two Dimensional Weighted Checksum Scheme

Esse modelo funciona de maneira semelhante ao modelo *One Dimensional Checksum Scheme* apresentado anteriormente, mas de modo a utilizar pesos associados. Para entender o esquema de *checksum* de uma dimensão com pesos associados devemos assumir que o programa está sendo executado em um ambiente computacional com mn processadores. O particionamento desses processadores deve ser feito em m grupos de n processadores cada. Além desse particionamento, outros k processadores devem ser dedicados para cálculo de *checksum*. Nessa configuração, cada grupo realiza operações de *checkpoint* utilizando o modelo básico de *weighted checksum* apresentado anteriormente, gerando um ambiente semelhante ao representado na figura 4.8.

Nessa configuração, o modelo consegue sobreviver a k falhas de processadores em cada grupo. A vantagem dessa abordagem, segundo (DONGARRA et al., 2008) está no fato de que o *checkpoint* é localizado em um sub-grupo específico de processadores, de modo que o *encoding* de cada grupo pode ser realizado de maneira paralela por todos os grupos do ambiente. Ainda segundo (DONGARRA et al., 2008), esse modelo, se comparado com o modelo básico de *weighted checksum*, apresenta ganhos consideráveis de desempenho na execução.

4.6 Avaliação da Abordagem

A abordagem de *checkpoint* em memória, se comparada com abordagens mais tradicionais de armazenamento em disco, apresenta algumas vantagens que justificam a sua utilização. Entretanto, junto com as vantagens, a abordagem também apresenta algumas limitações, que devem ser conhecidas e avaliadas para justificar a decisão de sua utilização ou não.

4.6.1 Vantagens

A abordagem de realizar *checkpoints* em memória em detrimento da realização em disco, de acordo com (SILVA; SILVA, 1998a; PLANK et al., 1998a; BOWMANM, 2006) traz uma série de benefícios, dentre eles:

- Latência e tempo de recuperação da informação reduzidos, se comparada com o *checkpoint* tradicional em disco;
- Reduz a necessidade de utilização de recursos compartilhados;
- Usa, de maneira eficiente, a memória disponível para armazenamento das informações úteis para o *checkpoint*;

- Não necessita adição de recursos de *hardware* extras para armazenar e realizar o *checkpoint*;
- Oferece um tempo de execução do *checkpoint* inferior, quando comparado com o *checkpoint* em disco.

4.6.2 Limitações

Da mesma forma, a utilização dessa abordagem também pode oferecer algumas limitações. Ainda de acordo com (SILVA; SILVA, 1998a), (PLANK et al., 1998a) e (BOWMANM, 2006), as principais limitações que a abordagem de *checkpoint* em memória pode trazer são:

- Utilização de *encoding*, memória, CPU e rede podem trazer pequenos *overheads* para a execução;
- Quando utilizado sozinho, o *checkpoint* em memória não consegue eliminar a ocorrência de falhas gerais de execução da máquina, pois o conteúdo da memória pode ser eliminado.

Em (BOWMANM, 2006), o autor cita que a abordagem de *checkpoint* em memória possui uma cobertura de falhas relativamente inferior, se comparada com o *checkpoint* em disco, já que nenhum dos componentes de um ambiente de *diskless-checkpoint* consegue sobreviver a uma falha genérica de execução, onde o recurso computacional deva ser desligado e reiniciado.

4.7 Considerações Finais

Nesse capítulo, foram apresentados os principais conceitos relacionados a *checkpoint* em memória, que não fazem uso de modificações de hardware. Nesse contexto, foram apresentadas as abordagens para realização de *checkpoints* em memória baseada em vizinhos, paridade, *checksum*, *checksum* com pesos e através de codificação.

O modelo proposto neste trabalho, tem por objetivo utilizar uma combinação de abordagens para realização de *checkpoints*. O meio de persistência principal é em memória, pelo fato deste se apresentar como um meio mais eficiente e rápido, se comparado com a persistência normal em disco. Essa rapidez é especialmente importante por se tratar de um mecanismo de *checkpoint* que tem como premissa realizar suas operações de maneira rápida. Entretanto, realizar as operações de *checkpoint* somente em memória pode ser arriscado, posto que manter as informações produzidas somente em memória pode se tornar um problema quando o recurso em questão sofrer alguma falha ou for reiniciado. Para mitigar esse problema, o modelo proposto engloba também recursos para realização de *checkpoints* periódicos em disco, através da realização de *snapshots* da execução. O protótipo implementado nessa dissertação não utiliza nenhuma técnica específica de *checkpoint* em memória. Para o mecanismo descrito aqui, todas as manipulações de objetos realizadas pela aplicação-alvo são encapsuladas dentro de um conceito de transação que é persistida de modo que, em caso de falhas ou interrupções, possam ser recuperadas para reutilização.

5 MODELO PROPOSTO

O termo *checkpoint* pode ser definido como a ação de armazenar o estado de uma aplicação em execução para sua recuperação posterior, evitando perda de informações processadas bem como garantindo a estabilidade de execução. O estudo apresentado nessa dissertação detalhou duas das principais abordagens para *checkpoint*: armazenamento baseado em disco e armazenamento baseado em memória, apresentando as principais implementações existentes, dentro dessas abordagens.

As conclusões derivadas desse estudo mostraram que ambas as abordagens possuem vantagens e desvantagens, se mostrando como soluções complementares. Com base nisso, este capítulo apresenta a idéia de um modelo para realização de *checkpoints* que é fundamentado em uma abordagem que unifica os dois mecanismos apresentados: *diskless-checkpoint*, realizado através da utilização do conceito de prevalência de objetos, combinado com a persistência periódica em disco, realizada em intervalos de tempo pré-definidos e de acordo com o recurso envolvido.

Esse capítulo também detalha a implementação de um protótipo para validar o modelo proposto. Além do detalhamento do modelo e das características do protótipo, também são apresentados alguns resultados preliminares, obtidos a partir da execução do modelo dentro de um cenário proposto.

5.1 Origem da Proposta

A proposta apresentada e estudada nessa dissertação surgiu da idéia da possibilidade de utilizar MFPs como recursos úteis para ambientes de computação voluntária. Uma MFP pode ser definida como uma máquina que incorpora a funcionalidade de múltiplos equipamentos em apenas um, oferecendo recursos e funcionalidades de impressora, scanner, foto-copiadora, fax e e-mail.

Para um correto entendimento da aplicação do modelo de *checkpoint* nesse tipo de recurso, é importante destacar os principais pontos em termos de hardware e software na arquitetura interna de uma MFP.

5.1.1 Arquitetura Interna de uma MFP

A arquitetura interna de uma MFP pode ser comparada à arquitetura de um desktop tradicional, pois apresentam características semelhantes em termos de hardware e software, embora tenham objetivos distintos. Como apresentado abaixo, grande parte das MFPs disponíveis no mercado atualmente possuem configurações suficientes para permitir a execução de aplicações voluntárias.

5.1.1.1 Hardware

Em termos de hardware, impressoras multifuncionais podem ser comparadas com componentes periféricos convencionais, mas que são capazes de funcionar normalmente sem a utilização de computadores propriamente ditos. Nesses termos, é correto afirmar que MFPs podem ser classificadas como tipos de computadores. De acordo com (HEWLETT-PACKARD, 2009), as impressoras multifuncionais modernas contém memória, um ou mais processadores e, freqüentemente, algum tipo de armazenamento físico, como disco físico ou memória flash.

A tabela 5.1 apresenta alguns exemplos de especificações de multifuncionais retiradas de (HEWLETT-PACKARD, 2009):

Tabela 5.1: Configurações de MFPs

Modelo	Vel. Processador	Memória	Disco
HP CM8060	1.6GHz	1GB	80GB
HP CM3530	515MHz	512MB	40GB
HP CM4730	533MHz	448MB	40GB
HP CM2320	450MHz	160MB	Não Possui
HP CM6030	835MHz	512MB	80GB

Da mesma forma, a tabela 5.2 apresenta multifuncionais retiradas de (LEXMARK, 2009).

Tabela 5.2: Configurações de MFPs

Modelo	Vel. Processador	Memória	Disco
Lexmark X546dtn	500MHz	256MB	Não Possui
Lexmark X860de 3	800MHz	256MB	20GB
Lexmark X782e XL	800MHz	768MB	40GB
Lexmark X654de	600MHz	256MB	40GB
Lexmark X544n	500MHz	128MB	Não Possui

Pelas informações apresentadas, é possível confirmar que a configuração desses equipamentos se assemelha bastante com características de computadores normais, permitindo que, em estando disponíveis para o processamento voluntário, produzam resultados válidos.

5.1.1.2 Software

Impressoras multifuncionais, assim como um computador normal, executam um conjunto de instruções a partir do seu *firmware*, que pode ser comparado a um sistema operacional. De fato, como o tamanho, complexidade e número de funcionalidades aumenta consideravelmente, grande parte das multifuncionais disponíveis no mercado fazem uso de sistemas operacionais tradicionais para possibilitar o controle das diversas funcionalidades e recursos oferecidos pela multifuncional. Em (INC, 2009) são apresentados como exemplos de sistemas operacionais, o Windows NT 4.0 Embedded, Windows XP Embedded, Windows CE e até mesmo Mac OS X.

Além das funcionalidades disponíveis no sistema operacional, as multifuncionais oferecem diversos recursos, como aplicações, *daemons* ou serviços. Esses recursos são cons-

truídos sobre o sistema operacional e possibilitam o controle do equipamento bem como a interação com os usuários. Como exemplos desses recursos, podem ser listados:

- Painel de controle, para permitir a interação com o usuário;
- *Web Server*, para as funções de gerenciamento remoto;
- Interpretadores *bytecode* ou máquinas virtuais, para aplicações de terceiros que possam estar hospedadas na MFP;
- Clientes e Servidores de rede, para envio e recebimento de documentos para diferentes destinos dentro da rede;
- Funções para conversão e processamento de imagens.

5.1.2 MFPs em Computação Voluntária

Apesar das MFPs serem equipamentos consistentes e com características semelhantes a computadores normais, não são equipamentos utilizados como recursos para ambientes de computação voluntária, especialmente por dois motivos:

- Produzem pequenos (porém constantes) intervalos de disponibilidade.
- Os *workers* disponíveis não são adaptados para esse tipo de recurso.

Com essas características, mesmo que um *worker* pudesse ser instalado em um recurso desse tipo, ele dificilmente conseguiria produzir algum resultado válido pois o período em que esse tipo de equipamento fica disponível para o processamento voluntário é muito curto. Isso significa que a MFP inicializaria a execução de uma aplicação voluntária mas não conseguiria atingir o intervalo de *checkpoint* para persistir os resultados produzidos, pois possivelmente receberia algum *job* para processar.

Em (HEWLETT-PACKARD, 2009) e (LEXMARK, 2009), conhecidos produtores de MFPs, é possível encontrar informações que dão conta que uma MFP permanece, em média, 70% do tempo de maneira ociosa. Esse período ocioso poderia ser utilizado para a produção de resultados para aplicações voluntárias, caso os problemas descritos acima fossem mitigados. Para verificar, em termos práticos, a disponibilidade de utilização de uma impressora multifuncional foi medida sendo que a imagem 5.1 mostra o gráfico produzido. Esse gráfico, em conjunto com a a tabela 5.3, representa a média de utilização de uma impressora MFP durante 8 horas de jornada de trabalho dentro de um laboratório de pesquisa com aproximadamente 30 usuários. Para a montagem do gráfico, a atividade de uma MFP foi medida, durante um período de um mês (descartando finais de semana). Para cada jornada em que as medições eram realizadas, o *status* da multifuncional era verificado em intervalos de 2 minutos. Com as informações recebidas, foi calculada uma média diária de utilização da impressora. Essa média aritmética simples teve desvio padrão de $\sigma = 1.23$. O resultado dessa medição pode ser verificado no gráfico 5.1, onde os picos indicam os períodos onde a MFP estava em *sleep-mode*.

Os valores descritos acima foram obtidos através de uma aplicação desenvolvida especialmente para esse fim. Trata-se de um aplicativo simples que é disparado e interrompido automaticamente em horários definidos. Durante a sua execução, em intervalos de 2 minutos, requisições SNMP eram realizadas à impressora e a resposta obtida armazenada em arquivos que foram, posteriormente, utilizados para coletar e analisar os resultados.

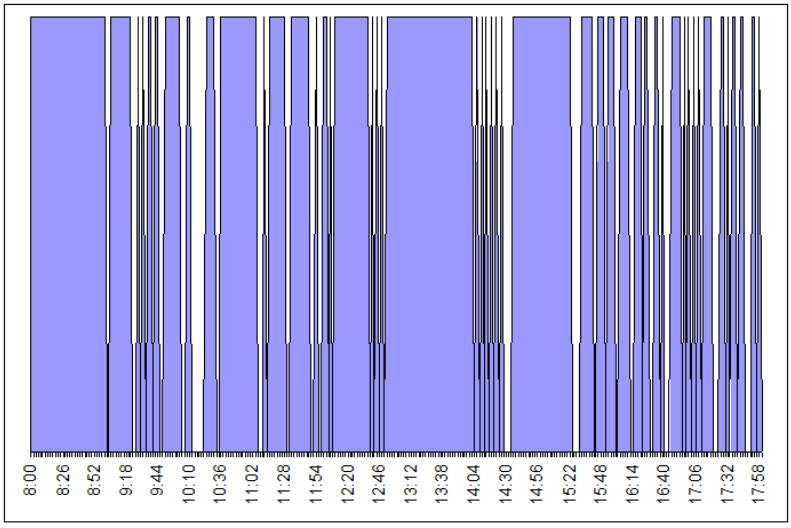


Figura 5.1: Fluxo de utilização diária de uma MFP

Tabela 5.3: Configurações de MFPs

Informação	Valor
Número de Jobs	117
Tempo em Sleep-Mode	05:25
Tempo Processando	02:35

Essa validação comprova a utilidade de produzir um *worker* que esteja adaptado a executar *jobs*, nesse tipo de recurso que apresenta curtos, porém constantes, períodos de ociosidade. Para que isso se apresente como viável, um dos primeiros passos é produzir um mecanismo de *checkpoint* que possa ser realizado de maneira freqüente e rápida, sem maiores custos ao processamento normal do recurso.

O modelo apresentado nesse capítulo trata de uma proposta para um mecanismo de *checkpoint*, focado em recursos com períodos curtos porém freqüentes de disponibilidade, utilizando conceitos de *diskless checkpoint* em conjunto com persistência em disco, utilizando prevalência de objetos. As principais características desse modelo são:

- *Checkpoint rápido:* o modelo deve possibilitar a realização de operações de *checkpoint* de maneira rápida e transparente. Isso significa que a utilização normal do recurso não deve ser afetada pelo mecanismo de *checkpoint* proposto;

- *Diskless Checkpoint:* para permitir a realização do *checkpoint* de maneira mais rápida, conceitos de *diskless checkpoint* são utilizados. Isso significa que as informações intermediárias produzidas são mantidas em memória, evitando gargalos de I/O;

- *Prevalência de Objetos:* o conceito de prevalência de objetos, descrito na sequência desse capítulo, é utilizado para permitir que as informações intermediárias produzidas sejam armazenadas em memória e, eventualmente, em disco.

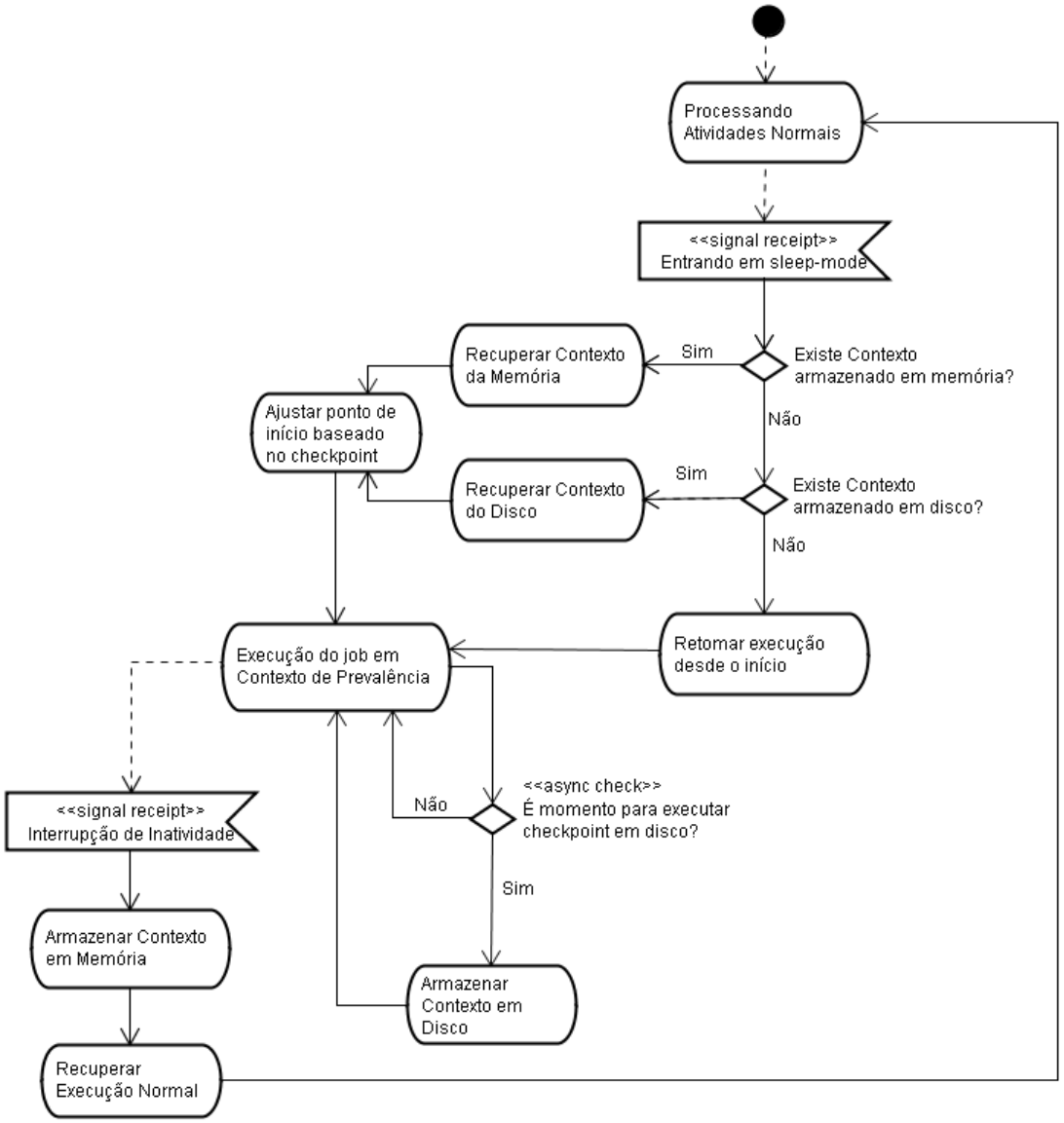


Figura 5.2: Fluxo básico de execução do Modelo

5.2 Visão Geral do Modelo

O modelo apresentado trata de uma solução que faz uso de abordagens de *checkpoint* em memória através de prevalência de objetos, aliada com *checkpoints* periódicos em disco. Nesse modelo, ao iniciar o período de inatividade, o mecanismo de checkpoint proposto busca em memória (e eventualmente em disco) por possíveis informações intermediárias já produzidas, a fim de retomar a execução sem perda dos resultados. Nessa proposta, tanto o armazenamento quanto a recuperação das informações devem ser realizados de maneira rápida, não afetando a utilização normal do recurso.

O diagrama de atividades apresentado na figura 5.2 apresenta o fluxo de execução que é obedecido pelo modelo.

Ao entrar em um período de inatividade, o recurso passa a estar apto a iniciar a execução de aplicações dentro do modelo de computação voluntária. Isso significa que, ao invés da MFP entrar em *sleep-mode*, ela passa a estar disponível para o início do processamento das aplicações, dentro da plataforma voluntária. No momento em que o sinal

de *sleep-mode* é reconhecido pelo device, o mecanismo de *checkpoint* proposto passa a ser executado, para recuperar informações intermediárias já produzidas e encapsular a execução do worker dentro de um contexto de prevalência, mantendo as informações produzidas para utilização futura.

O primeiro passo do algoritmo, implementado por esse modelo e apresentado no fluxo acima, é verificar a existência de dados intermediários já produzidos, em memória. O mecanismo de *checkpoint* será capaz de encontrar informações produzidas em memória caso a aplicação já tenha sido realizada dentro do contexto de prevalência, no recurso em questão. Caso não tenha sido encontrada nenhuma informação em memória, o mecanismo de *checkpoint* procura por informações persistidas em disco. Caso alguma informação tenha sido encontrada, esta é utilizada como ponto de início da aplicação. Nesse cenário, os resultados intermediários já produzidos são reaproveitados e a aplicação retoma sua execução no ponto onde foi previamente interrompida. Entretanto, caso não seja possível encontrar informações em memória ou disco, a execução da aplicação é retomada desde seu início. Nesse caso, nenhum resultado intermediário foi considerado, pois não foi possível encontrar informações que pudessem ser reaproveitadas.

Tendo sido recuperado o contexto de execução e o ponto de início da nova execução tenha sido marcado, a execução é retomada. A aplicação-alvo do modelo é encapsulada dentro de um contexto de prevalência de objetos de modo que todas as modificações em dados da aplicação sejam realizadas dentro de uma transação prevalente, para que tenham seu contexto armazenado. Isso significa que todas as informações produzidas serão armazenadas em um contexto prevalente e estarão disponíveis para futura recuperação, em caso de interrupções. Além do armazenamento das informações em memória, o mecanismo de *checkpoint* proposto realiza, de maneira periódica, eventuais serializações dos dados em disco físico. Esse armazenamento em disco é importante para evitar que as informações sejam mantidas apenas em memória, evitando possíveis erros ou perdas em caso de quedas de energia ou para eventuais operações de *restart* do equipamento.

A execução do *job* segue ativa até que o recurso receba um sinal, indicando o término do *sleep-mode*. Nessa situação, a execução da aplicação-alvo é imediatamente interrompida e os resultados até então produzidos são armazenados em memória. Quando o usuário proprietário do recurso solicita a utilização do meio, toda e qualquer atividade voluntária é interrompida e a capacidade de processamento do recurso é devolvida em sua totalidade para o proprietário. Entretanto, as informações intermediárias produzidas são mantidas em memória e os eventuais *checkpoints* em disco que tenham sido realizados permanecem disponíveis para consultas futuras. Essas informações mantidas em memória e em disco serão utilizadas como alimentação inicial para o mecanismo de *checkpoint* no próximo ciclo de inatividade.

Através da utilização de arquivos de configuração é possível determinar a realização dos *checkpoints* periódicos em disco. Essa configuração permite que *snapshots* da execução sejam realizados em períodos de tempo pre-estabelecidos e armazenados em disco físico. Essa atividade é realizada durante a execução da aplicação-alvo, sem prejudicar o desempenho e a produção dos resultados. A realização do *checkpoint* periódico não está associada a nenhuma premissa mínima e não possui nenhuma limitação em termos de intervalos mínimos. Isso significa que é possível realizar *checkpoints* em disco com a periodicidade necessária, de acordo com a aplicação a ser executada, dentro das características desejadas pelo administrador do sistema.

Todo o armazenamento das informações intermediárias produzidas é realizado através da utilização do conceito de prevalência de objetos, através da implementação em

Java, Prevayler. Os conceitos envolvidos com a prevalência de objetos são apresentados na seqüencia desse capítulo. Em relação à aplicação-alvo, esta deve ser brevemente modificada para implementar uma interface específica do modelo de *checkpoint*, estendendo uma classe específica do modelo. Essas modificações são importantes para que o mecanismo de *checkpoint* proposto possa ser aplicado à aplicação em que se deseja habilitar o conceito de *checkpoint*.

