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

Recuperação com Base em *Checkpointing*: uma Abordagem Orientada a Objetos

por

FRANCISCO ASSIS DA SILVA

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

Prof[a]. Dr[a]. Ingrid Eleonora Schreiber Jansch-Pôrto
Orientadora

Prof[a]. Dr[a]. Maria Lúcia Blanck Lisbôa
Co-Orientadora

Porto Alegre, novembro de 2002.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Silva, Francisco Assis da

Recuperação com base em *Checkpointing*: uma abordagem Orientada a Objetos / por Francisco Assis da Silva – Porto Alegre: PPGC da UFRGS, 2002.

157p.:il.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, 2002. Orientadora: Jansch-Pôrto, Ingrid E. S.; Co-orientadora: Lisboa, Maria Lúcia B.

1. Tolerância a falhas. 2. Recuperação de objetos. 3. Biblioteca de *checkpointing*. 4. Orientação a Objetos. 5. Java. I. Jansch-Pôrto, Ingrid E. S. II. Lisboa, Maria Lúcia B. III. Título.

Agradecimentos

Às amigas prudentinas da turma de Rolândia: Cláudia, Daniela e Liliane, pela amizade, incentivo e companheirismo durante os desafios vencidos. Em especial, agradeço às professoras Dra. Ingrid E. S. Jansch Pôrto e Dra. Maria Lúcia Blanck Lisbôa pela orientação imprescindível, pelos ensinamentos, pela paciência e confiança; ao Prof. Sergio L. Cechin pelo apoio nos momentos decisivos; à Universidade do Oeste Paulista (UNOESTE) pela oportunidade de realização deste curso de mestrado; e, agradeço a todas as pessoas que direta ou indiretamente colaboraram na execução desta dissertação.

Oferecimento

Ofereço esta dissertação primeiramente a Deus, por ter me concedido a oportunidade e a capacidade de ter realizado este trabalho.

À minha mãe e ao meu pai; que me educaram, me ensinaram a ser humilde, e que sempre me incentivaram a não desistir de alcançar um objetivo na minha vida que me fizesse feliz.

Sumário

Lista de Abreviaturas

API	Application Program Interface
CPU	Central Processing Unit
CORBA	Common Object Request Broker Architecture
RMI	Remote Method Invocation
JVM	Java Virtual Machine
MVJ	Máquina Virtual Java
NUMA	Non-Uniform Memory Access
UML	Unified Modeling Language
RAM	Random Access Memory
RPC	Remote Procedure Call
SMP	Simulating a Symmetric Multiprocessor
OMG	Object Management Group

Lista de Figuras

Lista de Tabelas

Resumo

Independentemente do modelo de programação adotado, no projeto e implementação de aplicações de alta disponibilidade, faz-se necessário usar procedimentos de tolerância a falhas. Dentre as atividades que trazem consigo interesse de pesquisa na área de Tolerância a Falhas, estão os mecanismos de recuperação em um sistema computacional. Do ponto de vista prático, estes mecanismos buscam manter próximo do mínimo o tempo total de execução de aplicações computacionais de longa duração, ao mesmo tempo em que as preparam para não sofrerem perdas significativas de desempenho, em caso de falhas.

Paralelamente à evolução dos sistemas computacionais, foi possível observar também a evolução das linguagens de programação, principalmente as que utilizam o paradigma orientado a objetos. O advento da área de tolerância a falhas na orientação a objetos resultou em novos problemas na atividade de recuperação quanto aos mecanismos de salvamento de estados e retomada da execução, principalmente no que se refere às dificuldades de gerenciamento e controle sobre a alocação de objetos. Entretanto, observa-se que a complexidade de implementação dos mecanismos de recuperação, por parte dos programadores, exige deles conhecimentos mais especializados para o salvamento dos estados da aplicação e para a retomada da execução. Portanto, a simplificação do trabalho do programador, através do uso de uma biblioteca de *checkpointing* que implemente os mecanismos de salvamento de estados e recuperação é o ponto focal deste trabalho.

Diante do contexto exposto, nesta dissertação, são definidas e implementadas as classes de uma biblioteca que provê mecanismos de *checkpointing* e recuperação. Esta biblioteca, denominada de *Libcjp*, visa aprimorar o processo de recuperação de aplicações orientadas a objetos escritas na linguagem de programação Java. Esta linguagem foi escolhida para implementação devido à presença dos recursos de persistência e serialização. Para a concepção do trabalho, são considerados ambos os cenários no paradigma orientado a objetos: objetos centralizados e distribuídos. São utilizados os recursos da API de serialização Java e a tecnologia Java RMI para objetos distribuídos. Conclui-se o trabalho com a ilustração de casos de uso através de diversos exemplos desenvolvidos a partir de seus algoritmos originais inicialmente, e incrementados posteriormente com os mecanismos de *checkpointing* e recuperação. Os componentes desenvolvidos foram testados quanto ao cumprimento dos seus requisitos funcionais. Adicionalmente, foi realizada uma análise preliminar sobre a influência das ações de *checkpointing* nas características de desempenho das aplicações.

Palavras-chave: tolerância a falhas, recuperação de objetos, biblioteca de *checkpointing*, orientação a objetos, Java.

TITLE: "CHECKPOINTING-BASED RECOVERY: AN OBJECT-ORIENTED APPROACH"

Abstract

When designing or implementing high availability applications, regardless of the programming model that is chosen, there is an inherent need for fault tolerance. Among these procedures, recovery mechanisms are of increasing interest to fault tolerant computing research due to their practical application. The basic task of these mechanisms is to keep the total execution time of time-consuming applications in normal (fault-free) operation close to minimum, while preparing the applications to be resilient with respect to faults, with minimum performance degradation.

As computing systems have evolved over the years, so have programming languages, especially those based on the object-oriented paradigm. The development of fault tolerance approaches in the object-oriented domain has resulted in new problems, such as those related to checkpointing and execution recovery/restart mechanisms. The main difficulties are related to object allocation management. An additional problem is that programmers generally lack the specialized knowledge that is needed to implement recovery mechanisms. The work presented here focuses on simplifying the programmer's task in that respect, through the use of a pre-defined checkpointing library that implements mechanisms for object persistence and application recovery.

In that context, we define and implement classes that provide checkpointing and recovery mechanisms, organized in a library called Libcjp. Our goal is to improve the recovery process of object-oriented applications written in the Java programming language. We chose Java because of the persistence and serialization facilities it provides. We have explored both centralized and distributed scenarios, using the Java serialization API and the Java Remote Method Invocation (RMI) technology to handle distributed objects. We conclude by showing practical examples, based on the implementation of classic mathematical algorithms. The original algorithms were initially implemented, and subsequently enhanced with the addition of checkpointing and recovery mechanisms through Libcjp method calls. Finally, the library components were tested against their functional requirements. In addition, some preliminary experiments were conducted to observe the influence of the checkpointing overhead on the performance of typical applications.