5.3 Prevalência de Objetos para Persistência

A prevalência de objetos pode ser definida como um modelo de persistência em memória que pode ser utilizado para armazenamento de objetos de uma determinada aplicação (VILLELA, 2002; DEARLE et al., 2009). A principal diferença entre o modelo de persistência através de prevalência de objetos, quando comparado com linguagens persistentes ou banco de dados orientados a objetos, está no fato de que a cópia primária dos objetos reside permanentemente em memória. Em um sistema de banco de dados orientado a objetos, a cópia primária das informações é mantida em disco e as informação são dinamicamente alocadas para memória, por um espaço de tempo específico.

Object Prevalence é um conceito proposto em (WUESTEFELD, ????). A idéia central desse mecanismo é *manter tudo em memória RAM, como se estivesse trabalhando diretamente com uma linguagem de programação* (VILLELA, 2002). *Object Prevalence* é um conceito simples que pode ser implementado em qualquer linguagem orientada a objetos, onde seja possível realizar serialização de objetos. Linguagens relativamente modernas, como C# e Java, dão suporte para que sejam realizadas serializações de objetos e, portanto, se oferecem como plataformas válidas para que o conceito de prevalência de objetos seja aplicado. O modelo proposto por Wuestefeld também apresenta implementações em outras linguagens de programação, como mostrado na tabela 5.4

Tabela 5.4: Implementações do Modelo de Prevalência

Implementação	Linguagem de Programação
Prevayler	Java
Bamboo	C#.NET
Py Per Syst	Python
Perlvayler	Perl
Common Lisp Prevalence	Common Lisp
Madeleine	Ruby

Em um sistema prevalente, toda e qualquer manipulação de objetos persistentes é realizada em memória. Isso significa que tanto alterações quanto buscas de informações previamente armazenadas é realizada em memória. Um sistema prevalente não faz uso de nenhum banco de dados ou servidor de aplicação. As alterações em objetos persistentes são realizadas através de transações específicas do modelo, garantindo que as informações sejam guardadas em memória para utilizações futuras.

A persistência dos objetos em memória, em sistemas prevalentes, é feita através da utilização de dois conceitos amplamente utilizados em sistemas gerenciadores de banco de dados orientados a objetos: *log* e *snapshot*. O *log* de transação é implementado pelo modelo de prevalência e está associado com a utilização de classes específicas para a representação de transações em um sistema prevalente. Isso significa que a aplicação

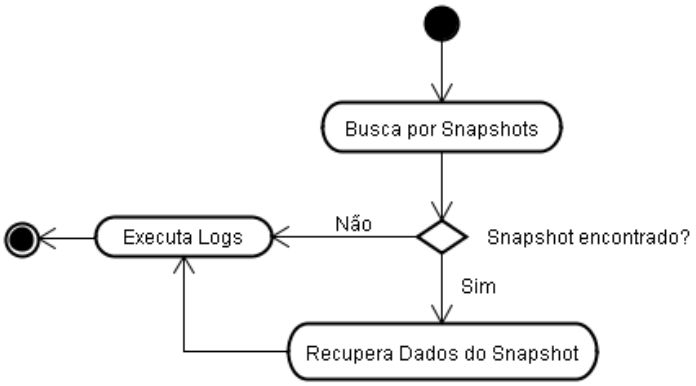


Figura 5.3: Atividades do Modelo de Prevalência

prevalente, ao utilizar o conceito de transações definido pelo modelo de prevalência, está habilitando a realização de *logs* para cada modificação realizada sobre os objetos envolvidos na transação.

O *snapshot* é um recurso que pode ser utilizado pela aplicação prevalente para realizar a persistência em disco das informações persistidas em memória. Em outras palavras, através da realização de operações de *snapshot*, o estado de todos os objetos persistidos em memória são serializados e as informações armazenadas em disco.

Os mecanismos de *log* e *snapshot* são complementares e utilizados em conjunto para permitir a persistência das informações através da prevalência de objetos. Como mencionado, para a operação realizada sobre as transações do modelo de prevalência, informações de *log* são geradas de maneira automática. Entretanto, a realização de *snapshots* somente é realizada através da intervenção direta da aplicação prevalente, não sendo realizada de maneira automática.

O algoritmo básico realizado pelo modelo de prevalência pode ser representado através do diagrama de atividades apresentado na figura 5.3. A partir do momento em que se realiza a restauração do sistema prevalente em caso de falhas ou no reinício normal da execução, o modelo prevalente busca por *snapshots* existentes, disponíveis em disco. Caso encontre alguma informação, os dados persistidos são recuperados em memória. O segundo passo é a execução de todos os *logs* criados após o *snapshot*. Nesse momento, as informações do *snapshot* são atualizadas com eventuais modificações que possam estar armazenadas em *log*. Quando não se confirma a existência de arquivos de *snapshot* em disco, apenas a atividade de re-execução das informações de *log* é realizada. Com isso, o estado da aplicação prevalente é recuperado, sendo equivalente ao estado anterior à falha ou ao desligamento do sistema.

5.4 Organização do Modelo

A implementação do modelo que valida o conceito apresentado, foi realizada na linguagem Java fazendo uso da solução Prevayler como modelo de prevalência de objetos. A solução está organizada em três pacotes básicos, desconsiderando o componente de *logs* que auxilia no processo de *debug* em caso de erros em tempo de execução. A figura 5.5 apresenta os pacotes de solução e o relacionamento existente entre eles:

O pacote `PeriodicCheckpoint` contém as classes que permitem a realização das operações de *checkpoint* periódicos. Nesse mesmo pacote, está disponível o arquivo de configuração que possibilita a parametrização do intervalo a ser obedecido para a realiza-

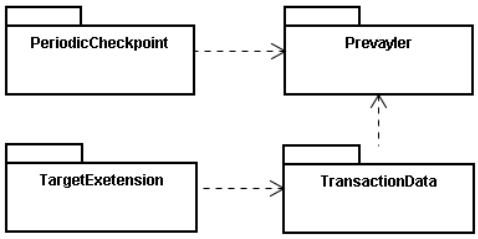


Figura 5.4: Organização dos Pacotes do Modelo

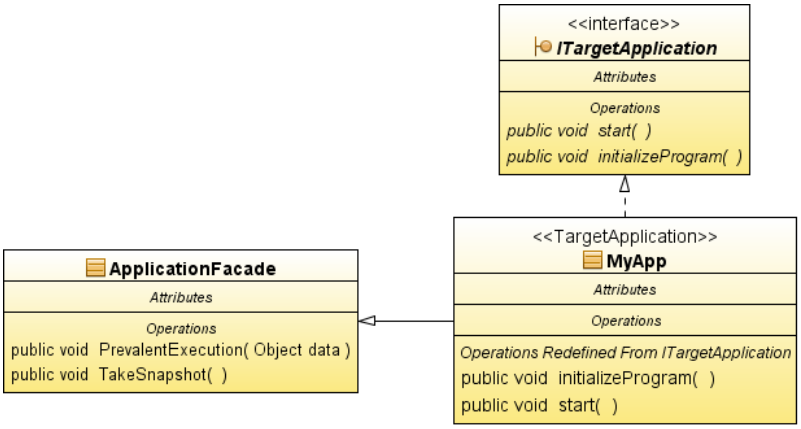


Figura 5.5: Modificações necessárias na aplicação-alvo

ção do *checkpoint*. As informações contidas no pacote `TargetExtension` permitem que a aplicação-alvo seja estendida e adaptada ao modelo de prevalência de objetos de maneira transparente. O pacote `TransactionData` contém as classes que realizam a configuração das transações prevalentes que são utilizadas dentro do modelo. É justamente através dessas classes prevalentes que a persistência em memória e disco é realizada. Os pacotes `PeriodicCheckpoint` e `TransactionData` estão diretamente relacionados com o componente de prevalência utilizado pelo modelo. Apesar da protótipo utilizar Prevayler como solução de prevalência, o modelo em si não está atrelado a nenhuma solução de prevalência específica. Isso significa que o componente de prevalência pode ser substituído por outra solução, caso necessário ou desejado. O modelo implementado também faz uso, de maneira ampla, de um componente de *logs* criado para possibilitar o armazenamento, de maneira transparente, de eventuais erros de execução do protótipo.

5.4.1 Aplicações

O mecanismo de *checkpoint* implementado nesse modelo está preparado para trabalhar em aplicações do tipo *bag-of-tasks*, que tenham sido previamente adaptadas ao modelo de *checkpoint* proposto. Aplicações BoT podem ser entendidas como um caso de aplicação paralela onde as tarefas são independentes. São, tipicamente, aplicações simples que podem ser utilizadas em uma gama de cenários, incluindo *data mining*, buscas massivas, simulações, processamento de imagens entre outros (CIRNE et al., 2003).

Para serem encapsuladas dentro do contexto de *checkpoint*, as aplicações devem sofrer três pequenas modificações, como descrito abaixo e representado através do modelo da figura 5.5:

1. Implementar uma interface. A aplicação deve implementar `ITargetApplication`,

que é a interface que define a existência do método para a inicialização de variáveis e do ponto de início da aplicação-alvo;

2. Utilizar a variável `StartupValue`. O mecanismo de *checkpoint* utiliza, via *reflection*, a variável `StartupValue` como objeto de inicialização da aplicação;

3. Estender uma classe. A aplicação-alvo deve utilizar as classes disponibilizadas pelo modelo para estender a classe `ApplicationFacade`;

4. Chamar o método de prevalência. O método `PrevalentExecution(Object data)` deve ser utilizado para encapsular a informação a ser persistida.

5.4.2 Implementando `ITargetApplication`

A aplicação-alvo a ser executada dentro do contexto de *checkpoint* proposto precisa implementar dois métodos: `start()` e `initializeProgram()`. O método `initializeProgram()`, como o nome define, deve ser utilizado para realizar as inicializações desejadas para a aplicação. A sua definição é mandatória, entretanto, pode ter o seu conteúdo vazio. Esse método é especialmente interessante quando a aplicação-alvo realiza manipulações de arquivos.

O método `start()` deve conter o início da aplicação. Para o protótipo definido, o método de `start()` deve ser entendido como o método `main(String args[])` de uma aplicação tradicional Java. Trata-se do método que será invocado, via *reflection*, para determinar o início da aplicação-alvo.

Em tempo de execução, o mecanismo de *checkpoint* proposto procura e invoca, via *reflection*, o método `initializeProgram()` para que possíveis variáveis da aplicação sejam inicializadas. Após, o método `start()` também é chamado, via *reflection*, para determinar o início da aplicação-alvo, dentro de um contexto de prevalência.

5.4.3 Utilizando a variável `StartupValue`

Os resultados intermediários produzidos e armazendos pelo mecanismo de *checkpoint* são utilizados em futuras execuções da aplicação. Para que seja possível a correta inicialização do modelo e conseqüente recuperação das informações produzidas, a aplicação-alvo deve contemplar a existência de uma variável de `StartupValue`. Essa variável é inicializada, em tempo de execução, com o último resultado válido produzido.

Não existe um tipo específico esperado para a variável. Isso significa que o tipo da variável `StartupValue` não precisa ser específico, apenas o nome da variável. Em tempo de execução o modelo vai recuperar a definição da variável, identificar o seu tipo, buscar a última informação produzida e inicializar o modelo de acordo.

5.4.4 Estendendo `ApplicationFacade`

A aplicação deve estender a classe `ApplicationFacade` para possibilitar a chamada do método `PrevalentExecution` que define o encapsulamento dentro do conceito de prevalência. A classe `ApplicationFacade` contém a implementação desse mecanismo, que é transparente para a aplicação-alvo. Em termos finais de utilização, ao projetar a aplicação-alvo, deve-se apenas ter o cuidado de realizar a chamada desse método, como descrito a seguir.

5.4.5 Invocando a Prevalência

É indicada a realização da chamada desse método sempre que algum resultado intermediário tenha sido produzido, seja ele válido ou não. Isso significa que, em uma aplicação BoT, a informação a ser utilizada pela aplicação deve ser encapsulada dentro do conceito de prevalência, através da chamada do método `PrevalentExecution`. Isso garante que, em caso de falhas ou interrupções, o mecanismo de *checkpoint* será capaz de recuperar a execução considerando os últimos resultados intermediários produzidos.

5.4.6 Exemplos de Modificações

Para exemplificar as modificações necessárias para permitir que uma aplicação possa ser corretamente interpretada dentro do mecanismo proposto, foi projetada e implementada a aplicação abaixo. A aplicação em questão busca por números de Mersenne primos, que são números inteiros positivos, primos, definidos por $M_n = 2^n - 1$ onde n também é um número inteiro positivo.

Listing 5.1: Exemplo de Aplicação-Alvo Modificada

```
1  public class Mersenne extends ApplicationFacade
2                       implements ITargetApplication
3  {
4      private int StartupValue = 3;
5      public void initializeProgram() {};
6
7      public void start(){
8          int upb = 5000;
9          for (int p = StartupValue; p <= upb; p += 2)
10             if (isPrime(p) && isMersennePrime(p))
11             {
12                 System.out.print(" M" + p);
13                 this.PrevalentExecution(p);
14             }
15     }
16
17     public static boolean isPrime(int p) {
18         if (p == 2)
19             return true;
20         else if (p <= 1 || p % 2 == 0)
21             return false;
22         else {
23             int to = (int)Math.sqrt(p);
24             for (int i = 3; i <= to; i += 2)
25                 if (p % i == 0)
26                     return false;
27             return true;
28         }
29     }
30
31     public static boolean isMersennePrime(int p) {
32         if (p == 2)
33             return true;
34         else {
35             BigInteger m_p = BigInteger.ONE.shiftLeft(p).subtract(BigInteger.ONE);
36             BigInteger s = BigInteger.valueOf(4);
37             for (int i = 3; i <= p; i++)
38                 s = s.multiply(s).subtract(BigInteger.valueOf(2)).mod(m_p);
39             return s.equals(BigInteger.ZERO);
40         }
41     }
42 }
```

As linhas `1` e `2` mostram a interface sendo implementada, bem como a classe sendo estendida. Como a aplicação em questão não precisa realizar nenhuma inicialização de variáveis ou propriedades específicas, o método `initializaProgram()` está defi-

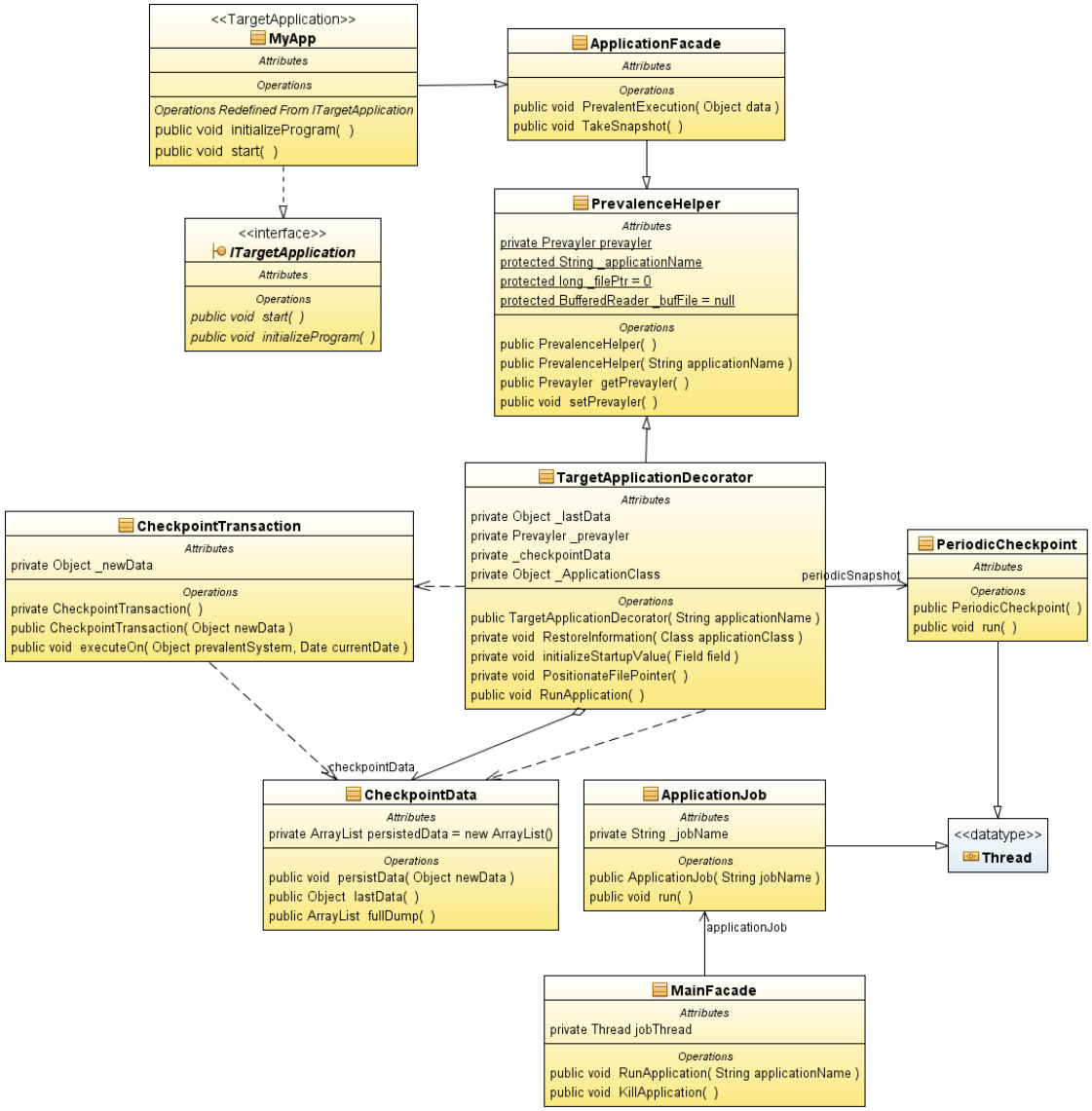


Figura 5.6: Diagrama de Classes do Mecanismo de *Checkpoint*

nido vazio na linha 5. A linha 4 mostra a definição da variável esperada pelo modelo. Por fim, a linha 7 define o método principal da aplicação-alvo. Com as modificações descritas acima, a aplicação está apta a ser executada dentro de um contexto de prevalência de objetos, para o modelo de *checkpoint* proposto.

5.5 Arquitetura do Mecanismo de *Checkpoint*

O mecanismo de *checkpoint* responsável pelo armazenamento das informações prevalentes é realizado através da interação das classes representadas pelo diagrama da figura 5.6.

Esse conjunto de classes é responsável por encapsular a execução das aplicações dentro do modelo de prevalência, permitindo que resultados intermediários produzidos sejam persistidos em memória e, eventualmente, serializados em disco. Cada um dos componentes do modelo mostrados no diagrama de classe da figura 5.6 é detalhado em uma das seções seguintes desse capítulo.

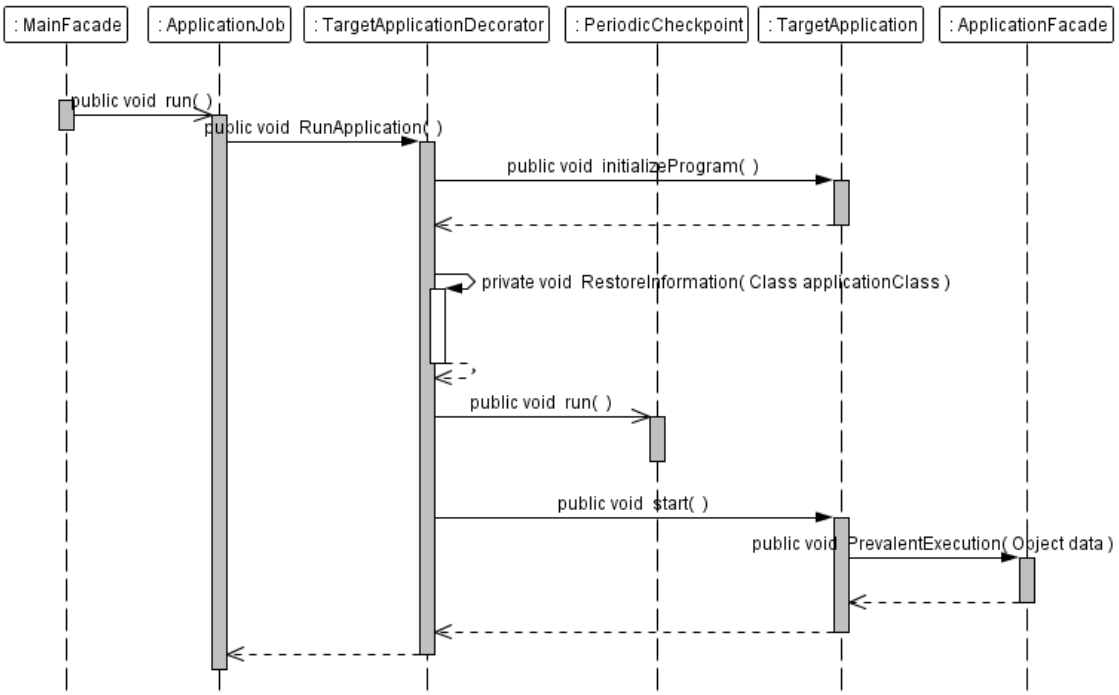


Figura 5.7: Diagrama de Sequência do Mecanismo de *Checkpoint*

Como dito anteriormente, as classes do modelo de *checkpoint* interagem entre si para habilitar um mecanismo de *checkpoint*. O diagrama de seqüencia mostrado na figura 5.7 detalha o fluxo básico seguido pelo mecanismo de *checkpoint*.

A classe `MainFacade` é responsável por determinar o início do fluxo básico do modelo, através de uma chamada assíncrona para a *thread* `ApplicationJob`. Esse componente é responsável por instanciar o `TargetApplicationDecorator` com os parâmetros de execução corretos para a aplicação-alvo e determinar o início da execução do *job*. A classe `TargetApplicationDecorator` engloba as principais funcionalidades do modelo, chamadas por meio de operações de *reflection*, dos métodos de inicialização e início da aplicação-alvo. Ao mesmo tempo, essa classe é responsável por instanciar a criação de uma nova *thread deamon* para a realização de *checkpoints* periódicos através da classe `PeriodicCheckpoint`. Com a aplicação-alvo inicializada, esta é responsável pela chamada das execuções prevalentes através da `ApplicationFacade`.

A organização das *threads* existentes no mecanismo de *checkpoint* proposto pode ser representada pela figura 5.8. Nela é possível identificar que a *thread* principal, através da classe `MainFacade` dispara o início de execução da *thread daemon* da aplicação-alvo, através de um *decorator* que, por sua vez, também é responsável pela inicialização do processo de execução periódica de *checkpoints*. A partir do momento em que a *thread* principal é finalizada, as demais *threads* são interrompidas, tendo seu ciclo de vida finalizado.

5.5.1 `ITargetApplication`

A interface `ITargetApplication` define os métodos que devem ser implementados pela aplicação-alvo, para que o conceito de prevalência possa ser aplicável. Ambos os métodos são chamados em tempo de execução pelo mecanismo de *checkpoint* para executarem as operações e instruções específicas da aplicação. Todo programa em que se deseja aplicar o modelo de *checkpoint* proposto deve, obrigatoriamente, implementar

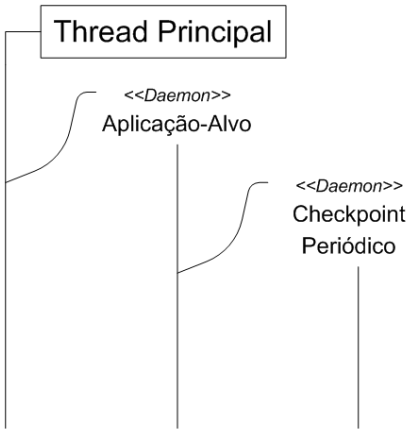


Figura 5.8: Organização das Threads do Modelo

essa interface e, por consequência, prover uma implementação para os métodos descritos abaixo.

5.5.1.1 `initializeProgram()`

Nesse método, apresentando na linha `5` da listagem da interface, devem estar contidas as inicializações específicas da aplicação. Variáveis públicas e privadas, abertura de arquivos e demais inicializações do programa podem ser realizadas nesse método. Em tempo de execução, o mecanismo de *checkpoint* realiza a chamada desse método via *reflection* permitindo que o programa seja corretamente inicializado.

5.5.1.2 `start()`

Demarca o ponto de início da aplicação-alvo. O mecanismo de *checkpoint* utiliza esse método, definido na linha `7` da listagem, como ponto de entrada da aplicação, de maneira equivalente ao método `main(String[] args)` de uma aplicação normal. Em linhas gerais, o ponto de início da aplicação em que se deseja aplicar os conceitos de prevalência deve ser o método `start()`.

5.5.1.3 *Listagem de Código*

Listing 5.2: Implementação da Interface `ITargetApplication`

```
1 package Checkpoint.TargetExtension;
2
3 public interface ITargetApplication {
4
5     public void initializeProgram();
6
7     public void start();
8
9 }
```

5.5.2 `MainFacade`

A classe `MainFacade` tem como principal objetivo esconder a complexidade envolvida na inicialização da aplicação-alvo. Os métodos expostos por essa classe e descritos na seqüência, permitem que a aplicação-alvo possa ser iniciada e interrompida de maneira

transparente, encapsulando as particulariedades envolvidas nesse processo.

5.5.2.1 `RunApplication(String applicationName)`

Método principal do modelo, que se apresenta como ponto de entrada para a execução de aplicações dentro do modelo de prevalência proposto pelo *checkpoint*. O método `RunApplication(String applicationName)` deve ser chamado pela plataforma de computação voluntária, recebendo como parâmetro o caminho completo da aplicação que deseja-se executar.

O nome da aplicação, recebido como parâmetro, é também utilizado para a criação do repositório específico da aplicação. Esse repositório serve para o armazenamento das informações de *snapshots* produzidas. A organização de *snapshots* dentro de *packages* específicos por aplicação permite que o modelo ofereça suporte à execução de múltiplas aplicações de maneira transparente. Nessa abordagem, ao iniciar a execução de uma aplicação, o mecanismo de *checkpoint* procura por um *package* com o nome da aplicação para então recuperar as informações de *snapshots*.

A linha `10` da listagem mostra que a aplicação-alvo é executada dentro do modelo como uma *thread daemon*. Esse tipo de *thread*, de acordo com (BRYANT; HARTNER, 2009) permite que um serviço contemplado pela *thread* seja executado como tarefa de *background*. *Threads daemon* tem o ciclo de vida delimitado pela execução do programa normal, o que significa que ela será mantida enquanto o programa principal estiver em execução. Dentro do modelo do mecanismo de *checkpoint* proposto, executar a aplicação-alvo como uma *thread daemon* significa que o *job* será executado enquanto não houverem interrupções.

O principal objetivo desse método é encapsular a complexidade de inicialização e chamada da *thread* que contém a execução da aplicação-alvo. A linha `11` determina o início da execução da *thread* da aplicação-alvo, através da classe `ApplicationJob`.

5.5.2.2 `KillApplication()`

Permite que a execução da aplicação-alvo seja interrompida, devolvendo o processamento para outras atividades de maior prioridade. Dentro do modelo proposto, esse método é chamado quando o proprietário do recurso computacional solicita a sua utilização. Nesse momento, toda e qualquer execução ou processamento de atividades relacionadas com o projeto voluntário deve ser interrompida imediatamente.

Como a aplicação-alvo está sendo executada como uma *thread daemon*, o fluxo principal de execução pode realizar a interrupção da execução. A partir da interrupção da *thread* da aplicação-alvo, as informações intermediárias produzidas dentro do contexto de prevalência de objetos são mantidas em memória e eventuais *snapshots* armazenados em disco.

A ação de interrupção da aplicação através da chamada ao método em questão resulta na finalização imediata da *thread daemon* e também de eventuais *threads* criadas pela aplicação-alvo. Esse processo inclui também o término da *thread* responsável pela execução periódica de *snapshots*, posto que esta também é definida como uma *thread daemon*, tópico que será coberto na seqüência.

5.5.2.3 *Listagem de Código*

Listing 5.3: Código da classe `MainFacade`

```
1  package Checkpoint.TargetExtension;
2
3  public class MainFacade {
4
5      private Thread jobThread;
6
7      public void RunApplication(String applicationName){
8          Runnable applicationJob = new ApplicationJob(applicationName);
9          jobThread = new Thread(applicationJob);
10         jobThread.setDaemon(true);
11         jobThread.start();
12     }
13
14     public void KillApplication(){
15         jobThread.interrupt();
16     }
17 }
```

5.5.3 `ApplicationJob`

A classe `ApplicationJob`, que estente a classe `Thread`, permite o início da aplicação-alvo. O construtor dessa classe recebe o nome da aplicação a ser executada como parâmetro e, através do método `run()` encapsula a execuçao da aplicação-alvo. Como essa *thread* foi definida como *daemon*, seu ciclo de vida fica contido dentro da `MainFacade`.

5.5.3.1 `public void run()`

A partir da chamada de execução realizada pela `MainFacade`, a `ApplicationJob` realiza a configuração do *decorator* para que a aplicação-alvo seja executada. Esse método instancia um novo `TargetApplicationDecorator` passando o nome da aplicação recebida pelo construtor como parâmetro e realiza a chamada do método responsável pela inicialização da aplicação-alvo. Através dos métodos disponíveis no *decorator*, a produção de resultados intermediários pode ser iniciada.

5.5.3.2 Listagem de Código

Listing 5.4: Código da classe `ApplicationJob`

```
1  package Checkpoint.TargetExtension;
2
3  public class ApplicationJob extends Thread{
4
5      private String _jobName;
6
7      public ApplicationJob(String jobName){
8          this._jobName = jobName;
9      }
10
11     @Override
12     public void run(){
13         TargetApplicationDecorator decorator = new TargetApplicationDecorator(this.↩
               _jobName);
14         decorator.RunApplication();
15     }
16 }
```

5.5.4 `TargetApplicationDecorator`

A classe `TargetApplicationDecorator` é o principal componente da arquitetura do mecanismo de *checkpoint* proposto. Através dela, estão disponíveis mecanismos para a recuperação das informações intermediárias produzidas, inicialização da aplicação-alvo, configuração e início do mecanismo de *checkpoint* periódico além do início da aplicação-alvo, propriamente dita.

5.5.4.1 `public void RunApplication()`

Trata-se do principal método do modelo de *checkpoint* proposto. Através desse método, a aplicação-alvo a ser executada dentro do ambiente prevalente é inicializada, parametrizada e tem sua execução invocada. Esse é o método que contém as operações de inicialização com a última informação produzida, além de contemplar as informações para chamada do método `initializeProgram` e `start`.

A partir do nome da aplicação a ser executada, o método `RunApplication()` busca pela *class* da aplicação, via *reflection* e cria uma nova instância da aplicação-alvo. A partir desse momento, o método `initializeProgram()` é procurado, dentro da aplicação-alvo e, ao ser encontrado, sua execução é invocada. Nesse momento, a aplicação-alvo está instanciada em memória e seus valores privados inicializados.

Com essas informações, o método `RestoreInformation()` (descrito na seqüência) é chamado, fazendo com que o modelo de *checkpoint* verifique a existência de possíveis valores intermediários produzidos e inicializando a aplicação, caso aplicável. Além disso, como é possível verificar nas linhas `14-16` da listagem abaixo, o componente responsável pela execução de *checkpoints* periódicos é inicializado.

Por fim, o algoritmo implementando no método `RunApplication()` chama, via *reflection*, a execução do método *start*, dando início à execução da aplicação-alvo. Nesse momento, o modelo de *checkpoint* proposto criou uma nova instância da aplicação-alvo, inicializou a mesma com possíveis valores já produzidos, configurou o início da execução periódica de *checkpoints* e inicializou a execução da aplicação-alvo. A partir desse momento, o fluxo de execução passa a ser controlado pela aplicação-alvo, até que uma interrupção ocorra ou que a aplicação-alvo finalize seu processamento.

Listing 5.5: Implementação do método `RunApplication`

```
1  public void RunApplication() {
2      try{
3          Class params[] = {};
4          Object paramsObj[] = {};
5
6          Class thisClass = Class.forName(_applicationName.toString());
7          _ApplicationClass = thisClass.newInstance();
8
9          Method initializeMethod = thisClass.getDeclaredMethod("initializeProgram", ↩
               params);
10         initializeMethod.invoke(_ApplicationClass, paramsObj);
11
12         this.RestoreInformation(thisClass);
13
14         Runnable periodicSnapshot = new PeriodicCheckpoint();
15         Thread thread = new Thread(periodicSnapshot);
16         thread.start();
17
18         Method startMethod = thisClass.getDeclaredMethod("start", params);
19         startMethod.invoke(_ApplicationClass, paramsObj);
20     }
21     catch (ClassNotFoundException cnf){
```

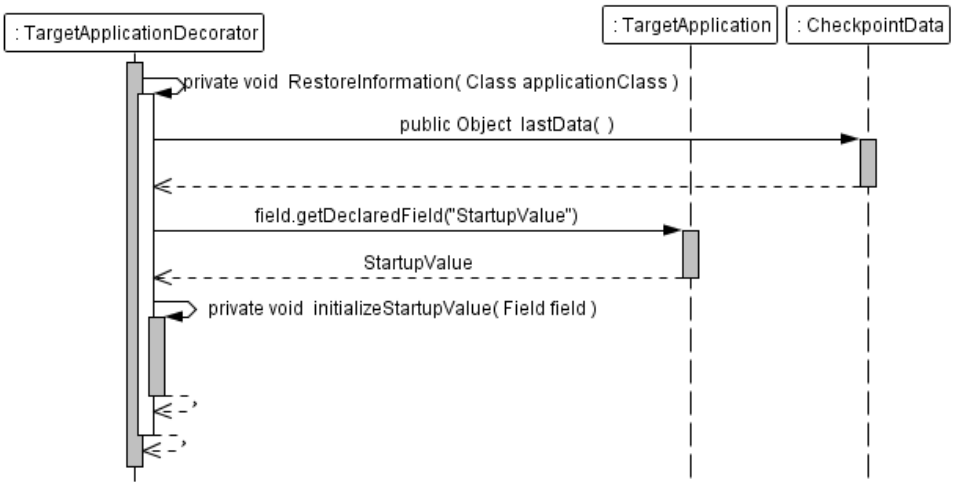


Figura 5.9: Diagrama de Seqüência do método `RestoreInformation`

```
22          CheckPrevalentLog.getInstance().logError("TargetApplicationDecorator", "↩
                RunApplication", "Caught an ClassNotFoundException when trying to run the ↩
                target application.", cnf);
23      }
24      catch (InstantiationException ie){
25          CheckPrevalentLog.getInstance().logError("TargetApplicationDecorator", "↩
                RunApplication", "Caught an InstantiationException when trying to run the ↩
                target application.", ie);
26      }
27      catch (IllegalAccessException iae){
28          CheckPrevalentLog.getInstance().logError("TargetApplicationDecorator", "↩
                RunApplication", "Caught an IllegalAccessException when trying to run the ↩
                target application.", iae);
29      }
30      catch (NoSuchMethodException nsm){
31          CheckPrevalentLog.getInstance().logError("TargetApplicationDecorator", "↩
                RunApplication", "Caught an NoSuchMethodException when trying to run the ↩
                target application.", nsm);
32      }
33      catch (InvocationTargetException ite){
34          CheckPrevalentLog.getInstance().logError("TargetApplicationDecorator", "↩
                RunApplication", "Caught an InvocationTargetException when trying to run ↩
                the target application.", ite);
35      }
36  }
```

5.5.4.2 *`private void RestoreInformation (Class appClass)`*

O processo de recuperação das informações intermediárias já produzidas é fundamental para o modelo de *checkpoint* proposto funcionar como esperado. Esse fluxo é implementado no *decorator* através do método `RestoreInformation()`. Esse método recebe por parâmetro a classe referente à aplicação que se deseja executar e busca os valores produzidos, incializando a aplicação de acordo.

O diagrama de seqüência apresentado na figura 5.9 detalha o fluxo observado por esse método. A classe `TargetApplicationDecorator` inicia o processo de recuperação das informações intermediárias produzidas chamando, pelo método `run()`, a execução do `RestoreInformation()`. Nesse momento, a instância da classe `CheckpointData` é utilizada para buscar o último resultado válido produzido, através do método `lastData()`. O valor retornado por essa execução será utilizado para inicializar a aplicação-alvo que será executada.

Com o último valor produzido por execuções anteriores identificado, o mecanismo

de *checkpoint* proposto está apto a inicializar a aplicação. Nesse momento, o mecanismo de *checkpoint* procura, na classe recebida por parâmetro, pela existência da variável `StartupValue`, que é definida como pré-requisito para inicialização da aplicação. Tendo sido identificado o último valor produzido e a variável `StartupValue`, o próximo passo do algoritmo é inicializar a variável com o valor, de acordo com o tipo específico que é utilizado pela aplicação, através do método `initializeStartupValue()`.

5.5.4.3 *`private void initializeStartupValue(Field field)`*

A implementação desse método, apresentado na listagem abaixo, oferece um mecanismo para que a variável `StartupValue` seja inicializada com o último valor produzido, em tempo de execução por meio de *reflection*. O pré-requisito para que a inicialização da aplicação seja correta, é a existência de uma variável com o nome `StartupValue` na aplicação-alvo. Entretanto, não existem restrições em relação ao tipo da mesma, o que permite variações de acordo com o interesse da aplicação.

O método `initializeStartupValue()` realiza uma verificação, em tempo de execução, para identificar o tipo associado à variável em questão. A partir dessa informação, *casts* específicos são realizados para permitir que a aplicação tenha o valor inicial ajustado conforme esperado. Para o caso onde a variável `StartupValue` está associada à utilização de arquivos, o método `PositionateFilePointer()` é utilizado para permitir que o ponteiro seja corretamente posicionado no arquivo.

Listing 5.6: Implementação do método `initializeStartupValue`

```
1  private void initializeStartupValue(Field field)
2  {
3      try{
4          if ( field.getType() == int.class){
5              if (this._lastData != null)
6                  field.set(_ApplicationClass, ((Integer)this._lastData).intValue());
7          }
8          if ( field.getType() == long.class){
9              if (this._lastData != null)
10                 field.set(_ApplicationClass, ((Long)this._lastData).longValue());
11         }
12         if ( field.getType() == double.class){
13             if (this._lastData != null)
14                 field.set(_ApplicationClass, ((Double)this._lastData).doubleValue());
15         }
16         if ( field.getType() == java.lang.String.class){
17             if (this._lastData != null)
18                 field.set(_ApplicationClass, ((String)this._lastData).toString());
19         }
20         if ( field.getType() == BufferedReader.class) {
21             _bufFile = (BufferedReader) field.get(_ApplicationClass);
22             if (this._lastData != null){
23                 PositionateFilePointer();
24                 field.set(_ApplicationClass, _bufFile);
25             }
26         }
27     } catch (IllegalAccessException iae){
28         CheckPrevalentLog.getInstance().logError("TargetApplicationDecorator", "↩
               initializeStartupValue", "Caught an IllegalAccessException when ↩
               initializing application values.", iae);
29     }
30 }
```

5.5.4.4 `private void PositionateFilePointer()`

O método apresentado na listagem abaixo é utilizado durante o processo de inicialização do campo `StartupValue` para os casos onde o mesmo esteja associado à utilização de arquivos. Nesse caso, esse método descarta os *chunks* de arquivos que já foram lidos, posicionando a leitura para a próxima informação a ser processada pela aplicação-alvo.

Listing 5.7: Implementação do método `PositionateFilePointer`

```
1 private void PositionateFilePointer(){
2     try{
3         String junk;
4         _filePtr = ((Long)this._lastData).longValue();
5         for (int i = 0; i < _filePtr; i++) { junk = _bufFile.readLine(); }
6    } catch (IOException io){
7          CheckPrevalentLog.getInstance().logError("TargetApplicationDecorator", "↩
               PositionateFilePointer", "Caught an IOException when eliminating junk ↩
               information", io);
8     }
9 }
```

5.5.5 `PeriodicCheckpoint`

O modelo do mecanismo de *checkpoint* proposto define que operações de *checkpoint* em disco devam ser executadas de maneira periódica. Esse armazenamento em disco das informações periódicas produzidas é realizado através da execução de *snapshots* cujo resultado produzido é armazenado em disco. A classe `PeriodicCheckpoint` é responsável pelo controle e execução da realização desses *snapshots* em disco, através do modelo de prevalência adotado.

A classe é projetada como uma *thread daemon* criada a partir do *decorator*, quando a configuração e reinício da aplicação-alvo é realizada. O *checkpoint* é realizado de maneira periódica, em intervalos de tempo pré-estabelecidos e configurados em um arquivo de parametrização.

Não existem limitações técnicas em relação ao intervalo a ser utilizado. Isso significa que o modelo pode ser configurado para realizar operações de *checkpoint* em disco em intervalos muito curtos, como segundos, ou longos, como horas. Entretanto, a execução de maneira muito frequente pode acarretar perdas de processamento, posto que para a realização da persistência em disco das informações, é necessária a realização da serialização das informações mantidas em memória.

A configuração do intervalo de tempo a ser respeitado pelo modelo deve ser realizada no arquivo `checkprev.properties`. A parametrização deve ser feita utilizando a propriedade `periodic_checkpoint` e o valor a ser utilizado deve ser expressado em *ms*.

5.5.5.1 `public void run()`

A execução de *checkpoints* realizados em intervalos de tempo definidos como descrito acima é realizado no método principal da *thread*, através de uma iteração infinita, como mostrado na linha `26`. Por ser classificada como *daemon*, a *thread* terá seu ciclo de vida finalizado assim que a execução da aplicação-alvo for interrompida. Enquanto a produção de resultados não é interrompida, a *thread* que realiza os *checkpoints* periódicos utiliza a classe `ApplicationFacade` para realizar chamadas de execução de *snapshots* do modelo de prevalência.

5.5.5.2 *Listagem de Código*

Listing 5.8: Código da classe `PeriodicCheckpoint`

```
1  package Checkpoint.PeriodicCheckpoint;
2
3  import Checkpoint.log.CheckPrevalentLog;
4  import Checkpoint.TargetExtension.ApplicationFacade;
5  import java.io.FileNotFoundException;
6  import java.io.IOException;
7  import java.util.Properties;
8
9  public class PeriodicCheckpoint extends Thread{
10
11     public PeriodicCheckpoint(){
12         setDaemon(true);
13     }
14
15     @Override
16     public void run(){
17         try{
18             ApplicationFacade extension = new ApplicationFacade();
19
20             Properties properties = new Properties() ;
21             String propPath = "/Checkpoint/PeriodicCheckpoint/checkprev.properties"
22             properties.load(this.getClass().getResourceAsStream(propPath));
23
24             int ckpt =Integer.parseInt(properties.getProperty("periodic_checkpoint"));
25
26             while (true){
27                 Thread.sleep(ckpt);
28                 extension.TakeSnapshot();
29             }
30         }
31         catch (InterruptedException ie){
32             CheckPrevalentLog.getInstance().logError("PeriodicCheckpoint", "run", "↩
                   Caught an InterruptedException on the PeriodicCheckpoint execution.", ↩
                   ie);
33         }
34         catch (FileNotFoundException fnf){
35             CheckPrevalentLog.getInstance().logError("PeriodicCheckpoint", "run", "↩
                   Caught an FileNotFoundException on the PeriodicCheckpoint execution.",↩
                    fnf);
36         }
37         catch (IOException io){
38             CheckPrevalentLog.getInstance().logError("PeriodicCheckpoint", "run", "↩
                   Caught an IOException on the PeriodicCheckpoint execution.", io);
39         }
40     }
41
42 }
```

5.5.6 `PrevalenceHelper`

A classe `PrevalenceHelper` permite configurar as informações referentes à prevalência, para o modelo de *checkpoint* proposto. Além dessa configuração básica das informações de prevalência, esse *helper* é utilizado como objeto-base, de onde as classes do tipo `TargetApplicationDecorator` e `ApplicationFacade` herdam. Em linhas gerais, a `PrevalenceHelper` é responsável por criar o repositório onde as classes especializadas irão depositar as informações intermediárias produzidas.

5.5.6.1 *`public PrevalenceHelper(String applicationName)`*

O construtor da classe recebe o nome da aplicação-alvo que será executada no ambiente voluntário. Essa informação recebida por parâmetro é importante para permitir a

configuração, em disco, dos *packages* onde as informações de *snaphot* serão armazenadas.

Quando o construtor do `PrevalenceHelper` é chamado, a configuração do ambiente de persistência de objetos através de prevalência é realizada através do método `setPrevayler()`.

5.5.6.2 *`public void setPrevayler()`*

No momento em que a instância da classe `PrevalenceHelper` é criada, o modelo de *checkpoint* proposto cria, através da `PrevaylerFactory` do modelo de prevalência utilizado, a base de persistência que será utilizada durante a execução da aplicação.

O conjunto de objetos criados através desse método possibilita que todos os objetos manipulados pelas transações prevalentes tenham seus resultados intermediários armazenados. Como resultado final, o método `setPrevayler()` disponibiliza para o modelo de *checkpoint* proposto, o repositório de informações produzidas, que é mantido no atributo estático `Prevayler`.