Keywords: fault tolerance, object recovery, checkpointing library, object-oriented programming, Java.

1 Introdução

Na área de Tolerância a Falhas, a necessidade de minimizar o tempo total de execução de aplicações computacionais de longa duração, na presença de falhas do sistema, e de aperfeiçoar serviços de disponibilidade, traz consigo um interesse intenso na pesquisa das atividades de recuperação de processos em um sistema computacional. A técnica tradicional de recuperação de processos em um sistema computacional baseia-se na restauração de um processo ou conjunto de processos para um estado operacional normal (estado livre de erros). A recuperação pode ser tão simples como reiniciar um computador (sistema computacional pessoal de baixo custo) com falha ou reiniciar um processo falho. Entretanto, a implementação genérica da recuperação para processos concorrentes com diferentes atribuições e funcionalidades é mais complexa. No grupo de Tolerância a Falhas da UFRGS, este assunto foi objeto de estudo detalhado na tese de Sérgio Cechin [CEC2002].

Os estados salvos durante a execução de um **processo**, para os quais o processo pode mais tarde ser restaurado, são conhecidos como **pontos de recuperação** ou ***checkpoints***[1] [SIN94]. O termo ***checkpointing***[2] é usado para a ação de estabelecer o ponto de recuperação. O estado de um programa em execução é periodicamente salvo para um meio estável (armazenamento com segurança) do qual pode ser recuperado depois da ocorrência de uma falha [PLA95].

Existem algumas bibliotecas e ferramentas com este objetivo: o salvamento dos estados da execução de um processo e a retomada da execução, mediante a ocorrência de uma falha do sistema. Pode-se mencionar a biblioteca *libckpt* [PLA95] (ferramenta para *checkpointing* em um único processador desenvolvida em nível de usuário), a ferramenta *Epckpt* [PIN98] (implementada no nível do *kernel* do *Linux*, pode estabelecer *checkpoint* de aplicações paralelas) e o sistema *Condor* [LIT97] (é um sistema para processamento distribuído para *Unix*, desenvolvido em nível do usuário) como sendo implementações desta técnica de *checkpointing* e recuperação ao nível de processos. Estas bibliotecas e ferramenta foram investigadas neste trabalho e são descritas na seção 1.2 (trabalhos correlatos).

O estabelecimento de pontos de recuperação introduz sobrecarga de processamento (*overhead*) proporcional ao tamanho do *checkpoint* [ELN92]. Isto significa que o *checkpointing* deve ser adequadamente planejado para que não onere demasiadamente a atividade normal do sistema. Quando os *checkpoints* contêm um conjunto muito grande de dados, a observação bem feita sobre as variações neste conjunto de dados (se há variações pontuais, por exemplo) pode levar ao uso de mecanismos otimizados na seleção de dados para o estabelecimento dos *checkpoints*. Tradicionalmente, otimizações em procedimentos de *checkpointing* são objetivos de programas científicos escritos nas linguagens de programação imperativas *Fortran* e *C*. Tais programas têm freqüentemente grande quantidade de dados usados somente na modalidade de leitura. Neste ambiente, uma técnica de otimização efetiva é a denominada de *checkpointing* incremental, que usa facilidades de nível de sistema para identificar páginas de memória virtual modificadas [PLA99]. Cada ponto de

[1, 2] Serão usados os termos em inglês, no decorrer do texto, já que suas traduções para o português implicam no uso de um conjunto de palavras. Adicionalmente, os termos em inglês encontram uso corrente no meio técnico.

recuperação, além dos estados dos registradores da CPU, variáveis globais, *heap* e *stack*, contém somente as páginas que foram modificadas desde o salvamento prévio.

A atividade de *checkpointing*, numa abordagem orientada a **objetos**, visa salvar o estado dos objetos do programa em execução em um estado operacional normal (livre de erros) para o armazenamento estável.

Programas escritos em uma linguagem orientada a objetos, tal como Java, introduzem novas demandas de *checkpointing* [LAW2000]:

- o estilo de programação orientado a objetos propicia a criação de muitos objetos pequenos. Cada objeto pode ter alguns campos que são usados apenas através de operações de leitura, e outros que são freqüentemente modificados;
- o programador de Java não tem nenhum controle sobre a alocação de objetos. Assim, é impossível assegurar que os objetos modificados freqüentemente sejam todos armazenados na mesma página de memória. Além disso, uma única página pode conter objetos vivos e objetos que aguardam o coletor de lixo para serem removidos da memória;
- programas Java são executados em uma máquina virtual que suporta processos simultâneos. Assim, a linguagem Java encoraja paralelismo como um método de engenharia de *software*; bibliotecas de programação, como para tratar a interface gráfica do usuário, criam muitos processos cujos estados nem sempre são proveitosos em um *checkpoint*. Também, a alocação de objetos não é usualmente gerenciada da mesma forma empregada com processos numa linguagem imperativa, neste caso adiciona memória desnecessária para o *checkpoint*.

Estes argumentos sugerem que um enfoque em nível de linguagem dirigido pelo usuário pode ser apropriado para programas Java. Mas, *checkpointing* realizado em nível de linguagem aumenta o tamanho do programa-fonte com um código para registrar os estados dos objetos do programa [KAS99, KIL99, RAM97]. Esse código de *checkpointing* deveria ser introduzido sistematicamente, e interferir o mínimo possível no comportamento padrão do programa. Um enfoque para isto é adicionar métodos a cada classe para salvar e restaurar o estado local. Assim, o *checkpointing* do programa é então executado por um método genérico que invoca os métodos de *checkpointing* de cada objeto. *Checkpointing* incremental pode ser implementado associando uma variável de instância (*flag*) para cada objeto, indicando se o objeto foi modificado desde o *checkpoint* prévio [KAS99, KIL99].

Em uma abordagem orientada a objetos, salvar dados de uma aplicação em execução é uma operação requisitada com freqüência. Essa representação de dados feita no armazenamento estável (representação externa) é tipicamente muito diferente da mesma representação de dados de um programa em execução (representação interna). Quando é requerida a reutilização dos dados salvos através do armazenamento estável, eles necessitam ser convertidos da representação externa para a representação interna. Em um programa orientado a objetos, dados são representados em forma de objetos, e produzir objetos persistentes é uma maneira eficiente de salvar os dados da aplicação no armazenamento estável [KAS99].