5.5.6.3 *Listagem de Código*

Listing 5.9: Código da classe `PrevalenceHelper`

```
1  package Checkpoint.TargetExtension;
2
3  import Checkpoint.TransactionData.CheckpointData;
4  import Checkpoint.log.CheckPrevalentLog;
5  import java.io.BufferedReader;
6  import java.io.IOException;
7  import org.prevayler.Prevayler;
8  import org.prevayler.PrevaylerFactory;
9
10 public class PrevalenceHelper {
11
12     private static Prevayler prevayler;
13
14     protected static String _applicationName;
15
16     protected static long _filePtr = 0;
17
18     protected static BufferedReader _bufFile = null;
19
20     @Deprecated
21     public PrevalenceHelper(){ };
22
23     public PrevalenceHelper(String applicationName) {
24         _applicationName = applicationName;
25          setPrevayler();
26     }
27
28     public Prevayler getPrevayler() {
29         return prevayler;
30     }
31
32     public void setPrevayler() {
33         try {
34             if (prevayler == null)
35                 prevayler = PrevaylerFactory.createPrevayler(new CheckpointData(), "↩
                       Checkpoint\\" + _applicationName);
36         }
37         catch (IOException io){
38             CheckPrevalentLog.getInstance().logError("PrevalenceHelper", "setPrevayler↩
                   ", "Caught an IOException when setting the Prevayler.", io);
39         }
40         catch (ClassNotFoundException cnf){
```

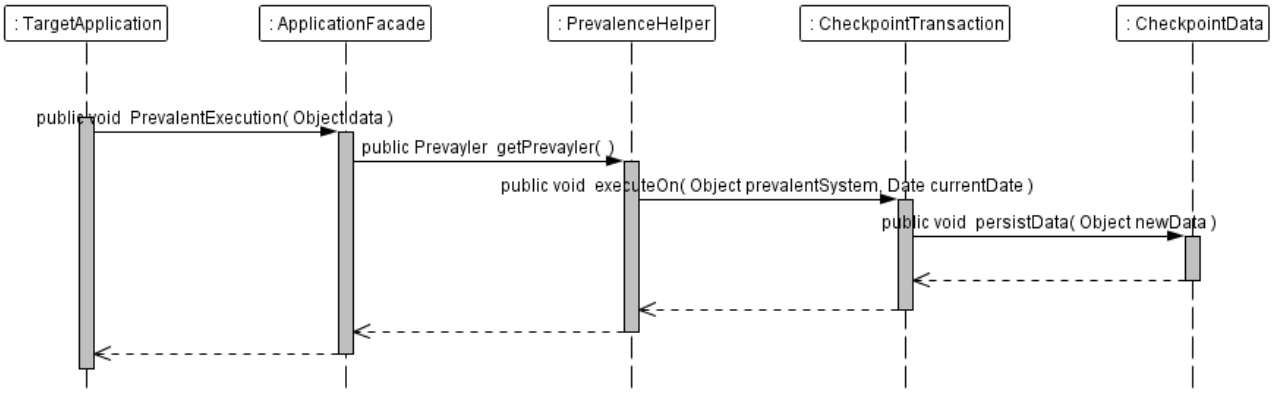


Figura 5.10: Diagrama de Seqüência para o método `PrevalentExecution`

```
41                CheckPrevalentLog.getInstance().logError("PrevalenceHelper", "setPrevayler↩
                      ", "Caught an ClassNotFoundException when setting the Prevayler.", cnf↩
                      );
42            }
43        }
44    }
```

5.5.7 `ApplicationFacade`

A aplicação-alvo a ser executada dentro do modelo de prevalência proposto pelo mecanismo de *checkpoint* deve implementar uma interface e estender essa classe, como visto anteriormente. A classe `ApplicationFacade` estende a classe `PrevalenceHelper`, disponibilizando métodos com funcionalidades de interesse da aplicação-alvo, para encapsulamento do processamento dentro de uma transação prevalente e também, caso seja interesse da aplicação, realizar a execução de *snapshots*.

5.5.7.1 `public void PrevalentExecution(Object data)`

O método `PrevalentExecution()` tem como responsabilidade encapsular a execução da transação do modelo prevalente. Através desse método, é possível armazenar as informações intermediárias produzidas. O diagrama de seqüência apresentado na figura 5.9 mostra o fluxo observado quando a aplicação-alvo invoca a execução desse método. Nele é possível identificar a iteração entre os diversos objetos do modelo de *checkpoint*, até chegar na classe `CheckpointTransaction`, que encapsula a persistência da informação produzida.

Durante a execução da aplicação-alvo, devem ser realizadas chamadas a esse método para que o resultado produzido até o momento seja persistido. A partir do momento em que a chamada a esse método é realizada, a informação a ser armazenada é encapsulada dentro do conceito de transação através da utilização de instâncias da classe `CheckpointTransaction`.

A aplicação-alvo executada dentro do modelo de prevalência proposto deve, obrigatoriamente, realizar chamadas eventuais ao método `PrevalentExecution()` para ter suas informações persistidas. Para os casos onde não sejam realizadas chamadas a esse método, a aplicação tem sua execução realizada fora do contexto de prevalência. Isso significa que os resultados intermediários produzidos não serão armazenados e, em caso de interrupções ou falhas, a execução será retomada desde o início.

Para os casos onde a aplicação-alvo trabalha produzindo resultados baseados em en-

tradas através de arquivos em disco, o método `PrevalentExecution` mantém atualizado um ponteiro para a posição atual do arquivo. Esse ponteiro é utilizado quando for necessário o reinício de uma aplicação após uma interrupção, para que seja possível identificar o local, a partir do qual, o reinício da atividade deve ser realizada.

5.5.7.2 *public void TakeSnapshot()*

Através desse método, a aplicação-alvo tem condições de chamar, de maneira direta, a realização de *snapshots*. Ao invocar o método `TakeSnapshot()`, todas as informações correntes da aplicação são serializadas em disco, permitindo recuperação e utilização futura.

5.5.7.3 *Listagem de Código*

Listing 5.10: Código da classe `ApplicationFacade`

```
1  package Checkpoint.TargetExtension;
2
3  import Checkpoint.TransactionData.CheckpointTransaction;
4  import Checkpoint.log.CheckPrevalentLog;
5  import java.io.IOException;
6
7  public class ApplicationFacade extends PrevalenceHelper{
8
9      public void PrevalentExecution(Object data) {
10         if (_bufFile != null){
11             _filePtr += 1;
12             data = (Object) _filePtr;
13         }
14         this.getPrevayler().execute(new CheckpointTransaction(data));
15     }
16
17     public void TakeSnapshot(){
18         try{
19             this.getPrevayler().takeSnapshot();
20         }
21         catch (IOException io){
22             CheckPrevalentLog.getInstance().logError("ApplicationExtension", "↩
                   TakeSnapshot", "Failed to take snapshot", io);
23         }
24     }
25
26 }
```

5.5.8 `CheckpointTransaction`

A classe `CheckpointTransaction` tem como principal funcionalidade utilizar a classe `CheckpointData`, persistindo a informação no modelo de prevalência de objetos. Essa classe implementa a interface `Transaction` definida pela solução de prevalência utilizada e, a partir disso, implementa o método `executeOn()`.

5.5.8.1 *public void executeOn(Object prevalentSystem, Date currentDate)*

Quando a ação de persistência é invocada, a solução de prevalência utilizada chama, de maneira indireta, a execução desse método. O `executeOn()` tem a responsabilidade de encapsular e persistir a informação recebida por parâmetro, como é possível ver na linha `17`. Tendo a informação intermediária a ser armazenada, o método utiliza o objeto de prevalência criado para então, através da classe `CheckpointData`, realizar a

persistência da informação.

Listing 5.11: Código da classe `CheckpointTransaction`

```
1  package Checkpoint.TransactionData;
2
3  import java.util.Date;
4  import org.prevayler.Transaction;
5
6  public class CheckpointTransaction implements Transaction {
7
8      private Object _newData;
9
10     private CheckpointTransaction () {}
11
12     public CheckpointTransaction(Object newData) {
13         _newData = newData;
14     }
15
16     public void executeOn(Object prevalentSystem, Date currentDate) {
17         ((CheckpointData)prevalentSystem).persistData(_newData);
18     }
19 }
```

5.5.9 `CheckpointData`

A classe `CheckpointData` é responsável por manter o repositório virtual onde as informações intermediárias produzidas pelo sistema prevalente são armazenadas. É essa classe que será utilizada pela aplicação prevalente, através do modelo proposto, para o armazenamento das informações produzidas.

Na linha 7 está definido um `ArrayList` que é a estrutura de dados onde a informação será armazenada. Quando o método `persistData` é chamado, a informação a ser armazenada é recebida por parâmetro e então adicionada a essa estrutura de dados. Com essa estrutura mantida, durante a execução do sistema prevalente, é possível identificar as informações já armazenadas, bem como adicionar novas informações.

A classe `CheckpointData` é atualizada pela classe `CheckpointTransaction`. Nessa operação, as novas informações intermediárias produzidas através das transações prevalentes são adicionadas à estrutura de dados mantida em memória. Da mesma forma, esse objeto pode ser referenciado pela `TargetApplicationDecorator` para obtenção das informações já produzidas, úteis quando a aplicação está sendo reiniciada após uma interrupção.

5.5.9.1 `public void persistData(Object newData)`

Uma das principais utilidades dessa classe é a manutenção, em memória, das informações intermediárias produzidas, através da utilização de um `ArrayList`. A manutenção dessas informações é realizada através do método `persistData`.

Ao ser chamado, o método espera receber como parâmetro um objeto. Essa informação, recebida por parâmetro, deve ser entendida como o resultado intermediário produzido e deve, portanto, ser adicionado à estrutura de dados.

5.5.9.2 `public Object lastData()`

Quando o reinício de uma aplicação que foi previamente interrompida é feito, o modelo de *checkpoint* proposto precisa buscar por possíveis informações já produzidas. Como definido na proposta, o primeiro lugar em que esses resultados intermediários são buscados é na memória.

A busca por resultados intermediários em memória é realizada através do método `lastData`. Esse método utiliza a estrutura de dados do `arrayList` para verificar a última informação adicionada e retornar o conteúdo. Dessa forma, através desse método, a aplicação-alvo pode ser reiniciada com o último resultado válido produzido.

5.5.9.3 `public ArrayList fullDump()`

Quando chamado pelo modelo, o método `fullDump()` retorna a estrutura de dados completa, com todos os valores intermediários produzidos. Esse método pode ser utilizado quando deseja-se verificar todos os valores produzidos até o momento.

5.5.9.4 *Listagem de Código*

Listing 5.12: Código da classe `CheckpointData`

```
1  package Checkpoint.TransactionData;
2
3  import java.util.*;
4
5  public class CheckpointData implements java.io.Serializable {
6
7      private ArrayList persistedData = new ArrayList();
8
9      public void persistData(Object newData) {
10         if (newData != null)
11             persistedData.add(newData);
12     }
13
14     public Object lastData() {
15         Object tmp = null;
16         if (persistedData.size() > 0)
17             tmp = persistedData.get(persistedData.size()-1);
18         return tmp;
19     }
20
21     public ArrayList fullDump() {
22         return persistedData;
23     }
24 }
```

5.6 *Checkpoint* para XtremWeb

O modelo de *checkpoint* descrito foi inicialmente projetado para ser adicionado à plataforma XtremWeb, que é um software de código aberto, desenvolvido pela universidade de Paris Süd, para construir *desktop grids* através da utilização de recursos disponíveis dos computadores pessoais. Dentre as principais funcionalidades, estão incluídos suporte a múltiplos usuários, múltiplas aplicações e desenvolvimento de aplicações entre plataformas, de maneira independente (INRIA, 2009).

A idéia para a construção do XtremWeb surgiu de uma requisição feita pelos físicos do observatório Pierre Auger, para que fosse possível atingir e contemplar um processamento cujos requisitos eram muito grandes para serem resolvidos por máquinas comuns (FEDAK et al., 2001). O problema era, basicamente, a execução anual de uma determinada simulação que continha aproximadamente 6×10^5 entradas distintas. Para esse processamento ser viável, seriam necessárias 6×10^6 horas por ano de um computador com 300MHz. Foi dentro desse contexto, que os pesquisadores da universidade de Paris Süd iniciaram o desenvolvimento da plataforma.

Ainda de acordo com (INRIA, 2009) através da utilização do XtremWeb, é possível transformar um conjunto de recursos voláteis, distribuídos através de uma LAN ou mesmo da internet, em um ambiente integrado para a execução de aplicações paralelas.

Na prática, a plataforma oferecida pelo XtremWeb pode ser utilizada para o desenvolvimento de aplicações nos campos das físicas de altas energias, aplicações biomoleculares, aplicações matemáticas e diversos outros domínios científicos ou industriais.

Como definido em (INRIA, 2009), XtremWeb é uma plataforma altamente programável e customizável, o que permite a sua aplicação tanto em soluções baseadas em *bag-of-tasks* ou mestre-escravo. Além disso, essa flexibilidade permite que a plataforma seja configurada para utilizar benefícios específicos da arquitetura onde está sendo executada (clusters, multi-lan e outros), de maneira escalável, gerenciável e muito segura.

Em (TOTH, 2004), o autor cita que o XtremWeb pode ser comparado ao BOINC, visto anteriormente. Nessa comparação, o autor cita que XtremWeb oferece ferramentas para auxiliar na criação de projetos para computação voluntária, assim como o BOINC. Como aspecto que diferencia as duas soluções, (TOTH, 2004) cita que a plataforma criada pela universidade de Paris Süd somente permite que cada cliente receba uma tarefa de cada vez. Nessa abordagem, após processar essa tarefa, o cliente sinaliza o término da execução e envia os resultados, estando apto a solicitar uma nova tarefa.

Em (INRIA, 2009), essa comparação é aprofundada e os autores citam que o XtremWeb pode ser, em termos de comparação, colocado como uma solução intermediária entre o BOINC e o Condor. Comparado com o BOINC, o XtremWeb assume que cada usuário ou nodo participante tem a possibilidade, se autorizado, de submeter *jobs*. Com o BOINC, essa abordagem não é permitida, onde somente os projetos podem submeter tarefas para a aplicação. Se comparado com o Condor, a plataforma XtremWeb assume que os recursos computacionais podem ser difundidos através da internet. A maior diferença entre as duas soluções é o fato de que o XtremWeb é baseado em um modelo onde os *workers* buscam tarefas do servidor, enquanto que o Condor escolhe a melhor máquina para executar determinada tarefa.

O objetivo principal da solução desenvolvida pela universidade de Paris é, baseado em (FEDAK et al., 2001), criar uma plataforma onde fosse possível tirar o maior proveito possível do conceito de computação global. Isso significa, disponibilizar mecanismos para que máquinas, distribuídas através do mundo pela internet, pudessem colaborar, de maneira transparente, segura e simples, para a solução de problemas e, além disso, fossem capazes de submeter novos problemas. Dentre as principais preocupações, estão:

- aumentar ou reduzir a quantidade de componentes (servidores, largura da banda de rede, escravos) do ambiente, de acordo com a aplicação;
- alta performance e execução segura (para todos os participantes);
- algoritmos de escalonamento devem ser responsáveis pelo gerenciamento da carga de trabalho;
- impacto do tipo da aplicação a ser executada.

5.6.1 Segurança e Execução de Código Nativo

O XtremWeb busca cumprir as atividades que lhe são propostas em um tempo aceitável. Em contraponto, é necessário também garantir aos participantes da grade segurança e confiabilidade na execução das tarefas. Para endereçar ambos os requisitos, o XtremWeb

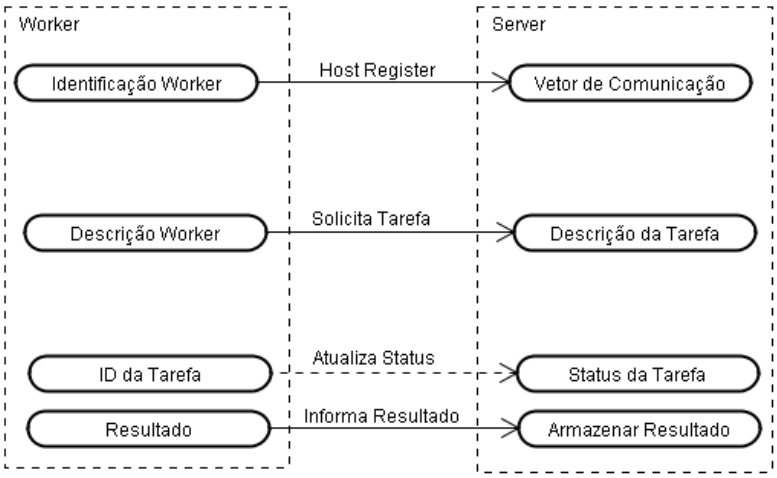


Figura 5.11: Protocolo de Comunicação entre workers

é escrito em Java, linguagem que oferece a proteção e a portabilidade da máquina virtual. Para cumprir com o objetivo de desenvolver as atividades com boa performance, todo o código final e as requisições de comunicações diretas com os sistemas operacionais são feitas de maneira nativa. No detalhamento dos *workers*, essa questão é apresentada com maiores detalhes.

5.6.2 Comunicação entre *Workers* e *Servers*

Desde o planejamento inicial do XtremWeb, a sua execução foi prevista para ser realizada em ambientes globais, através da internet, por exemplo. Para facilitar o *deployment* e configuração, toda e qualquer comunicação necessária parte dos *workers*. Essa abordagem se torna interessante, quando se pensa que em ambientes de computação globais, se as requisições partissem de um servidor (fora da rede local de um dos participantes, por exemplo), poderiam ser bloqueadas por *firewalls*.

De acordo com (FEDAK et al., 2001), o protocolo de comunicação a ser utilizado entre os *workers* e os servidores é totalmente independente da camada de comunicação. Essa camada, responsável por determinar como será estabelecida a comunicação entre os participantes, pode variar desde um modelo generalista como *sockets* via TCP-UDP/IP até um mecanismo de maior nível de abstração como CORBA ou Java RMI. Essa flexibilidade e desacoplamento permitem, também, que sejam utilizados protocolos específicos, adaptados à realidade do problema a ser endereçado, como SSL ou RSVP. De acordo com (FEDAK et al., 2001), a primeira motivação para essa abordagem é a segurança: com a comunicação *on-sided* a segurança dos *workers* é garantida através da autenticação, protegendo os dados que serão transmitidos. Outra motivação, ainda de acordo com (FEDAK et al., 2001) é a facilidade para desenvolvimento: as políticas de segurança do *dispatcher* são configuráveis, enquanto que as políticas do *worker* não são.

A figura 5.11 mostra como funciona essa camada de comunicação entre os *workers* e os servidores. O complexo fluxo de comunicação representado visualmente pela figura 5.11 pode ser resumido, baseado em (FEDAK et al., 2001), através dos seguintes passos:

1. O cliente solicita autenticação ao último servidor com que teve contato ou ao servidor principal. Com a autenticação realizada, o servidor envia ao cliente um conjunto

de informações (lista de servidores que estão aptos a distribuir tarefas, protocolo e porta a serem utilizados para comunicação) que serão utilizadas para a realização das atividades;

2. Estando autenticado e com informações suficientes para solicitar tarefas, o cliente solicita a um dos servidores uma tarefa para computar. Para isso, o cliente envia, junto com a requisição, informações sobre o seu ambiente, para que o servidor possa então determinar a tarefa que melhor esteja adequada ao cliente;

3. O servidor retorna então ao cliente a descrição da tarefa a ser realizada, bem como as entradas (se aplicável), os binários da aplicação correspondentes ao ambiente do cliente solicitante (baseado nas informações recebidas no passo anterior) e o endereço do servidor que está qualificado a armazenar os resultados que serão gerados;

4. Durante o período em que está processando uma tarefa, o cliente envia ao servidor mensagens indicando que o processamento está ativo. É necessário que o servidor monitore constantemente a chegada dessas mensagens para controlar possíveis *timeouts*. Quando um cliente (*worker*) permanece por muito tempo sem enviar esse tipo de mensagem, o servidor entende que o cliente desistiu ou ficou impossibilitado de prosseguir com o processamento e, a partir disso, a tarefa poderá ser realocada para outro cliente;

5. Por fim, quando o processamento da tarefa termina, os resultados computados são enviados ao endereço disponibilizado ao cliente. Essa chamada é então ecoada para o servidor de origem, que disponibilizou a tarefa geral, para sinalizar o término de parte do trabalho;

6. A partir desse momento, o cliente está apto a solicitar uma nova tarefa para o(s) servidor(es).

A comunicação entre as diferentes partes do XtremWeb pode ser feita através de chamada remota de procedimentos (RPC) e transferência direta de dados. Em linhas gerais, segundo (CAPPELLO et al., 2005), a arquitetura de comunicação utilizada pelo XtremWeb está fundamentada em três camadas de protocolos. O primeiro nível, chamado "Conexão", é dedicado a permitir comunicação entre as entidades participantes, que possam estar protegidas por um *firewall*, atrás de uma NAT ou mesmo via *proxy*. A segunda camada, chamada "Transporte", é responsável pelo transporte das mensagens, de maneira segura e confiável. Por fim, a terceira camada, chamada "Protocolo"é responsável por fornecer suporte a diversos tipos e versões de APIs para RPC.

Gerenciar bloqueios impostos por *firewalls* é a principal responsabilidade da camada de conexão. Um *firewall* com uma configuração padrão, tem como comportamento esperado o bloqueio de mensagens de entrada, exceto as realizadas em portas específicas e por *hosts* determinados. Além disso, geralmente, eles permitem a saída de mensagens, sem maiores restrições. Por essa preocupação, a arquitetura do XtremWeb prevê a utilização de um componente coordenador, que pode ser visível por outras entidades.

Por essa abordagem, os canais de comunicação nunca são iniciados pelo coordenador, mas sim pelos clientes, *workers* ou qualquer outro participante. Os canais são, portanto, usados para transmitir as mensagens de e para todos os participantes da comunicação. Isso implica, segundo (CAPPELLO et al., 2005) que nenhum participante, exceto o coordenador, precisa ficar a frente do *firewall* ou com portas abertas. As comunicações são

possíveis se o coordenador mantiver portas dedicadas abertas e implementar protocolos específicos. Por exemplo, JavaRMI necessita a porta `rmiregistry` e o XML-RPC utiliza a porta HTTP.

5.6.3 Arquitetura

De maneira geral, a arquitetura proposta e utilizada pelo XtremWeb é composta, basicamente, de três componentes: cliente, *worker* e coordenador. Essa arquitetura está adequada à maioria dos projetos de computação global (INRIA, 2009). Durante a execução, os *workers* buscam o servidor para busca de *jobs* a serem executados; em resposta, o servidor envia um conjunto de parâmetros e também, caso necessário, a aplicação a ser executada (isso acontece para as situações onde a aplicação não esteja armazenada no *worker*). Quando os *workers* terminarem o processamento de sua tarefa, eles acessam a figura do coordenador, para enviar os resultados, que deverão ser centralizados e armazenados. Dependendo do tamanho do resultado, a comunicação entre os *workers* e o componente responsável pela coleta de resultados pode ser feita através de protocolos distintos (INRIA, 2009). Baseado em (INRIA, 2009) e (FEDAK et al., 2001), os *workers* podem processar tarefas e enviar resultados com até 100 MBytes de tamanho.

XtremWeb pode também ser utilizado para construir sistemas centralizados de P2P. Entretanto, de acordo com (INRIA, 2009), o XtremWeb não permite o armazenamento de dados nos recursos de cliente e servidor. Nessa abordagem P2P, XtremWeb pode ser utilizado da seguinte forma: tipicamente, uma ou mais aplicações são armazenadas nos *workers*. Qualquer *worker* pode submeter *jobs*, através do servidor P2P. Nessa abordagem, o *worker* terá comportamento equivalente ao cliente na arquitetura P2P. Os *jobs* submetidos pelo cliente são registrados no servidor e agendados para os *workers* (INRIA, 2009).

Como visto anteriormente e detalhado pelo autor em (FEDAK et al., 2001), o XtremWeb não está limitado a uma arquitetura centralizada. Novas pesquisas migram para uma solução hierárquica. Outro fator relevante citado em (INRIA, 2009), é que os *workers* podem enviar os resultados diretamente aos clientes, para limitar o uso da banda de rede, no processo de comunicação.

XtremWeb segue a visão geral de sistemas distribuídos de larga escala, objetivando tornar um conjunto de recursos não-específicos (possivelmente voláteis), em um ambiente de execução para os serviços (módulos de aplicações, execução ou infra-estrutura) e oferecendo gerenciamento para a estrutura volátil.

Segundo (CAPPELLO et al., 2005), em linhas gerais, a arquitetura do XtremWeb é composta fundamentalmente quatro principais camadas. O papel da primeira camada é agregar recursos não-específicos (*clusters*, PCs particulares, PCs em uma rede privada e outros) para construção de um cluster completo, mas instável (possibilidade de nodos voláteis). A segunda camada tem por objetivo tornar esse cluster instável em uma solução virtual estável. Isso é feito por meio de exposição de menos recursos em relação ao número que realmente existe, devido ao gerenciamento de volatilidade (alguns recursos são mantidos como 'recursos extras', servindo como uma espécie de *backup*). A terceira camada é responsável por criar uma plataforma GC genérica. Por fim, a quarta camada tem como objetivo disponibilizar os módulos de execução para computação paralela, como mestre-escravo ou ambientes MPI. Nessa arquitetura, as aplicações dos clientes são executadas em cima da quarta camada.

Entretanto, essa arquitetura genérica de quatro camadas é transformada, na prática, através da utilização de outras sub-camadas, como apresentado na figura 5.12. Cada

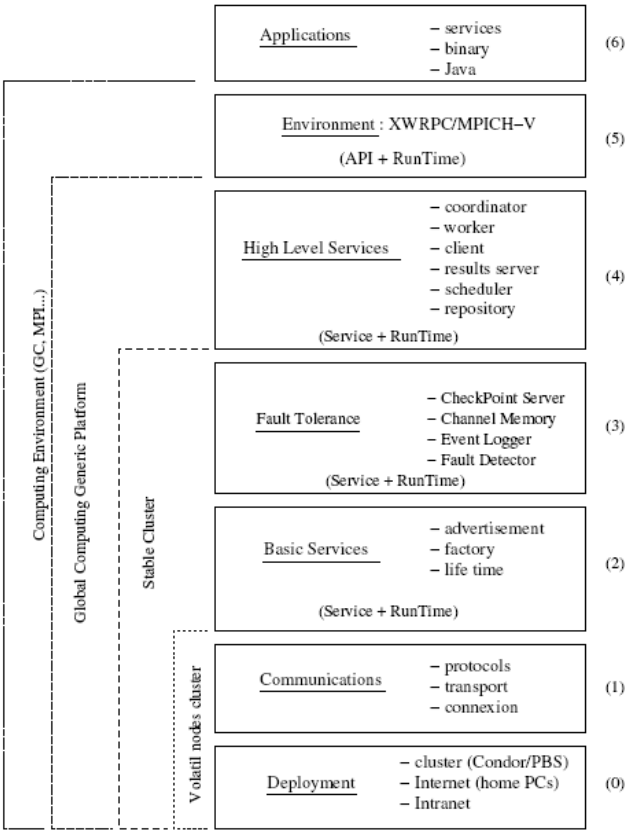


Figura 5.12: Arquitetura de sub-camadas da plataforma XtremWeb

uma dessas sub-camadas desempenham papéis e possuem responsabilidades específicas, dentro da arquitetura do XtremWeb.

O papel da camada mais inferior (0) é possibilitar o *deployment* do componente *Launcher*, que possibilita o reconhecimento de novos nodos ou recursos participantes (como JXTA). A sub-camada (1) encapsula as infra-estruturas de comunicação, possibilitando a troca de mensagens entre os nodos participantes e o componentes *Launcher*. A sub-camada (2) armazena e organiza a estrutura de serviços possibilitando a publicação, descoberta e construção de serviços. A sub-camada (3) encapsula e define recursos e mecanismos para tolerância a falhas. A próxima sub-camada (4) contém serviços de alto-nível (como *schedulers*, repositório de tarefas entre outros). API para computação paralela e ambientes para *runtime* estão na sub-camada (5). Por fim, a sub-camada (6) contém os serviços e os binários das aplicações que serão executadas pelos *workers*.

Nas seções seguintes são apresentadas, de maneira superficial, as principais características no que diz respeito à arquitetura utilizada na codificação dos *workers* e servidores da plataforma.

5.6.3.1 Worker

O *worker* deve ser entendido como o equipamento computacional que irá ceder, de maneira voluntária, a sua capacidade de processamento que não estiver sendo utilizada. De acordo com (INRIA, 2009), cada *worker* participante tem duas funcionalidades principais:

1. Oferecer a sua capacidade de processamento ociosa, como citado anteriormente, para a realização de tarefas de interesse comum, disponibilizadas pelo servidor;

2. Executar instruções de controle, da plataforma XtremWeb, para garantir que as tarefas a serem executadas de maneira voluntária não interfiram e atrapalhem as atividades dos usuários, proprietários das máquinas.

Isso significa que, para manter a aceitação dos conceitos de computação global e computação voluntária a que se vincula a plataforma, os *workers* foram planejados para dar ênfase no respeito do termo voluntários (FEDAK et al., 2001). Em outras palavras, o *worker* foi planejado para somente ser ativado e passar a computar tarefas, no momento correto, ou seja, quando o recurso realmente se torna ocioso. Para isso, de acordo com (FEDAK et al., 2001) e (INRIA, 2009), a arquitetura dos *workers* oferece mecanismos para configurações a serem feitas pelos usuários, além de um ambiente seguro e não intrusivo.

5.6.3.1.1 Implementação

A implementação dos *workers* é escrita, em grande parte, em Java. Entretanto, as chamadas a instruções de funcionalidades específicas do sistema operacional são feitas de maneira nativa, escritas em C. A comunicação entre as duas tecnologias é realizada através de JNI. Segundo (FEDAK et al., 2001), a escolha de Java se deu devido à portabilidade da linguagem. Com isso, todo o *kernel* do XtremWeb pode ser facilmente portado para diversas arquiteturas, sem a necessidade de re-escrever código. Como pode ser visto em (INRIA, 2009), os *workers* estão disponíveis para *download* para plataformas Linux Intel86 e Windows. Além disso, (FEDAK et al., 2001) cita em sua publicação que existe planejamento para portar os *workers* para PowerPC, SPARC/Solaris além de Windows Pocket, para poder ser executado em dispositivos móveis.

5.6.3.1.2 Arquitetura

Em termos gerais, a arquitetura de um *worker* na plataforma XtremWeb pode ser dividida através da exemplificação de cinco componentes, como mostra a figura 5.13.

Quando o recurso está sendo utilizado pelo proprietário, nada relacionado à tarefa da computação voluntária é realizada. Isso reforça a idéia de que, quando não existe recurso ocioso, nenhuma tarefa é realizada e nenhum tempo extra de processamento ou recurso de memória é consumido. Nessa situação, representada pelo lado esquerdo da figura 5.13, apenas existe um monitor de atividades rodando, com baixa prioridade. Esse monitor de atividades existe para monitorar, como o próprio nome sugere, as atividades desenvolvidas no recurso para buscar identificar, baseado nas restrições estabelecidas pelo usuário, o momento em que a máquina estará disponível para computar a tarefa.

No momento em que a máquina entra em um estado disponível para processamento de uma tarefa, o monitor de atividades detecta e aciona um outro conjunto de *threads*, representado no lado direito da figura 5.13. O principal mecanismo é representado por uma *thread* de controle, que mantém: uma nova *thread* de monitoramento, uma *thread* para execução da tarefa e uma *thread* de sinalização. Com as instâncias desses componentes disponíveis, a situação se mantém até que atividades do usuário sejam detectadas. A *thread* de monitoramento é responsável por, através das métricas e restrições estabelecidas pelos usuários, determinar o momento em que o recurso deixa de estar disponível para a grade voluntária e passa a ser de uso do usuário. A *thread* de execução é o mecanismo central, responsável por processar e executar os binários da tarefa, juntamente com seus parâmetros de entrada. A *thread* de sinalização, por fim, é responsável por mandar mensagens ao servidor indicando que o processamento está ativo e sendo executado, na

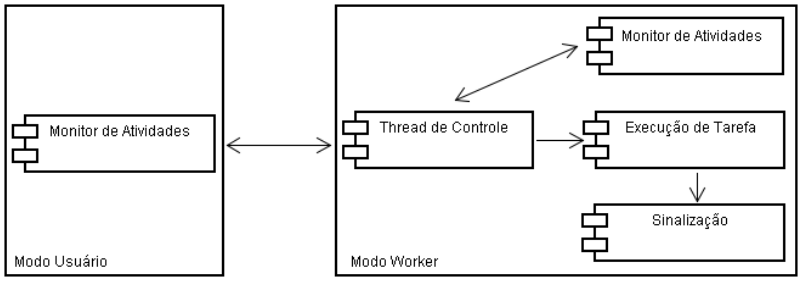


Figura 5.13: Arquitetura de um Worker

medida do possível.

Essa abordagem de concentrar as *threads* de serviço através de uma *thread* de controle é especialmente útil. Com ela, em caso de o usuário retomar as atividades do recurso, a *thread* central morre, levando consigo todas as *threads* de serviço. Com isso, evitam-se *deadlocks* e não existem riscos de tornar o recurso instável.

5.6.3.2 Server

Os servidores (ou coordenadores) do XtremWeb são componentes responsáveis pelo armazenamento das aplicações e tarefas submetidas pelos clientes. Essas aplicações são armazenadas na forma de binários pré-compilados para diversas arquiteturas. Outra responsabilidade dos servidores é disponibilizar, para os clientes, as tarefas e aplicações para serem computadas e processadas pelos *workers*.

A camada intermediária do XtremWeb implementa um conjunto de serviços para coordenação que, segundo (CAPPELLO et al., 2005) poderia ser projetada em uma arquitetura distribuída. Entretanto, os autores do XtremWeb decidiram implementar os serviços de maneira centralizada, por quatro motivos, conforme citado em (CAPPELLO et al., 2005):

1. Facilitar o desenvolvimento e *debug* do sistema;

2. Preocupações relacionadas com performance, quando analisando comparações entre serviços

3. centralizados, hierárquicos e totalmente distribuídos;

4. Incertezas teóricas em relação a estrutura a ser utilizada;

Em relação às incertezas, (CAPPELLO et al., 2005) menciona que não existem resultados teóricos sobre a classificação fundamental de sistemas P2P como soluções distribuídas. Justamente devido a essa incerteza, é que não se pode decidir em relação à confiabilidade ou não no desenvolvimento desse tipo de serviço de maneira distribuída.

O coordenador, no XtremWeb, é composto basicamente de três serviços:

1. Repositório de aplicações e serviços;

2. Escalonador;

3. Servidor de resultados;

Todos esses serviços, segundo (CAPPELLO et al., 2005) trabalham em conjunto em torno de um *pool* de tarefas, se mantendo sempre em um estado consistente em relação às tarefas desenvolvidas.

O repositório de serviços e aplicações e o *scheduler* implementam três serviços básicos. O repositório oferece um serviço de publicação dos serviços e aplicações, tornando-os disponíveis para os clientes, através das portas padrões para comunicação (Java RMI, XML-RPC). O *scheduler* gerencia a *factory* de serviços através da criação de instâncias e aplicações nos *workers* e gerencia o ciclo de vida. Segundo (CAPPELLO et al., 2005), o *scheduler* implementa o modelo *pull* para alocação das tarefas. Isso significa que as tarefas são alocadas nos *workers* por demanda, de acordo com a iniciativa dos *workers*. Após o processamento da atividade ou tarefa proposta, os *workers* enviam os resultados processados, que são, por fim, depositados no terceiro serviço oferecido pelos coordenadores: o depósito de resultados.

Segundo (CAPPELLO et al., 2005), o papel principal do coordenador pode ser definido através das seguintes responsabilidades:

1. Desacoplar os clientes dos *workers*;

2. Coordenar a execução das tarefas nos *workers*.

O coordenador aceita a requisição de tarefas de diversos clientes e as distribui, de acordo com a política de escalonamento, transferindo o código da aplicação (se necessário), supervisionando a execução das tarefas, monitorando erros ou desconexões por parte dos *workers*, re-iniciando tarefas que foram interrompidas, coletando resultados de tarefas processadas pelos *workers* e oferecendo os resultados finais aos clientes.

De acordo com (CAPPELLO et al., 2005), o coordenador não é, necessariamente, implementado por um nodo centralizado; ele pode ser baseado em uma solução de serviços distribuídos, instalados nos nodos participantes, juntamente com outros serviços.

5.6.3.2.1 Arquitetura

A arquitetura típica de um servidor da plataforma XtremWeb é composta por alguns componentes básicos. As seções seguintes apresentam esses componentes, detalhando suas funcionalidades e características. Através da figura 5.14, é possível identificar esses componentes, bem como a comunicação existente entre os mesmos.

Pool de Aplicações Como descrito em (FEDAK et al., 2001), XtremWeb foca em aplicações seqüenciais, com múltiplos parâmetros. A distribuição de uma aplicação consiste em prover, aos *workers*, os binários pré-compilados, de acordo com a plataforma do *worker* solicitante. Além disso, é necessária uma breve descrição da aplicação, uma definição de como os parâmetros devem ser interpretados e entendidos pelos *workers* além da especificação dos resultados que são esperados. Para todo esse armazenamento, além do controle de versão e *upgrades* das aplicações disponibilizadas, o *pool* de aplicações é utilizado.

Pool de *Jobs* Um cliente pode submeter uma tarefa para o XtremWeb através da disponibilização de uma referência à aplicação e um conjunto de parâmetros que caracteriza o *job* a ser executado. Parâmetro, de acordo com (FEDAK et al., 2001), pode ser diversos arquivos, leitura através da entrada padrão ou argumentos passados pela linha de comando. Através do *pool* de *jobs* é feito o armazenamento das tarefas. Quando uma tarefa é armazenada na base de dados do XtremWeb, ela recebe um identificador único, que será utilizado para referenciar a tarefa, nos diversos componentes da arquitetura.

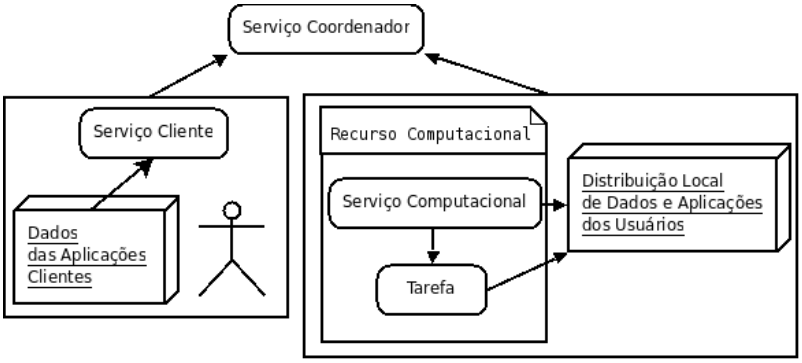


Figura 5.14: Arquitetura de um Servidor

Módulo de Accounting É o módulo responsável por gerenciar e executar o processo de armazenamento das informações sobre as tarefas. Essas informações são: a referência para o *host* que executou a tarefa (ou lista de *hosts*, caso interrupções tenham ocorrido), o identificador do cliente que submeteu a tarefa e diversas outras informações de data e tempo, relacionadas com as execuções.

5.6.3.2.2 Implementação

A implementação do servidor do XtremWeb utiliza, na maior parte do seu código, soluções livres, disponíveis na maior parte dos sistemas operacionais modernos. A principal linguagem utilizada no desenvolvimento é Java, que garante, invariavelmente, a portabilidade da solução.

Na versão original da plataforma, MySql é utilizado como base de dados para armazenamento das informações e configurações. Entretanto, a maior parte das linguagens utilizadas para desenvolver o servidor (Java, PHP e Perl), oferecem bases de dados de acesso direto (como por exemplo, *Java Database Connectivity* ou Perl Database Interface). Isso oferece a possibilidade de troca para outro mecanismo de armazenamento, sem a necessidade de modificar ou realizar ajustes no código da aplicação.

A implementação do servidor oferece ainda aos clientes e administradores, de acordo com (FEDAK et al., 2001), uma interface web para permitir maior iteração com o sistema. Através dela, é permitida a submissão de tarefas, acompanhamento do andamento dos processos (*jobs*) e recebimento dos resultados, entre outros recursos.

5.7 Modificações Propostas no Worker

Para que o mecanismo de *checkpoint* proposto seja utilizado, tendo como enfoque recursos com períodos de disponibilidade curtos, porém freqüentes, o *worker* do XtremWeb precisa sofrer algumas pequenas modificações. Essas modificações dizem respeito à maneira como o *status* de atividade do recurso deve ser monitorado, além da maneira como o processamento da aplicação-alvo deve ser inicializado e interrompido.

Anteriormente, nesse mesmo capítulo, foram apresentados os mecanismos para inicialização e interrupção de aplicações, disponibilizados pelo modelo de *checkpoint* proposto. Essas informações devem ser aplicadas ao *worker* da plataforma de computação voluntária para qual o protótipo foi pensado.

O projeto de solução descrito na seqüencia desse capítulo apresenta as modificações necessárias no *activator model* e também as alterações previstas para a inicialização e

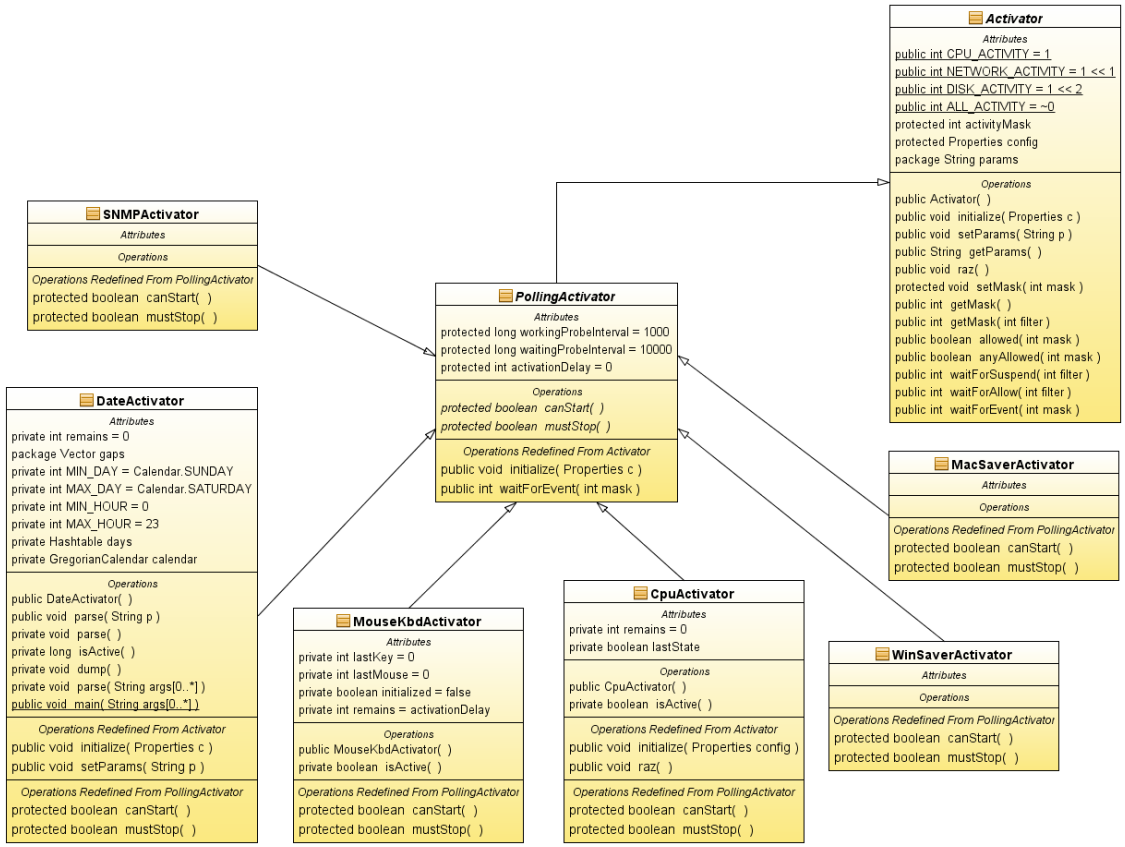


Figura 5.15: *Activator Model* do XtremWeb

interrupção da execução voluntária. Esse projeto tem como base a utilização do modelo de *checkpoint* implementado pelo protótipo acima, sendo utilizado para um modelo de *worker* da plataforma XtremWeb.

5.7.1 *Activator Model*

O modelo de *activators* presente no *worker* do XtremWeb pode ser representado pelo diagrama de classes mostrado na figura 5.15. As subclasses do `Activator` são utilizadas para monitorar a atividade no *worker*. Esse monitoramento pode ser feito por cinco classes distintas, mais o *activator* criado especificamente para o modelo proposto.

5.7.1.1 `DateActivator`

Essa abordagem permite que o processamento do *worker* seja ativado baseado em informações de data e hora, seguindo um modelo parecido com a *crontab*. Isso significa que, através da utilização do `DateActivator`, é possível determinar datas e horários específicos onde as atividades de processamento voluntário, realizado pelo *worker*, podem ser iniciadas.

5.7.1.2 `MouseKbdActivator`

A utilização do `MouseKbdActivator` permite a realização do monitoramento das atividades de mouse e teclado do dispositivo onde o *worker* está instalado. Através desse *activator*, ações sobre o *mouse* e o teclado são monitoradas para, caso aconteçam, permitirem que o processamento voluntário seja interrompido. A utilização do mouse ou do

teclado pelo proprietário do recurso indica, invariavelmente, que o mesmo deseja retomar o controle de execução do seu equipamento, o que indica que todo processamento voluntário em execução deva ser interrompido.

5.7.1.3 `CpuActivator`

De maneira equivalente aos *activators* já apresentados, essa solução oferece o monitoramento das atividades da CPU onde o *worker* está instalado. Através dela, é possível identificar se as atividades de processamento do *worker* podem ser iniciadas ou não, baseado na utilização da CPU.

5.7.1.4 Activators *de Proteção de Tela*

O modelo de *activators* do XtremWeb oferece dois mecanismos para verificar o estado das proteção de tela do recurso voluntário. Esses *activators* são responsáveis por verificar se a proteção de tela está sendo executada. Caso a proteção de tela esteja em execução, significa que o equipamento não está em uso pelo proprietário e, portanto, se apresentam como opções para o processamento voluntário. A partir dessa informação, o *worker* tem condições de avaliar se a aplicação-alvo pode ser executada ou não.

Por padrão, o modelo de *activators* oferece duas soluções nesse sentido:

5.7.1.4.1 `WinSaverActivator`

Activator que monitora a presença de proteção de tela em execução para ambientes Windows.

5.7.1.4.2 `MacSaverActivator`

Activator que monitora a presença de proteção de tela em execução para ambientes Mac OS X.

5.7.1.5 `SNMPActivator`

Dos diversos estados que uma impressora pode apresentar, alguns se tornam mais interessantes para o modelo que deseja-se implementar. Para aplicar um modelo de *checkpoint* dentro de um *worker* que se propõe a ser instalado dentro de uma multifuncional, o modelo de *activators* precisa ser ampliado para contemplar uma nova especialização.

Essa nova proposta é obtida através da observação do *status* geral da impressora, através de operações de SNMP. SNMP pode ser definido, de acordo com (TELECOM, 2009) como um protocolo de gerência típica de redes TCP/IP, da camada de aplicação, que facilita o intercâmbio de informação entre os dispositivos de rede. O SNMP possibilita aos administradores de rede gerenciar o desempenho da rede, encontrar e resolver seus eventuais problemas, e fornecer informações para o planejamento de sua expansão.

O *activator* proposto pelo modelo faz uso de operações SNMP locais para verificar o *status* da impressora, indicando se o processamento das atividades voluntárias pode prosseguir ou se deve ser interrompido. A RFC 1759 (RFC, 2009) define os objetos SNMP que devem estar disponíveis em impressoras para que estas estejam em conformidade com os padrões esperados pelo mercado.