O mecanismo de serialização de objeto em Java permite capturar o estado de um ou mais objetos e representá-los como uma seqüência de *bytes* em um formato independente de plataforma [SUN98, ARN97]. A partir desta seqüência de *bytes*, os objetos podem depois ser recriados em tempo de execução no mesmo ou em qualquer

outro sistema Java. O processo de capturar o estado de um objeto é denominado de serialização de objeto (*serializing*), e o processo de recriar o objeto a partir do estado capturado é denominado de-serialização do objeto (*deserializing*). Quando um objeto é de-serializado, um novo objeto é criado e seu estado é salvo a partir daquele representado pela seqüência de *bytes*; efetivamente é criada uma cópia do objeto serializado, ao invés de sobrescrever o estado de um objeto existente [HAN99].

A portabilidade é uma das características presentes na linguagem Java. Dessa forma, *checkpointing* implementado empregando mecanismos desta linguagem provê independência de máquina virtual. *Checkpointing* é conceitualmente semelhante à serialização, a conversão de uma estrutura de objeto em uma representação plana (*flat*). Em Java, serialização é implementada usando reflexão em tempo de execução. Reflexão é usada para determinar a estrutura estática de cada objeto (seu tipo, nome do campo, etc.), e para ter acesso aos valores de campos registrados [LAW2000].

Armazenamento estável, **meio estável** ou **memória estável** é um tipo de memória cujo conteúdo deve sobreviver ao mau funcionamento do sistema. Ele é o elemento-chave para estabelecer *checkpoint* e recuperação em sistemas tolerantes a falhas. É implementado normalmente fazendo o uso de discos rígidos, por causa de suas características não voláteis e robustez contra falhas de processador [CUN2001]. Pode-se citar o espelhamento de discos baseado em redundância (conjunto de imagens de um disco em dispositivos separados) como uma maneira de se obter um armazenamento mais confiável. De acordo com Banâtre *et al*. [BAN88], as duas propriedades principais de um armazenamento estável são: (1) a resistência com relação a problemas de *hardware* externas à memória ou falhas de *software* (falhas de processador, falhas de energia, acesso errado na memória) e também resistência contra falhas internas tais como problemas físicos de memória, e (2) a atomicidade de acesso de leitura e escrita.

Pesquisas têm demonstrado a preocupação com a confiabilidade dos meios de armazenamento onde os *checkpoints* são salvos. O meio mais comum para armazenamento estável é o disco. As principais razões para esta escolha estão relacionadas com as características não voláteis do disco, isto é, ele não perde o seu conteúdo quando a energia elétrica é interrompida, e, de fato, como mostrado por experiência, raramente o conteúdo do disco é significantemente afetado pelo mau funcionamento do processador [CUN2001]. Entretanto, acessos a disco são lentos e desse modo contribuem significantemente para a degradação do desempenho. Este problema limita severamente o uso de *checkpointing* em sistemas de controle que apresentem restrições de tempo real. Muitos controladores são até mesmo sem discos, especialmente controladores embutidos, o que impede o armazenamento em disco nestes casos. Uma alternativa para os problemas de velocidade e ausência de disco é armazenar *checkpoints* em memórias e processadores fisicamente independentes. Em vez de armazenar *checkpoints* em seu disco local; um processador envia os *checkpoints* para um processador remoto que os armazena em sua própria memória [PLA98]. Se um processador falhar, outro processador (não falho) que possui seu último estado válido retorna ao *checkpoint* armazenado em memória e substitui o processador falho, ou envia o estado a outro processador. Este enfoque elimina sobrecarga de disco, mas introduz outras fontes de sobrecarga (*overhead*), como: memória, processador e utilização da rede. Segundo Plank e Puening [PLA98], testes mostram que *checkpointing* sem uso do disco local (*diskless checkpointing*) não traz nenhum aumento de sobrecarga de disco, embora tenha uma latência menor (tempo para completar o estabelecimento de um *checkpoint*) e um menor tempo de recuperação.

Além destes, também foram identificadas na bibliografia, algumas formas de implementação de armazenamento estável em memória RAM. Segundo [CUN2001], a primeira implementação conhecida em memória RAM está descrita em Banâtre *et al*. [BAN86]. Consistiu de uma placa de memória RAM especialmente projetada com bateria de *backup*, onde havia um mecanismo de acesso complexo que tentava prevenir acessos descontrolados de processador. Uma versão mais poderosa foi implementada pelo mesmo grupo de pesquisa e pode ser encontrada em Banâtre *et al*. [BAN88]. Foi construído um mecanismo para manipular objetos estruturados com uma proteção mais eficaz contra defeitos do processador. Ambas as soluções eram bastante rápidas, mas bastante complexas, principalmente a segunda. Eles nunca foram usados amplamente fora do grupo de pesquisa onde foram desenvolvidos, por causa da configuração particular de *hardware* requerida [CUN2001]. Uma outra tentativa de usar memória RAM para armazenamento estável foi feita por Grygier e Dal Cin [GRY94]. É apresentada uma solução baseada em memória de comunicação compartilhada, usada para comunicação entre processos em MENSY, um multiprocessador experimental NUMA (*Non-Uniform Memory Access*). A desvantagem principal deste enfoque é a dependência forte desta arquitetura particular.

Vale ressaltar que, ao longo do texto desta dissertação, serão mencionadas utilizações de disco para salvar os estados dos objetos da aplicação no arquivo de *checkpoint*. Nesta dissertação, não são considerados os fatores de segurança e confiabilidade do meio de armazenamento; por esta razão, é utilizado, na prática, disco magnético convencional em configuração não redundante, local, ao invés de uma implementação de armazenamento estável.

As subseções que seguem se preocupam em relatar os objetivos e a metodologia que motivaram a escolha deste trabalho e o enfoque adotado nesta dissertação. Na seqüência, são apresentados os trabalhos investigados que auxiliaram no entendimento e no conhecimento dos mecanismos de *checkpointing* e recuperação tanto na estrutura de processos quanto na abordagem orientada a objetos. Por fim, é apresentado o delineamento de conteúdo dos demais capítulos.

1.1 Objetivos e Metodologia do Trabalho

Este trabalho considera um modelo de sistema composto por máquinas independentes que processam aplicações sem duplicação ou redundância múltipla, mas que podem ser atingidas por falhas/defeitos. Diante desta possibilidade de ocorrência de defeitos, deseja-se dispor de mecanismos para suporte à recuperação, de forma a minimizar as perdas decorrentes de falhas no sistema, correspondentes aos tempos de reprocessamento, além de minorar o impacto dos defeitos sobre a disponibilidade, mas sem obrigar o programador a encarregar-se de detalhes de implementação dos mecanismos. Adicionalmente, este trabalho vem enriquecer o Grupo de Tolerância a Falhas da UFRGS, servindo de base para experimentos de protocolos desenvolvidos.

Portanto, visando aprimorar o processo de recuperação de aplicações orientadas a objetos escritas em Java, objetivou-se com este trabalho desenvolver uma biblioteca que implemente métodos de *checkpointing* e recuperação, considerando ambos os cenários para execução de aplicações: objetos centralizados e distribuídos. Partiu-se de pesquisas e estudos para obtenção do conhecimento das técnicas e mecanismos de *checkpointing* e

recuperação enfocando o ambiente orientado a objetos. Também foram pesquisados e estudados os trabalhos correlatos ao proposto, a fim de conhecer os mecanismos funcionais mais adequados para a modelagem e construção das classes de *checkpointing* e recuperação numa abordagem orientada a objetos.

A descrição e conclusões apresentadas neste texto pretendem resultar em um estudo abrangente e relevante sobre recuperação com base em *checkpoint* num enfoque para aplicações orientadas a objetos. A demonstração da viabilidade da biblioteca é feita através da realização de experimentos, fazendo o uso de algumas aplicações. Através destas é demonstrado o uso dos componentes da biblioteca e a forma de inserção das chamadas na programação. Adicionalmente, experimentos conduzem a uma discussão sobre o impacto dos procedimentos de *checkpointing* no desempenho temporal das aplicações.

Foi definido o emprego da linguagem de programação Java devido à portabilidade, à independência de plataforma e aos recursos disponíveis para serialização de objetos presentes na API de serialização Java. A tecnologia RMI também contribuiu na definição de uso da linguagem Java para a realização deste trabalho.

1.2 Trabalhos correlatos

Nesta seção, resumem-se os resultados das investigações realizadas para obter conhecimento de trabalhos relacionados ao apresentado nesta dissertação, envolvendo inclusive abordagens diferentes. Esta busca objetivou a aquisição de conhecimento das técnicas e mecanismos de *checkpointing* e recuperação, anteriormente explorados. No princípio, foram investigadas bibliotecas e ferramentas que operam em nível de usuário, desenvolvidas em linguagens imperativas sobre o sistema operacional *Unix/Linux*. São elas: a biblioteca *libckpt* [PLA95], a ferramenta *Epckpt* [PIN98] e o sistema *Condor* [LIT97]. Na seqüência desta seção, e seguindo a ordem da metodologia empregada no estudo, são apresentados trabalhos baseados num enfoque orientado a objetos, que estão mais próximos da concepção desta dissertação. São eles: o trabalho proposto por Killijian, Fabre e Ruiz-Garcia [KIL99], o trabalho proposto por Lawall e Muller [LAW2000] e o trabalho proposto por Hansen [HAN99].

A biblioteca *libckpt*, proposta por Plank *et al.* [PLA95], é uma ferramenta para *checkpointing* em um único processador; foi desenvolvida para o sistema operacional *Unix*, e construída sem modificações no *kernel*. Mas pode ser usada em um modo que é quase transparente ao programador, como também mantém a incorporação de diretivas do usuário na criação de *checkpoints*. *Libckpt* é uma biblioteca em nível de usuário projetada para uso em situações onde se deseja otimizar o tempo de execução total de uma aplicação, na presença de defeitos do sistema. *Libckpt* implementa *checkpointing* incremental (somente a porção que mudou desde o *checkpointing* prévio necessita ser salva no armazenamento estável) e *copy-on-write* (escrita das páginas de memória que foram realmente modificadas, operações de leitura apenas não provocam regravação destas páginas), duas otimizações bem conhecidas na literatura que são aplicáveis para propósitos gerais em *checkpointing* com um único processador. *Checkpointing* incremental usa proteção de páginas de *hardware* para identificar a porção inalterada do *checkpoint*: salvando somente as posições modificadas, reduz-se o tamanho de cada

checkpoint, com a conseqüente redução da sobrecarga de processamento. Adicionalmente implementa otimizações controladas pelos usuários na tentativa de melhorar o desempenho das técnicas de *checkpointing* usadas pela biblioteca. São consideradas duas maneiras referentes às otimizações controladas pelo usuário: exclusão de memória (as alocações de memória que não são mais referenciadas pela aplicação ou que não foram alteradas não necessitam serem salvas no arquivo de *checkpoint*) e *checkpointing* síncrono (é uma diretiva de usuário que permite ao programador especificar pontos no código-fonte onde é vantajosa a ocorrência do *checkpointing*). A biblioteca *libckpt* foi projetada para trabalhar com aplicações científicas escritas nas linguagens de programação "*C*" e "*Fortran*". A operação da *libckpt* pode ser configurada através de parâmetros armazenados em um arquivo específico (`.ckptrc`).

Mesmo não sendo construída a partir de um paradigma orientado a objetos, a biblioteca *libckpt* foi um dos principais trabalhos explorados, serviu de inspiração para especificação funcional da biblioteca desenvolvida nesta dissertação. Neste contexto, merecem destaque a atividade de *checkpointing* não incremental e *checkpointing* incremental, com ou sem controle de tempo. Também foi inspirado naquele trabalho, um controle do número máximo de arquivos de *checkpoint* configurável através de parâmetros. Uma outra idéia extraída de lá, considerada de suma importância é o arquivo de parâmetros para configuração das operações da biblioteca.

A ferramenta *Epckpt*, proposta por Pinheiro [PIN98], é um aperfeiçoamento das ferramentas existentes em muitos aspectos, e possui novas implementações. Como principais características desta ferramenta, destacam-se: a possibilidade de estabelecer *checkpoint* de um número maior de aplicações, independente das características particulares de comportamento de cada uma; a possibilidade de limitar o tamanho da imagem do *checkpointing* para um tamanho mínimo. Como novas implementações, *Epckpt* pode: estabelecer *checkpoint* de aplicações paralelas; e enviar a imagem do *checkpoint* diretamente para seu destino, sem a necessidade de um sistema de arquivos ou servidor especial. *Epckpt* é uma ferramenta implementada em nível do *kernel* do *Linux*, isto justifica a capacidade de estabelecer *checkpoint* para uma ampla extensão de aplicações. Trabalhando em nível de sistema operacional, não há necessidade das aplicações sofrerem modificações no código-fonte e necessitarem de operação com uma biblioteca especial. A ferramenta *Epckpt* é capaz de efetuar o *checkpointing* de um grupo de processos executando em paralelo em uma máquina SMP (*Simulating a Symmetric Multiprocessor*). Três novas chamadas de sistema foram criadas no *Linux*: (i) **`checkpoint()`**, faz o trabalho de salvar a imagem do processo para um arquivo, uma conexão *socket*, um *pipe* (para a troca de mensagens entre os processos) ou qualquer outra abstração suportada pelos sistemas de arquivos modernos; (ii) **`restart()`**, é responsável por reiniciar um processo no instante após ter sido efetuado o *checkpointing*; (iii) **`collect_data()`**, é responsável por comunicar ao *kernel* que um dado processo possui algumas informações detectadas durante o tempo de execução que podem ser salvas posteriormente no *checkpoint*.