A RFC 1759 define três objetos que podem ser utilizados em conjunto ou individualmente para obter o estado geral da impressora. A tabela 5.5 mostra os objetos `hrDeviceStatus`, `hrPrinterStatus` e `hrPrinterDetectedErrorState` e alguns dos seus possíveis significados:

Tabela 5.5: Configurações de MFPs

Status	hrDeviceStatus	hrPrinterStatus	hrPrinterDetectedErrorState
Normal	*running*	*idle*	sem informação
Busy ou Indisponível	*running*	*printing*	sem informação
Alerta não-crítico	*warning*	*idle* ou *printing*	*lowPaper, lowTonner, ...*
Alerta crítico	*down*	outro	*jammed, noPaper, noToner,...*
Indisponível	*down*	outro	sem informação
Offline	*warning*	*idle* ou *printing*	sem informação

Para o modelo de *checkpoint* proposto, as informações de estado fornecidas para o objeto `hrPrinterStatus` são as que mais interessam na montagem do *activator*. Ainda de acordo com a RFC1759, o objeto `hrPrinterStatus` tem a seguinte especificação:

Listing 5.13: RFC 1759 - Printer MIB

```
1  hrPrinterStatus OBJECT-TYPE
2          SYNTAX INTEGER {
3              other(1),
4              unknown(2),
5              idle(3),
6              printing(4),
7              warmup(5)
8          }
9          ACCESS read-only
10         STATUS mandatory
11         DESCRIPTION
12                 "The current status of this printer device.  When in the idle(1), ↩
                       printing(2), or warmup(3) state, the corresponding hrDeviceStatus ↩
                       should be running(2) or warning(3).  When in the unknown state, the↩
                        corresponding hrDeviceStatus should be unknown(1)."
13         ::= { hrPrinterEntry 1 }
```

Dessa forma, a classe `SNMPActivator` deve observar o objeto `hrPrinterStatus` e somente permitir a execução das atividades da aplicação-alvo quando o resultado das operações de `SNMPget` foram iguais a `idle(3)`. Dessa forma, o *worker* instalado na multifuncional será capaz de monitorar a atividade da impressora e utilizar a capacidade de processamento do recurso para o ambiente de computação voluntária.

Para isso, o método `protected boolean canStart()` realiza operações de `SNMPget` no objeto `hrPrinterStatus` retornando `true` caso o valor retornado seja igual a `3`. Da mesma forma, o método `protected boolean mustStop()` responde com `true` caso o valor de retorno da operação de SNMP seja diferente de `3`.

5.7.2 Adaptando `ThreadLaunch`

A classe `ThreadLaunch`, no *worker* do XtremWeb, também sofre adaptações para permitir a utilização do mecanismo de *checkpoint* proposto. Essa *thread* possui um *looping* infinito que é responsável por aguardar o momento em que o processamento voluntário é permitido, baseado na política definida. Quando um período adequado para o processamento voluntário é encontrado, a `ThreadLaunch` é responsável por retomar a execução das atividades voluntárias no recurso em que está instalada. Também é responsabilidade desse componente identificar o momento em que a computação voluntária não é mais permitida e suspender toda e qualquer atividade nesse contexto.

Como mostra o diagrama de classe representado pela figura 5.2, essa classe utiliza

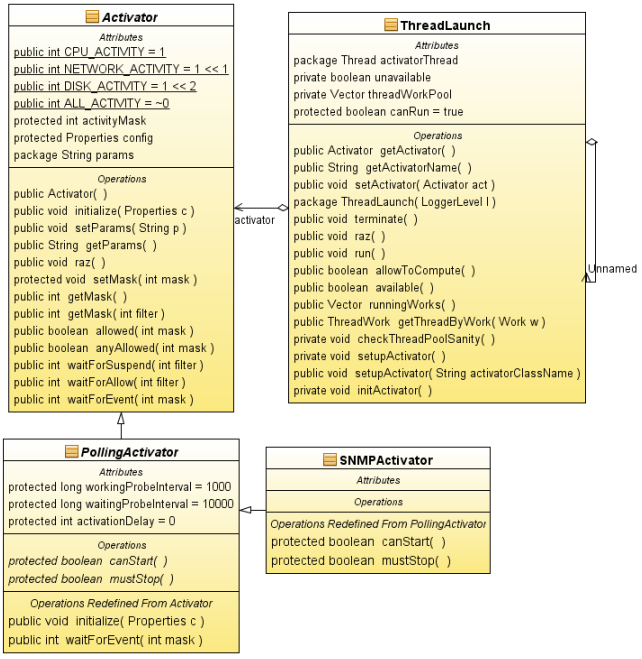


Figura 5.16: Diagrama de Classes da `ThreadLaunch`

um conjunto de objetos do tipo `Activator` para realizar o monitoramento do estado do recurso. Esse monitoramento, no caso do modelo proposto, deve ser realizado utilizando o `SNMPActivator`, descrito anteriormente.

Com o monitoramento da utilização do recurso controlado, a execução da aplicação pode então ser iniciada através da classe `MainFacade`, do componente de *checkpoint* descrito. Dessa forma, a execução da aplicação será iniciada considerando o modelo de prevalência, possibilitando o armazenamento das informações intermediárias produzidas. A interrupção da execução também passa a ser realizada adicionando a chamada para a classe `MainFacade`

5.8 Considerações Finais

Esse capítulo detalhou a implementação do modelo de solução baseado na proposta apresentada nessa dissertação, para permitir a realização de operações de *checkpoint* em memória através da prevalência de objetos. Além do detalhamento do mecanismo de *checkpoint* proposto, esse capítulo viabilizou o estudo em detalhes do *worker* da plataforma XtremWeb, fazendo com que suas principais características e componentes fossem detalhados.

Com a arquitetura do modelo de *checkpoint* apresentada e seus componentes detalhados, o presente capítulo também apresentou as abordagens e premissas que devem ser observadas para a utilização do modelo. Nesse sentido, aplicações-modelo foram utilizadas para exemplificar a aplicabilidade prática do modelo de *checkpoint* proposto.

Com a implementação do mecanismo de *checkpoint* proposto e tendo como base os estudos realizados em cima do *worker* da plataforma de computação voluntária escolhida, foi possível detalhar o projeto para utilização do componente criado, dentro da plataforma em questão.

Este capítulo também permitiu a observação e o detalhamento da utilização de impressoras multifuncionais. Esse tipo de recurso, até então descartado como equipamento vá-

lido para participação em ambientes de computação voluntária, foi estudado tendo como objetivo permitir avaliar com segurança a sua disponibilidade e capacidade de produzir resultados válidos, para esse tipo de ambiente oportunístico.

6 APLICAÇÃO DO MODELO E DESEMPENHO

A implementação do modelo de *checkpoint* com prevalência de objetos descrito no capítulo anterior foi analisada, em um primeiro momento, de maneira teórica, descrevendo os principais componentes e decisões de projeto. Em uma segunda etapa, torna-se interessante avaliar o modelo de uma maneira empírica. Essa validação foi feita através da utilização de uma aplicação real, fato que permitiu que fossem observados detalhes de implementação e *overheads* adicionados à aplicação original, quando da sua execução em um ambiente prevalente. Uma aplicação foi executada sem modificações e os resultados obtidos a partir dessa medição foram comparados a execuções de uma versão modificada, adaptada ao modelo prevalente com os recursos de *checkpoint*. Nesse sentido, esse capítulo apresenta a validação do protótipo do mecanismo de *checkpoint* implementado, para uma aplicação real, validando os principais fatores que podem ser determinantes para a utilização ou não do modelo de *checkpoint*.

6.1 Aplicação-alvo

Para validar a aplicação do modelo de *checkpoint* e possibilitar que resultados de desempenho fossem obtidos, optou-se por utilizar uma implementação da aplicação de *n-gramas*. A aplicação em questão tem por objetivo identificar a usabilidade de determinadas palavras, dentro de um determinado idioma, através da utilização de mecanismos de busca na internet. De acordo com (YAMAMOTO et al., 2001), *n-gramas* são amplamente utilizados como um modelo estatístico para processamento de linguagem, que permite identificar a probabilidade de utilização de determinado termo. Não é objetivo dessa medição, validar se a aplicação está correta, nem mesmo avaliar os resultados obtidos a partir da sua execução; apenas é de interesse verificar o comportamento da mesma quando executada dentro de um modelo de *checkpoint* prevalente.

6.1.1 Código Original

A implementação original do programa que busca pela utilização de *n-gramas* pode ser vista na listagem abaixo. O algoritmo básico implementado por essa aplicação, considera como entrada um arquivo texto onde cada linha representa um *n-grama* a ser pesquisado. Ao ser inicializada, a aplicação utiliza esse arquivo de entrada e conecta ao serviço fornecido pelo *yahoo* para verificar o número de ocorrências do resultado.

Listing 6.1: Aplicação N-Gramas Original

```
1 import com.yahoo.search.*;
2 import java.io.*;
```

```
3
4  public class NGramas {
5
6      private static final String DEFAULT_ID = "ngram001";
7      private static final int DEFAULT_SIZE = 5000;
8      private static final int DEFAULT_OFFSET = 0;
9      private static final String DEFAULT_LANG = "en";
10     private static final BufferedReader DEFAULT_INPUT = new BufferedReader( new ↩
           InputStreamReader( System.in ) );
11     private static BufferedReader in = DEFAULT_INPUT;
12     private static String clientId = DEFAULT_ID;
13     private static int size = DEFAULT_SIZE, offset = DEFAULT_OFFSET, count = 1;
14     private static String lang = DEFAULT_LANG;
15     private static File input;
16     private static SearchClient client;
17     private static WebSearchRequest request;
18     private static WebSearchResults results;
19
20     private void processArguments( ) {
21         try {
22             input = new File("/home/rafael/Mestrado/BaseCode/CheckPrevalent/src/↩
                   ngramlist.txt");
23             in = new BufferedReader( new InputStreamReader( new FileInputStream( input↩
                   ) ) );
24         } catch( FileNotFoundException fnfEx ) {
25             System.err.println( "Error in the input file:\n" + fnfEx.getMessage() );
26             System.exit( -1 );
27         }
28     }
29
30     private void offsetYahoo() {
31         client = new SearchClient( clientId );
32         request = new WebSearchRequest("Empty");
33         request.setType( "phrase" );
34         request.setLanguage( lang );
35         request.setResults( 0 );
36     }
37
38     private void search() {
39         String line = "";
40
41         try {
42             line = in.readLine();
43         } catch( IOException ioEx ) {
44             System.err.println( "Error on reading from input:\n" + ioEx.getMessage() )↩
                   ;
45             System.exit( -1 );
46         }
47         while ( line != null && !line.equals( "" ) ) {
48             try {
49                 if (count > offset && count <= offset + size) {
50                     request.setQuery( line );
51                     results = client.webSearch(request);
52                     System.out.println( line + " " + results.getTotalResultsAvailable↩
                           () );
53                 }
54                 line = in.readLine();
55                 count++;
56             } catch (Exception e) {
57                 System.err.println( "Exception caught." );
58                 System.err.println( e.getMessage() );
59             }
60         }
61     }
62
63     public static void main(String[] args) {
64         NewClass obj = new NewClass();
65         obj.processArguments( );
66         obj.offsetYahoo();
67         obj.search();
68     }
69 }
```

6.1.2 Modificações

Como citado no capítulo que descreve o modelo de *checkpoint* proposto, a aplicação a ser executada dentro desse contexto precisa sofrer algumas modificações. A primeira modificação a ser realizada é alterar a definição da classe para estender a classe `ApplicationFacade` e implementar a interface `ITargetApplication`, como feito na linha `6` da listagem abaixo. A partir dessa modificação, o método `main` deve ser alterado conforme mostra a linha `69`, para determinar ao mecanismo de *checkpoint* o ponto de início da aplicação. Além disso, o método `InitializeProgram` é definido na linha `25`, a variável `StartupValue` inicializada de acordo. Por fim, a linha `58` mostra a chamada ao método `PrevalentExecution`, que possibilita aos dados produzidos serem encapsulados ao modelo de prevalência. Esse conjunto de modificações pode ser observado na listagem de código abaixo.

Listing 6.2: Aplicação N-Gramas Modificada

```
1  import Checkpoint.TargetExtension.ApplicationFacade;
2  import Checkpoint.TargetExtension.ITargetApplication;
3  import com.yahoo.search.*;
4  import java.io.*;
5
6  public class NGramas extends ApplicationFacade implements ITargetApplication  {
7
8      private static final String DEFAULT_ID = "ngram001";
9      private static final int DEFAULT_SIZE = 5000;
10     private static final int DEFAULT_OFFSET = 0;
11     private static final String DEFAULT_LANG = "en";
12     private static final BufferedReader DEFAULT_INPUT = new BufferedReader( new ↩
           InputStreamReader( System.in ) );
13     private static BufferedReader in = DEFAULT_INPUT;
14     private static String clientId = DEFAULT_ID;
15     private static int size = DEFAULT_SIZE, offset = DEFAULT_OFFSET, count = 1;
16     private static String lang = DEFAULT_LANG;
17     private static File input;
18     private static SearchClient client;
19     private static WebSearchRequest request;
20     private static WebSearchResults results;
21
22     private File testFile = new File("/home/rafael/Mestrado/BaseCode/CheckPrevalent/↩
           src/ngramlist.txt");
23     private BufferedReader StartupValue;
24
25     public void initializeProgram() {
26       try{
27             StartupValue =  new BufferedReader(new FileReader(testFile));
28         } catch (FileNotFoundException fnf){ }
29     };
30
31     private void processArguments( ) {
32        in = StartupValue;
33     }
34
35     private void offsetYahoo() {
36         client = new SearchClient( clientId );
37         request = new WebSearchRequest("Empty");
38         request.setType( "phrase" );
39         request.setLanguage( lang );
40         request.setResults( 0 );
41     }
42
43     private void search() {
44         String line = "";
45         try {
46             line = in.readLine();
47         } catch( IOException ioEx ) {
48             System.err.println( "Error on reading from input:\n" + ioEx.getMessage() )↩
```

```
                ;
49              System.exit( -1 );
50          }
51          while ( line != null && !line.equals( "" ) ) {
52              try {
53                  if (count > offset && count <= offset + size) {
54                      request.setQuery( line );
55                      results = client.webSearch(request);
56                      String tmp = line + " " + results.getTotalResultsAvailable();
57                      System.out.println( tmp );
58                      this.PrevalentExecution(tmp);
59                  }
60                  line = in.readLine();
61                  count++;
62              } catch (Exception e) {
63                  System.err.println( "Exception caught." );
64                  System.err.println( e.getMessage() );
65              }
66          }
67      }
68
69      public void start() {
70          processArguments( );
71          offsetYahoo();
72          search();
73      }
74  }
```

6.2 Ambiente de Execução

A aplicação-alvo descrita anteriormente foi executada dentro de um ambiente que simula uma impressora multifuncional com 800MHz de capacidade de processamento e 768 MB de memória. Nesse ambiente onde o experimento foi executado, Microsoft Windows é utilizado como sistema operacional, rodando a versão 1.4 da plataforma Java.

O mecanismo de *checkpoint* proposto foi instalado nesse ambiente, juntamente com a aplicação-alvo descrita acima. Para possibilitar a interrupção e reinício das atividades, foi instalada também uma aplicação gráfica simples. Através dela, qualquer aplicação adaptada ao modelo do *checkpoint* pode ser executada e interrompida - essa aplicação somente é útil para facilitar a realização dos testes. Para os testes aqui descritos, a aplicação *n-gramas* foi utilizada conforme descrito nos cenários.

Os resultados foram obtidos através da utilização da ferramenta de *profiler* da IDE Netbeans. A execução da aplicação original e da aplicação alvo foi observada e monitorada através dessa ferramenta, que se apresenta como uma solução interessante para obtenção de informações durante o *runtime* da aplicação. Segundo (NETBEANS, 2009), o *profiler* permite monitorar, com baixo *overhead*, informações referentes ao estado das *threads* em execução, além do desempenho da CPU e utilização de memória.

6.3 Cenários de Testes

O conjunto de testes realizados para a medição do desempenho do modelo pode ser dividido em duas etapas. A primeira delas é a execução simples da aplicação *n-gramas*, sem nenhuma modificação referente ao modelo de *checkpoint*. A segunda etapa contempla a execução da mesma aplicação, dentro do conceito de *checkpoint* com prevalência proposto. Para a terceira etapa, foram realizadas medições referentes à execução da aplicação dentro do modelo de *checkpoint* proposto, contemplando a atividade de recuperação das informações intermediárias produzidas. As medições realizadas na aplicação original

são úteis quando comparadas às medições realizadas na aplicação modificada, para medir o *overhead* adicionado quando deseja-se utilizar o mecanismo de *checkpoint*.

Para essas três etapas, foram verificadas as seguintes informações:

1. Desempenho do *garbage collector*;
2. Utilização de memória;
3. Desempenho para realização de *snapshots*;
4. Tamanho das informações geradas em disco;
5. *Hotspots*[1] da aplicação.

Os cenários descritos foram realizados utilizando intervalos de tempo de dois minutos. Isso significa que a aplicação original (para a primeira etapa de testes) e a aplicação modificada (para a segunda e terceira etapas) foram executadas durante dois minutos e interrompidas. Nessa abordagem, dentro do modelo modificado, a aplicação foi executada dentro do modelo de *checkpoint* com prevalência por dois minutos e interrompida. Quando da execução do terceiro cenário, os valores produzidos pela segunda etapa são recuperados e o reinício adequado para que valores intermediários sejam utilizados.

6.4 Resultados Obtidos

Para verificar a usabilidade do modelo de *checkpoint* com prevalência de objetos proposto nessa dissertação, uma aplicação real foi submetida à execução dentro do conceito proposto. Essa execução foi monitorada por uma ferramenta de perfilamento, de onde diversas informações puderam ser extraídas e analisadas.

A primeira informação avaliada foi a utilização de memória, para verificar se a utilização do mecanismo de *checkpoint* poderia causar danos sensíveis ou até mesmo a quebra da execução da aplicação através de *exceptions*. De maneira relacionada à utilização da memória, foi observada a realização de chamadas ao processo de *garbage collector*, para verificar possíveis *memory leaks* e alocação indevida de objetos em memória. Além dos conceitos relacionados com memória, foi medido o *overhead* total de execução adicionado pelo modelo de *checkpoint* à aplicação original, além do tamanho das informações produzidas e armazenadas em disco pelo mecanismo de *checkpoint* proposto. Essas informações foram medidas para a aplicação original, sem modificações e para a versão adaptada ao modelo de *checkpoint* com prevalência de objetos. Os resultados obtidos por essas medições foram comparados e as conclusões descritas nas seções abaixo.

6.4.1 Utilização de memória *heap*

Um conceito importante de ser monitorado quando desenvolvemos aplicações, em especial em Java, é a alocação de objetos em memória. Quando a máquina virtual do Java inicia a sua execução, ela aloca uma quantidade pré-estabelecida de espaço para memória *heap*. Conforme a aplicação é executada, objetos são criados e armazenados dentro desse espaço e, quando não são mais necessários, o *garbage collector* do Java elimina essas informações, recuperando o espaço para ser novamente utilizado.

[1] Por *hotspot* devemos entender como o local da aplicação onde o maior número de instruções foi executada ou o local onde o desempenho foi mais prejudicial, em termos de tempo, durante a execução do programa.

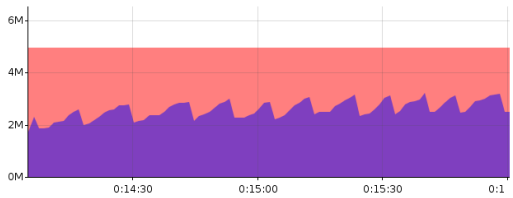
Como base nisso, o conceito de *espaço livre em memória* para aplicações Java pode ser definido como a quantidade de memória que foi alocada e ainda está disponível para uso. Quando essa quantidade de memória livre se aproxima de zero, as frequentes chamadas ao processo de *garbage collector* passam a impactar a execução normal da aplicação, em termos de performance. Quando o espaço alocado termina, ou seja, a quantidade de memória disponível é igual a zero, a JVM assinala que não é mais possível criar novos objetos, levantando uma exceção de *out of memory*. Como resultado visível disso, o sistema deixa de ser executado e a aplicação em questão é interrompida.

Abaixo são apresentadas as medições de utilização de memória *heap* para a aplicação-modelo original e para a aplicação modificada para o conceito de *checkpoint*. Os gráficos mostram, durante o tempo de execução da aplicação, o comportamento referente à utilização da memória por parte da aplicação. O aspecto a ser avaliado é a diferença entre as duas aplicações, para verificar o impacto adicionado pelo modelo de *checkpoint* em relação à aplicação original.

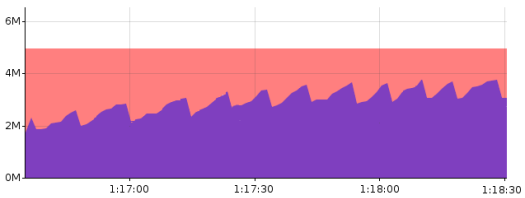
O contexto de execução tanto da aplicação original quanto da aplicação modificada tem `5MB` de memória alocada. Essa informação é representada pelo *layer* vermelho nos gráficos da figura 6.1(a) e 6.1(b). O gráfico 6.1(a) representa a quantidade de memória que foi efetivamente alocada para a execução da aplicação original. Ao longo do tempo em que a aplicação estava rodando, a utilização máxima de memória para alocar os objetos criados para a aplicação não chegou na marca de `3.5MB`.

A aplicação modificada foi executada dentro do mesmo contexto, com a mesma duração de execução. O gráfico 6.1(b) mostra as informações de alocação de memória para a aplicação-modificada.

Comparando os gráficos de alocação de memória *heap* dos dois cenários de execução, é possível verificar que a utilização do modelo de *checkpoint* com prevalência adiciona um pequeno *overhead* em termos de quantidade de memória alocada. Essa informação é esperada e aceitável, uma vez que os resultados intermediários produzidos são mantidos em memória, para uma eventual recuperação futura.



(a) Memória da Aplicação Original



(b) Memória da Aplicação Modificada

6.4.2 Desempenho do *garbage collector*

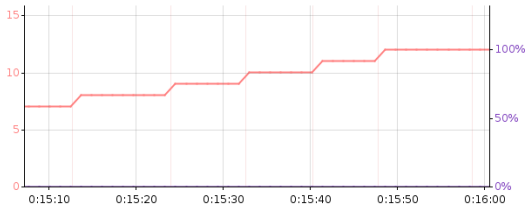
Como citado anteriormente, a utilização do *garbage collector* é fundamental para aplicações Java, a fim de remover objetos alocados em memória que não possuem mais referências e, portanto, não são mais utilizados pela aplicação. Entretanto, a utilização de chamadas frequentes ao GC causam *overhead* no desempenho da aplicação e devem ser minimizadas.

O desempenho do GC para a aplicação original e para a modificada foi avaliado sob dois conceitos: a primeira medição avalia o tempo necessário para a execução do processo de GC e a segunda medição mostra o número de objetos que sobrevivem ao processo de GC. A informação da primeira medição nos mostra o tempo que é adicionado à execução

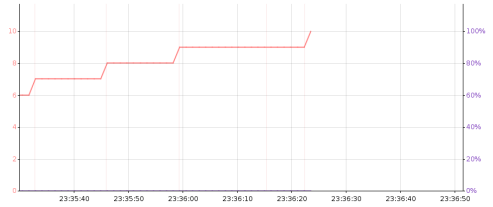
normal da aplicação, para que o processo de GC possa ser realizado no sistema. Já a segunda medição mostra a quantidade de objetos alocados em memória que sobrevivem ao processo de GC. Essa segunda informação é especialmente importante pois possibilita verificar o crescimento da quantidade de objetos alocados em memória ao longo da execução da aplicação.

O gráfico 6.1(c) mostra a utilização das operações do processo de *garbage collector* para a aplicação original. Através da análise sobre esse gráfico, é possível verificar que o tempo das operações do *garbage collector* não impacta sobre o processamento da aplicação. Entretanto, é possível verificar que a aplicação modelo avaliada apresenta uma situação que configura um potencial *memory leak*, posto que a quantidade de objetos alocados cresce gradativamente ao longo da execução.