O sistema *Condor*, proposto por Litzkow, Tannenbaum, Basney e Livny [LIT97], é um sistema de processamento distribuído para *Unix* desenvolvido em nível do usuário. Este sistema repassa serviços (processos a serem executados) para estações sem atividades na rede, resultando em uma utilização mais eficiente de recursos. Quando a estação de trabalho começa a ser utilizada por algum usuário, o sistema *Condor* imediatamente suspende a execução do serviço, e novamente migra para outra estação de trabalho sem atividade ou alternativamente para uma fila até que alguma estação

fique sem atividade. Um dos maiores componentes do Condor é a facilidade para transparentemente estabelecer *checkpoints* e reiniciar um processo, possivelmente em uma máquina diferente (migração de processo). O mecanismo de transparência é implementado em nível do usuário, sem nenhuma modificação no *kernel* do *Unix*. O método geral do *Condor* para *checkpointing* é armazenar todas as informações dos estados dos processos em um arquivo (ou conexão *socket*). Do ponto de vista do sistema operacional, o processo não é reiniciado ou migrado completamente, simplesmente é criado um novo processo e os estados são manipulados para emular os estados dos processos velhos.

Estes dois últimos trabalhos investigados, a ferramenta *Epckpt* [PIN98] e o sistema *Condor* [LIT97] contribuíram para reforçar a aquisição de conhecimento dos mecanismos de *checkpointing* e recuperação existentes, apesar de abordarem enfoques muito diferentes do proposto por esta dissertação.

O trabalho proposto por Killijian, Fabre e Ruiz-Garcia [KIL99] aborda programas orientados a objetos usando um enfoque reflexivo, provendo métodos de *checkpointing* para classes de aplicações. A técnica proposta é parte da definição de um protocolo meta-objeto para implementação de tolerância a falhas em aplicações CORBA (*Common Object Request Broker Architecture*). Usando definições de classes e implementações em tempo de compilação, é possível gerar métodos automaticamente, tais como **`SaveState`** e **`RestoreState`**, responsáveis para capturar e restaurar o estado dos objetos de uma certa classe de aplicação. Os autores propõem uma otimização deste enfoque usando reflexão, possibilitando efetuar *checkpointing* somente dos atributos que foram modificados desde o último *checkpoint*. Um objeto possui dois tipos de informações, o estado interno (denominado de variáveis de estado) e a informação em trânsito (mensagens invocadas). O estado interno de um objeto ativo é mapeado em um nível muito baixo para objeto em memória (segmentos, regiões, etc.) na qual são manipulados ou pelo sistema operacional ou por um *middleware* (uma camada em tempo de execução para aplicações de objetos). Outro enfoque desse trabalho consiste em prover ao usuário uma biblioteca de funções ou classes que trabalham com protocolos tolerantes a falhas e informações de estado. Em termos de orientação a objetos, classes de aplicação devem herdar de uma classe base que possui dois métodos: **`SaveState`** e **`RestoreState`**, que são definidos como métodos virtuais. Este segundo enfoque é claramente não transparente para o usuário e confia na habilidade deste para usar as funções da biblioteca ou para implementar métodos virtuais corretamente.

O trabalho mencionado no parágrafo anterior faz uso de reflexão em tempo de compilação para definir e gerar automaticamente a implementação dos métodos **`SaveState`** e **`RestoreState`**, responsáveis por capturar e restaurar o estado dos objetos da aplicação. Esse enfoque proposto pelos autores necessita ser aplicado a um modelo de objetos conveniente, como CORBA, por exemplo, que provê tal modelo. Esta dissertação diferencia-se muito daquele trabalho porque não foi utilizada reflexão para capturar os estados dos objetos da aplicação; os objetos da aplicação são passados via parâmetro para a biblioteca e então são serializados no arquivo de *checkpoint* através do mecanismo de *checkpointing*. Tampouco foi proposto que a biblioteca implementada nesta dissertação trabalhe com modelo de objetos utilizando CORBA.

O trabalho proposto por Lawall e Muller [LAW2000] investiga a otimização de *checkpointing* de programas Java em nível de linguagem. Eles descrevem como associar sistematicamente *checkpoint* incremental com classes Java. São adicionados métodos a

cada classe para salvar e restaurar o estado local. *Checkpointing* é então executado por um método genérico que invoca o método *checkpointing* de cada objeto. É proposto o uso de especialização de programas automáticos para otimizar automaticamente um algoritmo genérico de *checkpointing*, com base em informações fornecidas pelo programador sobre os aspectos fixos da estrutura do programa. A especialização de programa otimiza um programa para um contexto específico de uso. Essa técnica restringe a aplicabilidade de um programa, na troca por uma implementação mais eficiente. A especialização no contexto da linguagem orientada a objetos Java descreve como uma classe deveria ser especializada, declarando propriedades dos campos e métodos da classe especializada. Os métodos declarados são então especializados com respeito a essas informações. As especializações de classes são compiladas pelo JSCC - *Java Specialization Class Compiler* dentro de diretivas para o programa especializador, não fazendo parte da execução do programa. Lawall e Muller consideram que o estado de um programa orientado a objetos pode ser recuperado do conteúdo de campos de objetos presentes nos *checkpoints*. A implementação consiste da interface **`Checkpointable`**, a qual especifica os métodos que devem ser providos por cada objeto que realizará o *checkpointing*, e um objeto **`Checkpoint`**, que conduz o processo de salvamento dos estados dos objetos.