A aplicação modificada, quando executada dentro do mesmo contexto, apresenta o comportamento descrito no gráfico 6.1(d). De maneira semelhante ao comportamento da aplicação original, a aplicação executada dentro do modelo de *checkpoint* proposto não apresenta chamadas ao *garbage collector* que possam influenciar no desempenho de execução da aplicação. Da mesma forma, o comportamento de possuir uma quantidade crescente de objetos alocados permanece, da mesma forma que na aplicação original. Entretanto, é importante destacar que a quantidade de objetos nessa condição permanece inalterado. Isso significa que o mecanismo de *checkpoint* em si, não está introduzindo problemas de *memory leak* para as aplicações.



(c) Utilização GC da Aplicação Original



(d) Utilização GC da Aplicação Modificada

Comparando os gráficos 6.1(c) e 6.1(d), é possível inferir que o comportamento em relação à utilização do *garbage collector* e da quantidade de objetos alocados em memória ao longo da execução é semelhante para a aplicação original e para a modificada. Dessa forma, é correto afirmar que as aplicações que optarem por utilizar o mecanismo de *checkpoint* proposto não irão sofrer impactos em termos de utilização de operações de *garbage collector*, nem mesmo terão objetos desnecessários alocados em memória.

6.4.3 Realização de *Snapshots*

O processo de realização de *snapshots* periódicos é importante para permitir que as informações intermediárias produzidas possam ser armazenadas em meio físico, evitando que os resultados sejam perdidos em caso de falhas. Esse processo é realizado de maneira periódica, dentro de um intervalo de tempo pré-estabelecido e configurado no mecanismo de *checkpoint*. Entretanto, a realização desse processo de armazenamento em disco envolve a serialização das informações e, como visto ao longo dessa dissertação, a operação de IO pode se apresentar como gargalo para a aplicação.

6.4.3.1 Criação de um snapshot

Com o objetivo de identificar o custo associado à realização do processo de *checkpoint* em disco, a execução dos métodos envolvidos foi monitorada e é apresentada abaixo. A

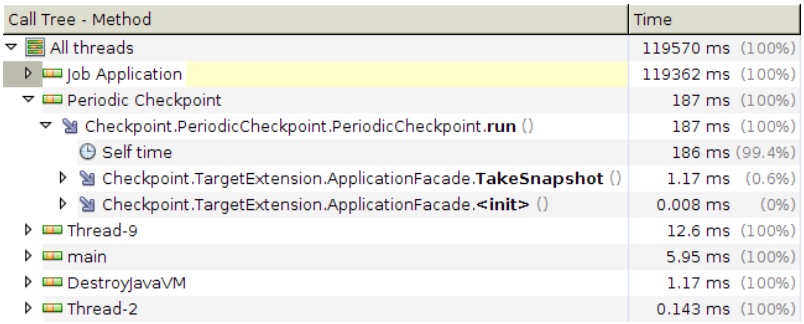


Figura 6.1: Árvore de Chamadas do *checkpoint* Periódico

aplicação original não apresenta essa estatística pois ela não está encapsulada no conceito de *checkpoint* com prevalencia de objetos e, portanto, não possui um mecanismo de *checkpoint* periódico em disco associado.

A figura 6.1 apresenta a estrutura para a chamada do processo de execução periódica de *snapshots*. Através dessa medição, é possível verificar que o tempo total para execução do método que realiza um *snapshot* é de `187ms`. Esse valor deve ser entendido como o tempo que será adicionado ao tempo normal da aplicação, para que os resultados intermediários produzidos e residentes em memória possam ser serializados em disco.

6.4.3.2 Informações em disco

Como detalhado no capítulo anterior sobre o modelo de *checkpoint* proposto por essa dissertação, em intervalos de tempo configurados pelo administrador do sistema, execuções de *checkpoints* em disco periódicas são realizadas. Quando a execução de um *snapshot* é realizada, as informações intermediárias produzidas são serializadas em disco. Esse processo envolve a criação de arquivos em disco que podem ter seu tamanho aumentado, de acordo com a quantidade de informações produzidas. O tamanho do arquivo está diretamente relacionado com o intervalo de *checkpoint* utilizado. Isso significa que quanto maior o tempo de intervalo utilizado, mais informações são produzidas e armazenadas em memória e, por conseqüência, maior a quantidade de informações serializadas.

Para verificar o comportamento relacionado com as operações de *snapshot*, o tamanho físico dos arquivos produzidos em disco foi observado ao longo da execução da aplicação. Ao longo da execução da aplicação, as operações de *snapshot* são realizadas e os dados até então produzidos são serializados em disco. A medida que a execução transcorre, mais resultados são produzidos e, por consequencia, maiores ficam os arquivos de *snapshot* produzidos.

O gráfico 6.2 permite ter uma idéia das informações produzidas em disco, quando a aplicação *n-gramas* modificada é executada dentro do contexto anteriormente citado. Para essa medição, o intervalo de tempo configurado para a realização de *snapshots* foi de `30` segundos. Isso significa que a cada `30` segundos, as informações armazenadas na memória foram serializadas para disco.

6.4.3.3 Recuperação das Informações

Essas informações produzidas em disco são especialmente importantes quando a execução normal do equipamento é interrompida devido a falhas. Nessas situações, ao ser retomada a execução de uma atividade voluntária, o mecanismo de *checkpoint* proposto procura por informações já produzidas para retomar a execução. De acordo com o deta-

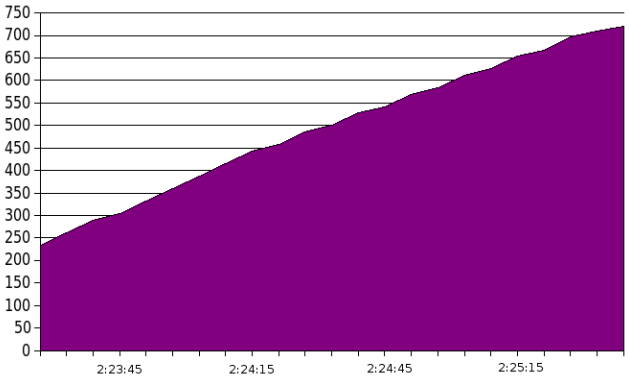


Figura 6.2: Tamanho Físico do Snapshot em Disco

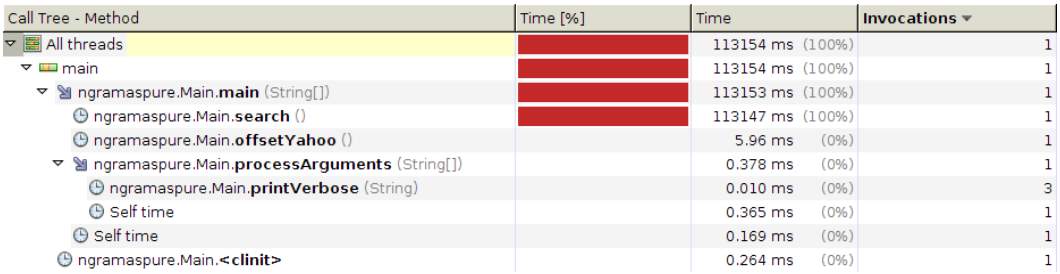
Call Tree - Method	Time
Job Application	116873 ms (100%)
Checkpoint.TargetExtension.ApplicationJob.**run** ()	116873 ms (100%)
Checkpoint.TargetExtension.TargetApplicationDecorator.**RunApplication** ()	116304 ms (99.5%)
NGramas.**start** ()	116221 ms (99.4%)
Checkpoint.TargetExtension.TargetApplicationDecorator.**RestoreInformation** (Class)	63.5 ms (0.1%)
Checkpoint.TargetExtension.TargetApplicationDecorator.**initializeStartupValue** (java.	61.6 ms (0.1%)
Checkpoint.TargetExtension.TargetApplicationDecorator.**PositionateFilePointer** ()	40.3 ms (0%)
Self time	21.3 ms (0%)
Self time	1.86 ms (0%)
Checkpoint.TransactionData.CheckpointData.**lastData** ()	0.014 ms (0%)
Self time	19.4 ms (0%)
NGramas.**<clinit>**	0.229 ms (0%)
NGramas.**initializeProgram** ()	0.143 ms (0%)
NGramas.**<init>** ()	0.127 ms (0%)
Checkpoint.PeriodicCheckpoint.PeriodicCheckpoint.**<init>** ()	0.071 ms (0%)
Checkpoint.TargetExtension.TargetApplicationDecorator.**<init>** (String)	526 ms (0.5%)
Self time	41.9 ms (0%)
Checkpoint.TargetExtension.PrevalenceHelper.**<clinit>**	0.016 ms (0%)
Periodic Checkpoint	379 ms (100%)
main	25.6 ms (100%)
Thread-9	1.41 ms (100%)

Figura 6.3: Tempo de execução para recuperação de *snapshots*

lhamento oferecido no capítulo sobre o modelo, o primeiro local onde as informações intermediárias são produzidas é em memória, sendo seguida por uma busca em disco. Dessa forma, torna-se interessante medir não só o tempo para realizar o *snapshot* e o tamanho das informações produzidas em disco, mas também o tempo necessário para buscar essas informações e recuperá-las em memória para serem, de fato, novamente úteis ao modelo.

Dessa forma, foi medido o tempo necessário para recuperar o arquivo de *snapshot* do disco e realizar as devidas operações de deserialização, recuperando as informações intermediárias produzidas e reposicionando o início da aplicação no último resultado válido produzido.

Na imagem 6.3 é possível verificar que o tempo transcorrido para que os *snapshots* produzidos fossem recuperados do disco e realocados em memória, estando disponíveis para sua efetiva utilização na aplicação em questão, causam um *overhead* de `63.5ms`.



Call Tree - Method	Time [%]	Time	Invocations
All threads		113154 ms (100%)	1
main		113154 ms (100%)	1
ngramaspure.Main.**main** (String[])		113153 ms (100%)	1
ngramaspure.Main.**search** ()		113147 ms (100%)	1
ngramaspure.Main.**offsetYahoo** ()		5.96 ms (0%)	1
ngramaspure.Main.**processArguments** (String[])		0.378 ms (0%)	1
ngramaspure.Main.**printVerbose** (String)		0.010 ms (0%)	3
Self time		0.365 ms (0%)	1
Self time		0.169 ms (0%)	1
ngramaspure.Main.**<clinit>**		0.264 ms (0%)	1

Figura 6.4: *Hotspots* da Aplicação Original

6.4.4 *Hotspots* das Execuções

Como visto anteriormente, para que seja possível a utilização do conceito de *checkpoint* com prevalência de objetos proposto nessa dissertação, algumas modificações devem ser realizadas na aplicação original. Essas modificações, detalhadas anteriormente nesse mesmo capítulo, poderiam causar *overhead* prejudicando o desempenho da aplicação.

Para verificar possíveis danos causados ao desempenho da aplicação, os principais *hotspots* da aplicação original foram medidos. Da mesma forma, a execução dos métodos da aplicação adaptada foi observada. A partir dos dados obtidos, é possível realizar uma comparação das execuções que possibilita avaliar o tempo (*overhead*) adicionado pelo mecanismo de *checkpoint* proposto à aplicação original.

A imagem 6.4 mostra os métodos executados pela aplicação original e seus respectivos tempos de execução. Observando esses valores, é possível perceber que apenas instruções de negócio da aplicação são executados.

Nessa mesma linha, a imagem 6.5 mostra os *hotspots* de execução da aplicação modificada. Através da análise dessas informações, é possível identificar que o tempo de execução da aplicação original sofreu modificações, devido aos novos métodos responsáveis pela manutenção de prevalência das informações. Isso significa que, além dos métodos mostrados na imagem 6.4, novas instruções foram realizadas, o que gera um acréscimo de tempo na execução da aplicação.

6.5 Considerações Finais

O capítulo de aplicação do modelo e desempenho apresentou uma avaliação empírica de uma aplicação real, dentro do modelo de *checkpoint* proposto. Para que esse objetivo fosse atingido, foi utilizada uma aplicação *bag-of-tasks* bastante utilizada para pesquisa e processamento de linguagem natural, que utiliza mecanismos de busca na internet para verificar a utilização de termos idiomáticos. Essa aplicação original foi executada dentro de um ambiente restrito em termos de memória, disco e processamento e teve seus principais conceitos avaliados. Da mesma forma, foram realizadas pequenas adaptações na aplicação utilizada, para que esta pudesse estar adaptada ao modelo de *checkpoint* com prevalência de objetos proposto. A aplicação modificada foi realizada dentro do mesmo contexto de execução, tendo as mesmas informações capturadas e comparadas às informações originais.

Como foi visto nas medições apresentadas, o tempo para o armazenamento das informações produzidas, quando em caso de interrupção, não é considerável nem mesmo quando envolve a serialização das informações em disco. Em contrapartida, o *overhead*

Hot Spots - Method	Self time	Invocations
NGramas.**search** ()	116482 ms (97.4%)	1
Checkpoint.TargetExtension.ApplicationFacade.**PrevalentExecution** (Object)	2545 ms (2.1%)	43
Checkpoint.TargetExtension.PrevalenceHelper.**setPrevayler** ()	243 ms (0.2%)	1
Checkpoint.PeriodicCheckpoint.PeriodicCheckpoint.**run** ()	186 ms (0.2%)	1
Checkpoint.TargetExtension.TargetApplicationDecorator.**RunApplication** ()	37.1 ms (0%)	1
Checkpoint.TargetExtension.TargetApplicationDecorator.**initializeStartupValue**..	31.1 ms (0%)	1
NGramas.**offsetYahoo** ()	13.0 ms (0%)	1
org.prevayler.foundation.DeepCopier$Receiver.**run** ()	12.6 ms (0%)	1
Checkpoint.TargetExtension.MainFacade.**RunApplication** (String)	3.96 ms (0%)	1
Checkpoint.TargetExtension.ApplicationJob.**run** ()	3.81 ms (0%)	1
Checkpoint.TargetExtension.TargetApplicationDecorator.**RestoreInformation** (...	2.7 ms (0%)	1
MainInterceptor.**main** (String[])	1.88 ms (0%)	1
Checkpoint.TargetExtension.ApplicationFacade.**TakeSnapshot** ()	1.17 ms (0%)	1
java.lang.ApplicationShutdownHooks.**run** ()	1.11 ms (0%)	1
Checkpoint.TransactionData.CheckpointTransaction.**executeOn** (Object, java.ut...	0.368 ms (0%)	86
Checkpoint.TransactionData.CheckpointData.**<init>** ()	0.278 ms (0%)	1
Checkpoint.TransactionData.CheckpointData.**persistData** (Object)	0.191 ms (0%)	86
NGramas.**<clinit>**	0.185 ms (0%)	1
java.util.logging.LogManager$Cleaner.**run** ()	0.144 ms (0%)	1
NGramas.**initializeProgram** ()	0.139 ms (0%)	1

Figura 6.5: *Hotspots* da Aplicação Modificada

total do tempo de execução é um fator que deve ser observado com cuidado. Como visto, a aplicação, quando executada dentro do contexto de *checkpoint*, precisa realizar (de maneira direta ou indireta) chamadas a diversos métodos que não estão diretamente relacionados ao negócio específico da aplicação. De modo geral, é correto considerar que a execução de aplicações com o modelo de *checkpoint* aplicado em impressoras multifuncionais é aceitável.

7 CONCLUSÃO

Essa dissertação apresentou uma proposta para um mecanismo de *checkpoint* com prevalência de objetos, focada para recursos que, além de possuírem capacidades limitadas de memória e processamento, apresentam períodos de disponibilidade curtos, porém freqüentes. Um protótipo de solução foi projetado, implementado e testado em um ambiente que simula as configurações de *hardware* e *software* de uma impressora multifuncional. Esse protótipo foi, então, projetado para ser adaptado dentro do *worker* da plataforma XtremWeb, para oferecer uma solução de *checkpoint* para essa plataforma.

Os resultados das avaliações, realizadas através da simulação da execução de aplicações reais dentro desse modelo, permitiram constatar que a aplicabilidade da solução é viável. Com base nos resultados, foi possível verificar que o modelo não adiciona nenhum tipo de *overhead* considerável, seja em termos de memória, processamento ou utilização de disco, que exceda as expectativas. Dessa forma, é correto assumir que a utilização do mecanismo de *checkpoint* proposto dentro de uma plataforma voluntária como a XtremWeb, pode trazer benefícios em termos de armazenamento de resultados produzidos.

No que diz respeito à utilização de memória, as avaliações realizadas mostram que o modelo de *checkpoint* com prevalência de objetos faz uso de uma quantidade de memória que varia de acordo com a quantidade de tempo em que permanece desenvolvendo a atividade voluntária. Isso significa que quanto mais tempo o recurso estiver processando atividades voluntárias sem realizar operações de *snapshot*, maior será a quantidade de memória alocada para o armazenamento temporário das informações. Dependendo da quantidade de memória do recurso, esse espaço alocado pode tornar-se um problema. Entretanto, como visto nessa dissertação, as impressoras multifuncionais modernas evoluem a cada dia, apresentando recursos de *hardware* cada vez mais próximos dos *desktops*. Dessa forma, o espaço utilizado para esse armazenamento temporário não chega a se apresentar como um problema grave, embora seja um dos fatores críticos do modelo.

Em termos de utilização em disco, o modelo de *checkpoint* proposto usa poucos recursos. Isso se deve ao fato de que a persistência das informações em disco físico é uma abordagem complementar, ao mecanismo principal de persistência em disco. O capítulo que mostrou a avaliação do modelo apresentou medições de utilização de disco. Essas medições mostraram que, conforme esperado, quanto maior a quantidade de informação intermediária produzida, maior o tamanho físico do arquivo produzido em disco. Entretanto, o crescimento dessa informação é tido como aceitável, dentro das configurações dos modelos de impressoras multifuncionais. Isso significa que a utilização do disco físico da impressora para armazenamento de informações de *snapshot* não prejudicam nem interferem na execução e utilização normal do recurso.

Por fim, uma das preocupações consideradas desde o início da pesquisa, era a de não

prejudicar a execução normal do recurso. Isso significa que, ao utilizar o mecanismo de *checkpoint* proposto em uma multifuncional, o tempo para reinício de uma impressão não fosse prejudicado ao interromper o processamento voluntário e armazenar as informações produzidas. Pela avaliação realizada, dentro de um ambiente muito semelhante ao real, a retomada de execução normal do recurso sofre um atraso que pode ser considerado como aceitável. Dessa forma, em termos de tempo para retorno ao processamento normal, o modelo de *checkpoint* com prevalência de objetos teve um resultado satisfatório.

7.1 Revisão do Trabalho Desenvolvido

A dissertação de mestrado apresentada aqui foi divida em seis capítulos que, juntos, apresentaram desde as definições conceituais necessárias para o entendimento do modelo de *checkpoint* proposto até o detalhamento de projeto e implementação do modelo propriamente dito. Inicialmente, no capítulo 1 foi feita a introdução do trabalho, onde foram apresentadas as visões gerais sobre os conceitos que envolvem o modelo de *checkpoint* proposto. Nesse capítulo, também foram apresentados os objetivos principais da pesquisa e as contribuições obtidas a partir dos estudos realizados.

O capítulo 2 apresentou uma visão geral sobre os diversos termos que envolvem o conceito de computação distribuída, dando ênfase especial ao modelo de computação voluntária, que é onde o trabalho proposto por essa dissertação se aplica. Nesse detalhamento, foram apresentados os principais conceitos relacionados com computação em *cluster*, *peer-to-peer* e computação em grade.

Na seqüencia, o capítulo 3 apresentou as principais implementações de mecanismos para *checkpoint* e *restart*. Para isso, o estudo apresentou as implementações disponíveis nas plataformas de computação voluntária, implementações em nível de usuário e sistema operacional, além de abordagens de persistência ortogonal. Esse estudo foi especialmente importante para que fosse possível identificar as abordagens existentes, possibilitando projetar o mecanismo proposto e identificar aspectos que o diferenciam das soluções existentes.

O capítulo 4 apresentou um estudo sobre o conceito de *checkpoint* em memória, com os principais modelos e arquiteturas. Esse conceito de *diskless checkpoint* está diretamente relacionado com o modelo proposto, pois este faz uso da memória como mecanismo principal para persistência das informações intermediárias produzidas.

Finalmente, os capítulos 5 e 6 apresentaram o projeto, implementação e validação do modelo proposto. No capítulo 5, o modelo foi apresentado e todos os componentes da solução foram detalhados. Juntamente com esse detalhamento, a proposta para mecanismo de *checkpoint* foi projetada para ser adaptada à plataforma de computação voluntária XtremWeb, onde as modificações necessárias para o *worker* dessa plataforma foram mostradas. O capítulo 6, na seqüencia, utilizou o modelo implementado para validar a sua utilização dentro de um ambiente de execução que simula uma impressora multifuncional. Nessa capítulo, foram apresentadas de maneira detalhada as modificações e adaptações que foram necessárias para que a aplicação fosse corretamente executada dentro do modelo *checkpoint* proposto. Nesse mesmo capítulo, os resultados referentes à utilização de memória, espaço em disco e outras avaliações, obtidas a partir da execução de uma aplicação real, foram apresentados e analisados.

7.2 Contribuições

O objetivo deste trabalho foi apresentar uma proposta de mecanismo de *checkpoint* para plataformas de computação voluntária, através da prevalência de objetos. Essa proposta é direcionada para recursos com capacidades limitadas de processamento e memória, que apresentam períodos curtos, porém freqüentes, de disponibilidade para projetos voluntários. Nesse contexto, a primeira contribuição foi a de possibilitar que esse tipo de recurso, tipicamente descartado por ambientes de computação voluntária, possam ser utilizados e passem a produzir resultados válidos e úteis, agregando valor ao processamento total do ambiente voluntário.

Outra contribuição foi a de apresentar um mecanismo de *checkpoint* que inova ao utilizar o conceito de prevalência de objetos para a persistência dos resultados intermediários. Conforme apresentado ao longo dessa dissertação, o modelo de *checkpoint* proposto utiliza o conceito de prevalência de objetos em memória, combinado com o processo periódico de *checkpoint* em disco para armazenamento dos resultados produzidos ao longo do processamento voluntário.

Além disso, é interessante incluir como contribuição, a utilização de conceitos relacionados a *diskless checkpoint* combinados com a *disk checkpoint* para armazenamento dos resultados. Essa abordagem híbrida se mostra como um fator diferencial no modelo de *checkpoint* pois combina as vantagens do *checkpoint* tradicional em disco com a rapidez das operações realizadas em memória.

Como objeto final do estudo, o projeto para adaptação do *worker* do XtremWeb permite que seja possível utilizar um mecanismo de *checkpoint* nessa plataforma que, por padrão, não possui esse tipo de mecanismo. Com essa contribuição, a plataforma XtremWeb poderia usufruir dos benefícios de armazenamento dos resultados intermediários produzidos de maneira transparente e eficiente.

Para que as medições de disponibilidade de processamento voluntário das impressoras multifuncionais fossem possíveis, foi desenvolvido um aplicativo que realiza verificações periódicas para obter o *status* do recurso em questão. A implementação dessa ferramenta que realizou as medições de disponibilidade, também pode ser considerada como uma contribuição prática obtida a partir da pesquisa realizada através dessa dissertação.

Ao longo do desenvolvimento dessa pesquisa alguns artigos e resumos foram escritos e submetidos a eventos regionais, nacionais e internacionais. Nessas publicações, foram relacionadas algumas contribuições e resultados obtidos durante a realização do trabalho. Para eventos internacionais, um artigo foi submetido para o *XV Congreso Argentino de Ciencias de la Computacion* (Jujuy, Argentina). Em eventos nacionais, um artigo foi submetido e publicado nos anais do III EPAC - Encontro Paranaense de Computação (Paraná, Brasil). Além disso, um resumo foi submetido ao ERAD 2010.