O trabalho investigado e descrito no parágrafo anterior é o que mais serviu de fonte de conhecimento sobre *checkpointing* orientado a objetos em Java para o trabalho proposto nesta dissertação, embora tenha formas de solução distintas. No trabalho de Lawall e Muller, métodos para prover o estabelecimento de *checkpoint* e recuperação são adicionados a cada classe da aplicação, o que aumenta muito o código-fonte; um *flag* também é adicionado no caso do mecanismo de *checkpoint* incremental para indicar se algum campo do objeto foi modificado desde o último *checkpoint*. O enfoque do trabalho desta dissertação diferencia-se nestes aspectos, com a simples adição de linhas para prover o estabelecimento de *checkpoint* e recuperação no código-fonte da aplicação. Com relação ao *checkpointing* incremental, é mantida, em memória principal, durante a execução da aplicação, a forma serializada dos estados dos objetos mais atuais. Estes estados são comparados com os próximos estados dos objetos para identificar se há modificação desses objetos. Os estados dos objetos da aplicação são recuperados a partir dos estados serializados dos objetos mais atuais presentes nos arquivos de *checkpoint*, ao contrário da recuperação de cada campo dos objetos a partir de arquivos de *checkpoint*, proposto por Lawall e Muller.

O trabalho proposto por Hansen [HAN99] apresenta um sistema para *checkpointing* de *threads* de programas em Java, ou seja, o objetivo central do autor é capturar e restaurar o estado da *thread* de um programa. O sistema é chamado de JCP – *Java Checkpointing* e possui alguns enfoques importantes. O sistema é independente de plataforma e pode efetuar o *checkpointing* de *threads* em qualquer padrão de implementação Java. O sistema garante a heterogeneidade, pois um programa para o qual foram estabelecidos *checkpoints* pode ser reiniciado em qualquer padrão de implementação Java, sem levar em consideração a plataforma de origem do *checkpoint*. Não é completamente transparente ao programador da aplicação; o sistema requer um esforço do programador para usá-lo. O sistema está baseado em transformação de programa. Um programa que usa as funcionalidades do sistema de *checkpointing* é transformado automaticamente em um programa semanticamente equivalente, no qual a funcionalidade de *checkpointing* é implementada pelo próprio programa. Para a atividade de *checkpointing*, o sistema JCP, adiciona uma quantidade pequena de sobrecarga de processamento. A baixa sobrecarga de processamento é alcançada às

custas do tamanho do código. O tamanho do programa transformado pode triplicar ou quadruplicar o tamanho do programa original. O trabalho provê um novo pacote (*package*) padrão denominado de **`jcp.api`**, que contém uma nova classe *thread* chamada **`CpThread`**; esta é uma subclasse da classe *Thread* (existente na linguagem Java), adicionando primitivas de *checkpointing*.

Nesse trabalho anterior, segundo Hansen [HAN99], não há como efetuar o *checkpointing* de programas Java sem perder a independência de plataforma, porque Java não possui mecanismo de *checkpointing* embutido. Segundo o autor, para estabelecer *checkpoints* em um programa em Java, teria que usar um padrão de implementação desvinculado da linguagem com um mecanismo de *checkpointing* embutido ou executar toda a implementação Java em um mecanismo de *checkpointing* geral implementado no sistema operacional ou sobre ele. A solução apresentada pelo autor é prover um sistema para *checkpointing* de *threads* de um programa em Java, usando transformação de programa para capturar e restaurar o estado dos objetos. Estas considerações não foram adotadas para o projeto da biblioteca proposta por esta dissertação, que se baseia num enfoque diferente. Destina-se basicamente a prover *checkpointing* e recuperação de aplicações computacionais escritas em Java, não visando programas que façam uso de *threads*. O enfoque adotado permite o uso da biblioteca tanto no sistema operacional *Windows* como no *Linux*. Com relação ao tamanho do código, este apenas terá o acréscimo de linhas que se destinam basicamente para prover o salvamento dos estados dos objetos e a recuperação.

Para finalizar, foi investigado o trabalho proposto por Kasbekar *et al* [KAS99], que não aborda especificamente características usadas no projeto da biblioteca proposto por esta dissertação. Esse trabalho apresenta um mecanismo de persistência implementado em uma biblioteca denominado de *Member Analyzer*. A biblioteca provê um mecanismo aperfeiçoado para *checkpointing* de aplicações orientadas a objetos desenvolvidas na linguagem de programação C++. Satisfazer todas as exigências necessárias para persistência de objetos em um nível transparente para o usuário é uma das metas desta biblioteca. A única preocupação do programador é a invocação do mecanismo de *checkpointing* para a atividade de salvamento dos estados dos objetos. O mecanismo implementado por esta biblioteca pode ser capaz de manipular de forma confiável e eficaz qualquer estrutura de dados de C++, mesmo as consideradas mais complexas. A linguagem C++ possui várias características que são difíceis e complexas de manipular. O uso da biblioteca não deve mudar o modo característico que a linguagem trabalha, não exigindo aos programadores a mudança do estilo de codificação de cada um; também não impõe restrições no uso de qualquer característica de programação. A biblioteca, em tempo de execução, contém um objeto global denominado de **`RuntimeSystem`** que gerencia todos os dados requeridos à execução do programa. Para salvar e restaurar o estado de um objeto, são usados os métodos: **`Checkpoint(thisPtr)`** e **`Restore(thisPtr)`**. Uma chamada ao método **`Checkpoint()`** insere o objeto no **`CheckpointList`** (uma lista contendo ponteiros para os objetos a serem salvos no arquivo de *checkpoint*) e os serializa para o meio estável. Uma chamada ao método **`Restore()`** executa os mesmos passos usando o **`RecoverList`** ao invés do **`CheckpointList`** para retomar os estados dos objetos do programa.

1.3 Estrutura desta Dissertação

Após este capítulo introdutório, a presente dissertação está organizada conforme a estrutura que segue.

No capítulo 2, são elucidados os aspectos relevantes aos conceitos básicos que servem de suporte para o texto desta dissertação. Um primeiro enfoque é dado à recuperação, onde são esclarecidas as questões de recuperação na estrutura de processos e na estrutura de objetos. Após, são mostradas as características do paradigma da orientação a objetos, são abordados os conceitos de persistência de objetos, conceitos de serialização e aspectos importantes da tecnologia Java RMI.

O capítulo 3 aborda as características e funcionalidades da biblioteca proposta nesta dissertação: mecanismos de *checkpointing* não incremental e *checkpointing* incremental com ou sem controle de tempo, mecanismo de recuperação, controle do número de arquivos de *checkpoint* salvos no disco e registros de informações de tempos de execução.

No capítulo 4, é feita a descrição das classes propostas para a biblioteca desta dissertação. Inicialmente, comenta-se um pouco sobre a linguagem UML e é ilustrado um diagrama contendo as classes propostas. Na seqüência, todas as classes com seus códigos correspondentes são explicados e ilustrados.