7.3 Trabalhos Futuros

Os trabalhos futuros prevêem a continuidade do desenvolvimento e utilização do modelo de *checkpoint* proposto, no sentido de habilitar a utilização de recursos com capacidades limitadas de *hardware* em ambientes de computação voluntária. O desenvolvimento desse mecanismo de *checkpoint* pode ser considerado como o primeiro passo de um conjunto de atividades que podem viabilizar a participação desse tipo de recurso.

Dessa forma, abrem-se possibilidades para diversos trabalhos futuros, tanto dentro do contexto do mecanismo de *checkpoint* proposto aqui, quanto da avaliação e utilização

do mesmo dentro de plataformas voluntárias. A primeira idéia para trabalho futuro está na implantação do modelo de *checkpoint* proposto na plataforma XtremWeb, de acordo com o projeto realizado aqui nessa dissertação. Trata-se de uma atividade prática, mas que oferece vantagens interessantes não só ao modelo proposto aqui, mas principalmente à plataforma voluntária, que passa a ter uma opção para armazenamento dos resultados intermediários produzidos.

Um aspecto que também poderia ser observado é a quantidade de *checkpoints* realizados pelo recurso. Com o projeto atual, não existem limites de utilização para o modelo, isto é, não existe um número máximo de operações de *checkpoint* que podem ser realizadas. Entretanto, é de interesse do projeto voluntário como um todo realizar o processamento das atividades no menor tempo possível. Com base nessa premissa, torna-se interessante discutir até que ponto é vantagem permanecer realizando o processamento em curtos intervalos de tempo, armazenando a informação produzida ou repassar a atividade para outro recurso do ambiente voluntário. A partir dessa análise, pode ser feita a utilização de um mecanismo para escalonar as atividades entre os nodos, como proposto por outros trabalhos dentro do nosso grupo de pesquisa.

Dentro da solução e projeto atual, identificam-se alguns ajustes que podem ser úteis e certamente ofereceriam melhorias ao projeto. O primeiro aspecto seria o de descartar o armazenamento de *snapshots* antigos. No modelo atual, todos os *snapshots* realizados são mantidos em disco, ocasionando uma utilização crescente e desnecessária, sob o ponto de vista de recuperação posterior das informações produzidas. Nesse sentido, descartar *snapshots* antigos trariam o benefício de utilizar menos espaço em disco do recurso, já que este possui configurações limitadas.

Nessa mesma linha, é interessante ampliar o escopo de aplicações que podem ser transformadas e adaptadas para o modelo de *checkpoint* apresentado aqui. O primeiro passo para que essa ampliação seja possível, é permitir que o modelo de *checkpoint* seja capaz de lidar com aplicações que fazem uso de parâmetros para sua invocação. O projeto atual já está preparado para tal suporte, restando apenas alguns pequenos ajustes necessários.

Como visto ao longo da dissertação, especialmente no capítulo que trata diretamente sobre o modelo proposto, para que a utilização do mecanismo de *checkpoint* seja possível, algumas alterações são necessárias na aplicação-alvo. Nesse contexto, é imprescindível que todas as regras definidas sejam observadas, caso contrário, o modelo de *checkpoint* não terá êxito na tarefa de armazenar os resultados intermediários produzidos. Dessa forma, uma sugestão para trabalho futuro é o projeto e implementação de um pré-compilador que teria como principal funcionalidade verificar se as regras em questão estão presentes na aplicação adaptada. Com a existência desse compilador específico, seria possível garantir, com segurança, que a aplicação em questão será corretamente utilizada dentro do contexto de prevalência, permitindo a realização de *checkpoints*.

REFERÊNCIAS

AGBARIA, A.; FRIEDMAN, R. Starfish: fault-tolerant dynamic mpi programs on clusters of workstations. In: IEEE INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, 8., Washington, DC, USA. Proceedings... IEEE Computer Society, 1999. p.31.

AHMAD, I. Cluster Computing: a glance at recent events. IEEE Concurrency, Los Alamitos, CA, USA, v.8, n.1, p.67–69, 2000.

AMIRI, K. et al. Dynamic Function Placement in Active Storage Clusters. [S.l.]: Carnegie Mellon University, 1999.

ANDERSON, D. P. Boinc: a system for public-resource computing and storage. In: IEEE-ACM INTERNATIONAL WORKSHOP ON GRID COMPUTING, 5. Anais... [S.l.: s.n.], 2004. p.4–10.

ANDERSON, D. P. et al. SETI@home: an experiment in public-resource computing. Commun. ACM, [S.l.], v.45, n.11, p.56–61, November 2002.

ANDERSON, D. P. et al. Designing a runtime system for volunteer computing. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2006., New York, NY, USA. Proceedings... ACM, 2006. p.126.

ASHOK, N. W. Grid in action: harvesting and reusing idle compute cycles. 2007.

ATKINSON, M. et al. An Orthogonally Persistent JavaTM. ACM SIGMOD Record, [S.l.], v.25, p.1–10, 1996.

ATKINSON, M.; MORRISON, R. Orthogonally persistent object systems. The VLDB Journal, Secaucus, NJ, USA, v.4, n.3, p.319–402, 1995.

ATKINSON, M. P. et al. An approach to persistent programming. Readings in object-oriented database systems, San Francisco, CA, USA, p.141–146, 1990.

BAKER, M. et al. Non-volatile memory for fast, reliable file systems. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), [S.l.], 1992.

BAKER, M. et al. Cluster Computing: a high-performance contender. Computer, [S.l.], v.32, p.79–80, 1999.

BARAK, A. et al. Scalable Cluster Computing with MOSIX for LINUX. In: IN PROCEEDINGS OF LINUX EXPO '99. Anais... [S.l.: s.n.], 1999. p.95–100.

BARKAI, D. An Introduction to Peer-to-Peer Computing. Microcomputer Research Lab Intel Corporation.

BOWMANM, I. Diskless Checkpointing Presentation. 2006.

BROWN, A. L. Persistent Object Stores. [S.l.]: Software Engineering Journal, 1988.

BRYANT, R.; HARTNER, B. Java, Threads, and Scheduling in Linux. 2009.

CAPPELLO, F. et al. Computing on Large Scale Distributed Systems: xtremweb architecture, programming models, security, tests and convergence with grid. FGCS Future Generation Computer Science, [S.l.], v.21, n.3, p.417–437, March 2005.

CAROTHERS, C.; SZYMANSKI, B. Checkpointing Multithreaded Programs. Dr. Dobbs Journal, [S.l.], 2002.

CHANDRASEKARAN, S.; KEHOE, B. Technical Comparison of Oracle9i Real Application Clusters vs. IBM DB2 UDB EEE v8.1. [S.l.]: Oracle White Paper, 2002.

CHEN, P. M. et al. RAID: high-performance, reliable secondary storage. ACM Computing Surveys, [S.l.], 1994.

CHEN, P. M. et al. The Rio file cache: surviving operating system crashes. In: ASPLOS-VII: PROCEEDINGS OF THE SEVENTH INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, New York, NY, USA. Anais... ACM, 1996. p.74–83.

CHIUEH, T.; DENG, P. Efficient checkpoint mechanisms for massively parallel machines. International Symposium on Fault-Tolerant Computing (FTCS), [S.l.], 1996.

CHIUEH, T.; DENG, P. Evaluation of Checkpoint mechanisms for massively parallel machines. Proceeding of 26th Fault-Tolerance Computer Symposium, [S.l.], p.370–379, 1996.

CIRNE, W. et al. Running Bag-of-Tasks Applications on Computational Grids: the mygrid approach. In: ICPP. Anais... [S.l.: s.n.], 2003. p.407–.

CORPORATION, V. Db4o: native java e .net open source object database. 2009.

CROGRID. CRO-GRID Infrastructure: questions and answers. Disponivel em http://www.srce.hr/.

CRONK, D. et al. Metacomputing: an evaluation of emerging systems. 2000.

DEARLE, A. et al. Orthogonal Persistence Revisited. 2009.

DEVICES, U. Gridmp: hpc systems management for the complete distributed computing lifecycle. Disponível em http://www.univaud.com.

DIETER, W. R. et al. User-Level Checkpointing for Linux Threads Programs. In: FREENIX TRACK. 2001 USENIX ANNUAL TECHNICAL CONFERENCE, Berkeley, CA, USA. Proceedings... USENIX Association, 2001. p.81–92.

DOHERTY, M. Database Cluster for e-Science. UK e-Science All Hands Meeting, [S.l.], 2003.

DONGARRA, J. J. et al. Disaster Survival Guide in Petascale Computing: an algorithmic approach. 2008.

ECONOMIST. A grid by any other name. Disponivel em http://public.eu-egee.org/files/EconomistDec04.pdf.

ELNOZAHY, E. et al. The performance of consistent checkpoint. 11th Symposium on Reliable Distributed System, [S.l.], p.39–47, 1992.

FEDAK, G. et al. XtremWeb: a generic global computing system. In Proceedings of the IEEE International Symposium on Cluster Computing and the Grid (CCGRID'01), [S.l.], v.582-587, 2001.

FELDMAN, S. I.; BROWN, C. B. IGOR: a system for program debugging via reversible execution. ACM SIGPLAN, Workshop on Parallel and Distributed Debugging, New York, NY, USA, v.24, n.1, p.112–123, 1989.

FIELDS, S. Hunting for wasted computing power: new software for computing networks puts idle pcs to work. 2008.

FOSTER, I.; KESSELMAN, C. Globus: a metacomputing infrastructure toolkit. International Journal of Supercomputer Applications, [S.l.], v.11, p.115–128, 1996.

FOSTER, I.; KESSELMAN, C. Globus: a metacomputing intrastructure toolkit. International Journal of Supercomputer Applications, [S.l.], v.22, p.115–128, 1997.

FOSTER, I. T. What is the Grid? A Three Point Checklist. GRIDtoday, [S.l.], 2002.

FOSTER, I. T. et al. The Anatomy of the Grid: enabling scalable virtual organization. International J. Supercomputer Applications, [S.l.], v.15, num 3, 2001.

FOSTER, M. Pursuing the AP's to Checkpointing with UCLiK. In: INTERNATIONAL LINUX SYSTEM TECHNOLOGY CONFERENCE, 10., Washington, DC, USA. Proceedings... IEEE Computer Society, 2003. p.147.

GALLAGER, R. G. Information Theory and Reliable Communication. [S.l.]: John Wiley and Sons, 1968.

GARCIA-MOLINA, H.; SALEM, K. Main memory database systems: an overview. IEEE Transactions on Knowledge and Data Engineering, [S.l.], v.4, 1992.

GIBSON, D. Checkpoint/restart for Solaris and Linux. 2009.

GIBSON, G. A. Redundant Disk Arrays: reliable, parallel secondary storage. MIT Press, [S.l.], 1992.

GRIMSHAW, A. S. et al. The Legion vision of a worldwide virtual computer. Commun. ACM, New York, NY, USA, v.40, n.1, p.39–45, 1997.

HARGROVE, P. H.; DUELL, J. C. Berkeley lab checkpoint/restart (blcr) for linux clusters. In: PROCEEDINGS OF SCIDAC. 2006. Anais... Online]. Available: http://stacks.iop.org/17426596/46/494, 2006. p.2006.

HENDRIKS, E. VMADump. 2009.

HEWLETT-PACKARD. Hewlett-Packard - Computers, Laptops, Servers and Printers. 2009.

HIGAKI, H. et al. Checkpoint and Rollback in Asynchronous Distributed Systems. In: INFOCOM 97: PROCEEDINGS OF THE INFOCOM 97. SIXTEENTH ANNUAL JOINT CONFERENCE OF THE IEEE COMPUTER AND COMMUNICATIONS SOCIETIES. DRIVING THE INFORMATION REVOLUTION, Washington, DC, USA. Anais... IEEE Computer Society, 1997. p.998.

HUNG, E. Fault Tolerance and Checkpointing Schemes for Cluster of Workstations. ELEC6062 Scalable Parallel Computing, [S.l.], 1998.

INC, X. Splash G640 Color Server for Xerox DocuColor 12 Series. 2009.

INRIA. XtremWeb: the open source platform for desktop grids. Disponivel em http://www.xtremweb.net/.

JAMES S. PLANK YOUNGBAE KIM, J. D. Fault-tolerant matrix operations for networks of workstations using diskless checkpointing. Parallel Distributed Computing, [S.l.], v.43, p.125–138, 1997.

JATIT. Journal of Theoretical and Applied Information Technology. Disponivel em http://www.jatit.org/.

JOSEPH, A. D. et al. Mobile Computing with the Rover Toolkit. IEEE Transactions on Computers, Los Alamitos, CA, USA, v.46, n.3, p.337–352, 1997.

KASBEKAR, M. et al. Selective Checkpointing and Rollbacks in Multithreaded Object-oriented Environment. In: IEEE TRANSACTIONS ON RELIABILITY. Anais... [S.l.: s.n.], 1999. p.325–337.

KAWAGUCHI, A. et al. A flash-memory based file system. In: TCON95: PROCEEDINGS OF THE USENIX 1995 TECHNICAL CONFERENCE PROCEEDINGS ON USENIX 1995 TECHNICAL CONFERENCE PROCEEDINGS, Berkeley, CA, USA. Anais... USENIX Association, 1995. p.13–13.

KESSELMAN, C. et al. A resource management architecture for metacomputing systems. Proceedings of 12th International Parallel Processing Symposium e 9th Symposium on Parallel and Distributed Processing Workshop on Job Scheduling Strategies for Parallel Processing, [S.l.], 1998.

KULKARNI, D. et al. LINQ to SQL: .net language-integrated query for relational data. 2009.

LAADAN, O. et al. Transparent Checkpoint-Restart of Distributed Applications on Commodity Clusters. Proceedings of IEEE International Conference on Cluster Computing, [S.l.], 2005.

LEXMARK. Lexmark Products. 2009.

LIO, W. Trustworthy service selection and composition reducing the entropy of service-oriented web. 2005.

LU, C.-D. Scalable Diskless Checkpointing for Large Parallel Systems. 2005. Dissertação (Mestrado em Ciência da Computação) — University of Illinois at Urbana-Champaign.

LéON, J. et al. Fail-Safe PVM: a portable package for distributed programming with transparent recovery. [S.l.]: Carnegie Mellon University, 1993.

MEYER, N. User and Kernel Level Checkpointing. Proceedings of the Sun Microsystems HPC Consortium Meeting, [S.l.], p.15–23, 2003.

MICROSYSTEMS, S. Java Persistence API. 2009.

MILOJICIC, D. S. et al. Peer-to-Peer Computing. HP Laboratories Palo Alto.

MOSS, J. E. B.; HOSKING, A. L. Approaches to Adding Persistence to Java. 1996.

NERI, V. et al. Xtremweb: a generic global computing system. 1st IEEE International Symposium on Cluster Computing and the Grid, [S.l.], p.582–587, 2001.

NETBEANS. NetBeans Profiler. 2009.

O'CONNOR, M. Process Migration on Chorus. 2009.

ORGANICK, E. The MULTICS system: an examination of its structure. [S.l.]: MIT Press, 1972.

OSMAN, S. et al. The Design and Implementation of Zap: a system for migrating computing environments. In: IN PROCEEDINGS OF THE FIFTH SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION (OSDI 2002. Anais... [S.l.: s.n.], 2002. p.361–376.

OVEREINDER, B. et al. A dynamic load balancing system for parallel cluster computing. Future Gener. Comput. Syst., Amsterdam, The Netherlands, The Netherlands, v.12, n.1, p.101–115, 1996.

PAN, D.; LINTON, M. Supporting Reverse Execution of Parallel Programs. ACM Sigplan Notices, Workshop on Parallel and Distributed Debugging, [S.l.], v.24, p.124–129, 1989.

P.ANDERSON, D.; FEDAK, G. The Computational and Storage Potential of Volunteer Computing. In: CCGRID 06: PROCEEDINGS OF THE SIXTH IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, Washington, DC, USA. Anais... IEEE Computer Society, 2006. p.73–80.

PARRINGTON, G. D. et al. The Design and Implementation of Arjuna. 1995.

PATTERSON, D. et al. A case for Redundant Array of Inexpensive Disks (RAID). ACM SIGMOD Conference of Management of Data, [S.l.], p.109–116, 1988.

PENG, L.; KIAN, L. N1GE6 Checkpointing and Berkeley Lab Checkpoint/Restart. 2009.

PINHEIRO, E. Truly-Transparent Checkpointing of Parallel Applications. 1998.

PLANK, J. et al. Diskless Checkpointing. IEEE Transactions Parallel and Distributed System, [S.l.], v.9, p.972–986, 1998.

PLANK, J.; LI, K. Faster checkpointing with N+1 parity. International Symposium on Fault-Tolerant Computing (FTCS), [S.l.], 1994.

PLANK, J. S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. Software – Practice and Experience, [S.l.], v.27, p.995–1012, 1997.

PLANK, J. S. et al. Libckpt:transparent checkpointing under unix. Usenix Winter Technical Conference, [S.l.], p.213–223, 1995.

PLANK, J. S. et al. Diskless Checkpointing. [S.l.]: University of Tenesse, 1998.

PLANK, J. S.; LI, K. Faster checkpointing with n+1 parity. FTCS, [S.l.], v.288–297, 1994.

PRESS, H. Folding@home and genome@home: using distributed computing to tackle previously intractable problems in computational biology. 2003.

PROCESS, J. C. Java Data Objects (JDO) Specification. 2009.

PRUITT, P. N. An Asynchronous Checkpoint and Rollback Facility for Distributed Computations. 2002. Tese (Doutorado em Ciência da Computação) — College of William and Mary.

RECHERCHE NUCLéAIRE, O. E. pour la. Grid Cafe @ CERN. 2009.

REED, I.; SOLOMON, G. Polynomial Codes Over Certain Finite Fields. SIAM Journal of Applied Math, [S.l.], v.8, p.300–304, 1960.

RFC. The RFC Archive. 2009.

ROMAN, E. A Survey of Checkpoint/Restart Implementations. [S.l.]: Berkeley Lab, 2002.

SANCHO, J. C. et al. Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. In: IPDPS '05: PROCEEDINGS OF THE 19TH IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS'05) - WORKSHOP 18, Washington, DC, USA. Anais... IEEE Computer Society, 2005. p.300.2.

SARMENTA, L. F. G. Volunteer Computing. 2001. Tese (Doutorado em Ciência da Computação) — Department of Electrical Engineering and Computer Science. Massachusetts Institute of Technology.

SHRIVASTAVA, S. K. et al. An Overview of the Arjuna Distributed Programming System. IEEE Softw., Los Alamitos, CA, USA, v.8, n.1, p.66–73, 1991.

SILVA, L. M.; SILVA, J. An experimental study about diskless checkpointing. 24th Euromicro Conference Proceeding., [S.l.], v.1, p.395–402, 1998.

SILVA, L. M.; SILVA, J. G. The Performance of Coordinated and Independent Checkpointing. In: IPPS 99/SPDP 99: PROCEEDINGS OF THE 13TH INTERNATIONAL SYMPOSIUM ON PARALLEL PROCESSING AND THE 10TH SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, Washington, DC, USA. Anais... IEEE Computer Society, 1999. p.280–284.

SILVA, L.; SILVA, J. G. An Experimental Study about Diskless Checkpointing. In: EUROMICRO 98: PROCEEDINGS OF THE 24TH CONFERENCE ON EUROMICRO, Washington, DC, USA. Anais... IEEE Computer Society, 1998. p.10395.

SILVA, M. M. et al. Checkpointing SPMD applications on Transputer networks. Scalable High-Performance Computing Conference (SHPCC), [S.l.], 1994.

SKLAR, B. Digital Communications: fundamentals and applications. [S.l.]: Prentice-Hall, 1988.

STAFF, B. BOINC Preferences. 2008.

STEEN, A. van der. Supercomputer and Cluster Computing. [S.l.]: Faculty of Science Physics and Astronomy, Utrecht University, 2007.

STELLNER, G. CoCheck: checkpointing and process migration for mpi. In: IPPS 96: PROCEEDINGS OF THE 10TH INTERNATIONAL PARALLEL PROCESSING SYMPOSIUM, Washington, DC, USA. Anais... IEEE Computer Society, 1996. p.526–531.

STERLING, T. et al. BEOWULF: a parallel workstation for scientific computation. Proceedings of the 24th International Conference on Parallel Processing, [S.l.], p.11–14, 1995.

STONE, N. et al. A Checkpoint and Recovery System for the Pittsburgh Supercomputing Center Terascale Computing System. Pittsburgh Supercomputing Center, Pittsburgh.

SUDAKOV, O. et al. CHPOX: transparent checkpointing system for linux clusters. In: IPDPS '05: PROCEEDINGS OF THE 14TH IEEE WORKSHOP ON INTELLIGENT DATA ACQUISITION AND ADVANCED COMPUTING SYSTEMS: TECHNOLOGY AND APPLICATIONS (IDAACS'07), Dortmund. Anais... IEEE Computer Society, 2007. p.159–164.

SUNDERAM, V. S. et al. Numerically stable real-number codes based on random matrices. Proceedings of the Computational Science - ICCS 2005, 5th International Conference, [S.l.], p.115–122, 2004.

TAKAHASHI, T. et al. PM2: a high performance communication middleware for heterogeneous network environments. In: SUPERCOMPUTING 00: PROCEEDINGS OF THE 2000 ACM/IEEE CONFERENCE ON SUPERCOMPUTING (CDROM), Washington, DC, USA. Anais... IEEE Computer Society, 2000. p.16.

TANENBAUM, A. S. Modern Operating Systems. [S.l.]: Prentice Hall Press, 2007.

TEAM, C. Condor Version 7.2.4 Manual. 2009.

TELECOM, D. SNMP Tutorial. 2009.

THAIN, D. et al. Distributed computing in practice: the condor experience: research articles. Concurr. Comput. : Pract. Exper., Chichester, UK, v.17, n.2-4, p.323–356, 2005.

TOTH, D. M. Improving the Productivity of Volunteer Computing. Worcester Polytechnic Institute, [S.l.], 2004.

VILLELA, C. An introduction to object prevalence. 2002.

WANG, A.-I. et al. Conquest: better performance through a disk/persistent-ram hybrid file system. In: ATEC 02: PROCEEDINGS OF THE GENERAL TRACK OF THE ANNUAL CONFERENCE ON USENIX ANNUAL TECHNICAL CONFERENCE, Berkeley, CA, USA. Anais... USENIX Association, 2002. p.15–28.

WANG, Y.-M. et al. Checkpointing and Its Applications. In: FTCS 95: PROCEEDINGS OF THE TWENTY-FIFTH INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, Washington, DC, USA. Anais... IEEE Computer Society, 1995. p.22.

WILSON, P.; MOHER, T. Demonic memory for process histories. SIGPLAN, Conference on Programming Language Design and Implementation, New York, NY, USA, v.24, n.7, p.330–343, 1989.

WU, M.; ZWAENEPOEL, W. eNVy: a non-volatile, main memory storage system. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), New York, NY, USA, v.28, n.5, p.86–97, 1994.

WUESTEFELD, K. Prevayler, a prevalence layer for Java. http://www.prevayler.org.

YAMAMOTO, H. et al. Multi-Class Composite N-gram language model for spoken language processing using multiple word clusters. In: ACL 01: PROCEEDINGS OF THE 39TH ANNUAL MEETING ON ASSOCIATION FOR COMPUTATIONAL LINGUISTICS, Morristown, NJ, USA. Anais... Association for Computational Linguistics, 2001. p.531–538.

ZHENG, G. et al. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. IEEE International Conference on Cluster Computing, [S.l.], 2004.

ZHONG, H.; NIEH, J. CRAK: linux checkpoint / restart as a kernel module. [S.l.]: Department of Computer Science. Columbia University, 2002.