O capítulo 5 apresenta as aplicações utilizadas para testes e avaliação funcional da biblioteca, bem como as formas de utilização. Também são mostrados resultados dos experimentos realizados visando a avaliação de desempenho.

Por fim, o capítulo 6 apresenta as conclusões desta dissertação, bem como os problemas encontrados, e aspectos que ficaram em aberto como sugestão para o direcionamento de trabalhos futuros.

O código-fonte das classes que compõem a biblioteca, assim como os códigos-fonte das aplicações usadas para teste funcional, estão em anexo, ao final desta dissertação.

2 Conceitos básicos e aspectos de implementação

Neste capítulo, são apresentados os conceitos básicos para uma melhor compreensão dos assuntos abordados nesta dissertação. Inicialmente, são descritas informações relevantes sobre recuperação; posteriormente, as características do paradigma de orientação a objetos, utilizado para a implementação prática da biblioteca proposta nesta dissertação. Por fim, aproveita-se para detalhar um pouco mais os conceitos de persistência e serialização de objetos e apresentar os aspectos importantes da tecnologia Java RMI.

2.1 Recuperação

A recuperação pode parecer uma tarefa simples quando for associada a ela a idéia de que basta reiniciar o computador após a ocorrência de uma falha e re-executar o processo para recomeçar as atividades interrompidas. Entretanto, questões de disponibilidade e o desejo de não desperdiçar tempo empregado em operações realizadas, minimizando o tempo total de execução de aplicações computacionais, diante de perdas causadas por falhas, fazem com que a questão da recuperação não seja resolvida tão facilmente.

Recuperação, no âmbito de tolerância a falhas em um **sistema computacional**[3], refere-se a restaurar um sistema para o estado operacional normal. Se, por exemplo, um processo falhou após ter modificado parcialmente uma base de dados (sem concluir a operação), então é importante que todas as modificações feitas na base de dados pelo processo falho sejam desfeitas. Em outras palavras, se o processo executou por algum tempo antes de falhar, a melhor opção é executar novamente o processo desde o ponto onde ocorreu a falha e retornar à execução. O reinício, a partir do ponto da falha, evita a situação de ter a retomada da execução do processo desde o início da operação, pois isto incorreria em maior tempo consumido e operação custosa [SIN94].

Com base na estrutura de processos, a recuperação pode ser implementada segundo dois princípios conhecidos como **recuperação por avanço** (*forward recovery*) e **recuperação por retorno** ou por retrocesso (*backward recovery*) [AND81]:

- **recuperação por avanço**: baseia-se na transformação do estado errôneo, efetuando correções para remover erros. Assim o sistema é conduzido a um novo estado, não ocorrido anteriormente, ou a um estado por onde o sistema não passou desde a última manifestação de erro;
- **recuperação por retorno**: baseia-se na restauração de um processo para um estado anterior livre de falhas [RAN79]. Por isso, algoritmos baseados em recuperação por retorno salvam periodicamente os estados de um processo durante uma execução livre de falha e, após uma falha, reiniciam a partir do estado mais recente, reduzindo a quantidade de trabalho perdido.

[3] Um sistema computacional consiste de um conjunto de componentes de *hardware* e *software* para prover um serviço específico.

O projeto da recuperação por retorno é mais simples que o de recuperação por avanço, pois a recuperação por avanço depende de uma análise prévia das falhas a serem suportadas, enquanto que na recuperação por retorno não é necessário conhecê-las.

Para a realização da atividade de recuperação, um arquivo de *checkpoint* normalmente composto do conteúdo da memória do processo e os estados dos registradores são lidos do meio de armazenamento. Um novo processo é gerado, o qual inicializa o conteúdo da memória a partir do arquivo de *checkpoint* e então reajusta o estado dos registradores efetuando um reinício do programa que havia falhado [PLA97]. Este é o princípio básico para a recuperação na estrutura de processos, o que difere muito do paradigma orientado a objetos, o qual esta dissertação enfoca. A recuperação com base na abordagem orientada a objetos é realizada a partir dos estados de objetos salvos nos arquivos de *checkpoints* por meio do mecanismo de *checkpointing*. Este enfoque não trabalha diretamente com as páginas de memória e estado dos registradores, pois a alocação de objetos não é de controle do programador, objetos modificados não são todos armazenados na mesma página de memória, além de que uma única página de memória pode conter também objetos cujos estados não são interessantes para serem salvos no arquivo de *checkpoint* [LAW2000].

Algoritmos genéricos para *checkpointing* e recuperação, num enfoque orientado a objetos, para aplicações escritas em Java, não são comumente encontrados na literatura, como acontece tradicionalmente com os algoritmos de recuperação na estrutura de processos. Apenas em publicações mais recentes foram encontrados estratégias e métodos para prover *checkpointing*, como nos trabalhos de Lawall e Muller [LAW2000] e Hansen [HAN99].

Em Lawall e Muller [LAW2000], por exemplo, os *checkpoints* são criados usando um protocolo bloqueante, e são salvos para o armazenamento estável de forma assíncrona. Cada objeto cujo estado será salvo no arquivo de *checkpoint* contém um único identificador e métodos que descrevem como registrar o estado do objeto e seus objetos filhos. Neste enfoque, métodos para prover o estabelecimento de *checkpoints* e recuperação são adicionados a cada classe da aplicação. *Checkpointing* é executado por um método **`checkpoint`** genérico que executa o mecanismo associado a cada objeto. Adicionalmente, para implementar o *checkpoint* incremental, os autores propõem a inclusão de um *flag* indicando se algum campo do objeto foi modificado desde o último *checkpoint* prévio. Segundo Kasbekar *et al*. [KAS99] e Killijian *et al*. [KIL99], esse código de *checkpointing* pode também ser adicionado manualmente ou gerado automaticamente usando um pré-processador. Em cada caso, um local no código é escolhido para inclusões de chamadas para salvar e restaurar o estado de um objeto, sempre respeitando as características impostas pela programação orientada a objetos, como o encapsulamento. Desse modo aumenta a segurança global do programa, e simplifica a manutenção do mesmo.

Lawall e Muller mostram que o estado de um programa pode ser recuperado do conteúdo de campos de objetos a partir de arquivos de *checkpoint*. No contexto de *checkpointing* incremental de programas Java, o mecanismo de *checkpointing* se preocupa em percorrer recursivamente os objetos e registrar o estado local mais atual de cada um; a pilha é omitida. Estratégias similares foram propostas por outros autores, incluindo Kasbekar *et al*. [KAS99] e Killijian *et al*. [KIL99], cujos trabalhos foram descritos na seção 1.2.

2.2 Orientação a Objetos

O modelo de objetos apresenta-se como um modelo promissor para o desenvolvimento de *software* mais confiável devido a características inerentes ao próprio modelo de objetos, tais como abstração de dados, encapsulamento, herança e reutilização de objetos (componentes). O uso de técnicas orientadas a objetos facilita o controle da complexidade do sistema porque promove uma melhor estruturação de seus componentes, e também permite que componentes já validados sejam reutilizados [BUZ98].

O método orientado a objetos para o desenvolvimento de *software* é, com certeza, uma parte do fluxo principal, simplesmente porque tem sido provado seu valor para a construção de sistemas em todos os tipos de domínios de problemas, abrangendo todos os graus de tamanho e de complexidade. Além disso, muitas linguagens, sistemas operacionais e ferramentas contemporâneas são, de alguma forma, orientadas a objetos, fortalecendo a visão de mundo em termos de objetos [BOO2000].

Linguagens orientadas a objetos renovam e enfatizam o conceito de **reutilização** de código, sob forma de classes de objetos genéricos mantidos em bibliotecas e selecionados por critério de funcionalidades e adequação à aplicação em desenvolvimento [LIS95]. A tecnologia orientada a objetos conduz para o reuso, e o reuso de componentes de programas conduz para um desenvolvimento de *software* mais rápido e programas de maior qualidade. *Softwares* orientados a objetos são mais fáceis de serem mantidos porque suas estruturas são naturalmente independentes [RUM94].

As vantagens decorrentes da utilização de objetos na construção de aplicações são muitas, mas podem ser citadas as seguintes [AMA2001]:

- **simplicidade**: os objetos escondem a complexidade do código. Pode-se criar uma complexa aplicação gráfica usando botões, janelas, barras de rolagem, etc., sem conhecer a complexidade do código utilizado para criá-los;
- **reutilização de código**: um objeto, depois de criado, pode ser reutilizado por outras aplicações, ter suas funções estendidas e ser usado, em combinação com outros, como bloco básico para a construção de sistemas mais complexos;
- **inclusão dinâmica**: objetos podem ser inseridos dinamicamente no programa, durante a execução. Isso permite que vários programas compartilhem os mesmos objetos e classes, reduzindo o seu tamanho final.

Segundo Buzato e Rubira [BUZ98], diversos autores divergem quanto às características que fazem uma linguagem ser orientada a objetos, mas a maioria concorda que o paradigma se baseia em quatro princípios básicos: **abstração**, **encapsulamento**, **herança** e **polimorfismo**.

A **abstração** consiste na concentração nos aspectos essenciais (próprios) de uma entidade e em ignorar suas propriedades acidentais. No desenvolvimento de sistemas, isso significa concentrar-se no que um objeto é e faz, antes de decidir como ele deve ser implementado. Muitas linguagens modernas permitem abstrações de dados, mas a capacidade de usar herança e polimorfismo proporciona maior poder. Na prática, as abstrações são fortemente acopladas com algum modelo de execução, que descreve o comportamento destas em tempo de execução [GAR98].

A área de orientação a objetos é extensa e crescente. O modelo de objetos representa em *software* objetos que podem ser encontrados no mundo real. Esses objetos podem ser de vários tipos, representando entidades físicas (por exemplo: aviões, robôs, etc.) ou abstratas (por exemplo: listas, pilhas, filas, etc.) [BUZ98]. O sucesso da orientação a objetos se deve, provavelmente, à habilidade em modelar o domínio da aplicação ao invés da arquitetura da máquina em questão (**enfoque imperativo**) ou mapear conceitos matemáticos (**enfoque funcional**).

A característica mais importante (e diferente) da abordagem orientada a objetos para desenvolvimento de *software* é a unificação, através do conceito de objetos, de dois elementos que, tradicionalmente têm sido considerados separadamente em paradigmas de programação tradicionais: **dados** e **funções**. A unificação destes dois elementos de programação resulta no chamado encapsulamento[4] [BUZ98].

Um **objeto** é uma entidade que encapsula informação de estado ou dados e possui um conjunto de operações associadas que manipulam estes dados. Um objeto pode ser usado para modelar entidades do mundo real ou de um mundo imaginário [MEY88]. Uma **operação** é definida como sendo alguma ação que um objeto realiza sobre seus próprios dados que pode resultar em uma mudança de estado. Em geral, o estado de um objeto é completamente escondido e protegido de outros objetos e a única maneira de examiná-lo é através da invocação de uma operação (envio de uma mensagem) para este fim. Objetos representam um comportamento bem definido e uma identidade que é única. **Comportamento** define o modo como um objeto age e reage em termos das suas mudanças de estado e envio de mensagens e é completamente definido pelas suas operações. **Identidade** é a propriedade de um objeto que o distingue de outros objetos – cada objeto possui um tipo, um identificador único e seus próprios dados.

Uma **classe** é a descrição de um molde que especifica as propriedades e o comportamento para um conjunto de objetos similares. Toda classe possui um nome e um corpo que define o conjunto de atributos e operações que suas instâncias possuem. Todo objeto é **instância** de apenas uma classe. Os **atributos** são propriedades nomeadas de um objeto e armazenam o estado abstrato de cada objeto. Operações ou **métodos** caracterizam o comportamento de um objeto e é o único meio para fazer acesso, manipular e modificar os valores dos atributos de um objeto [BUZ98]. O mundo externo interage com um objeto invocando-lhe métodos ou passando-lhe mensagens [MEY88].

Encapsulamento, também chamado de ocultamento de informações, consiste na separação dos aspectos externos de um objeto dos detalhes internos da implementação. O encapsulamento impede que um programa se torne tão interdependente que uma pequena modificação possa causar grandes efeitos de propagação. A implementação de um objeto pode ser modificada sem que isso afete as aplicações que o utilizam. Pode-se modificar a implementação de um objeto para melhorar o desempenho, eliminar um erro ou consolidar um código.

Encapsulamento é o processo de combinar tipos de dados, dados e funções relacionadas em um único bloco de organização e só permitir o acesso a eles através de métodos determinados; é definido como sendo uma técnica para minimizar as interdependências entre módulos programados independentemente através de interfaces externas restritas [BUZ98].

A **herança** é um mecanismo para derivar novas classes a partir de classes existentes através de um processo de refinamento. Uma classe derivada herda a

[4] Refere-se ao acesso restrito à representação dos objetos.

representação de dados e operações de sua classe base, mas pode seletivamente adicionar novas operações, estender a representação de dados ou **redefinir** a implementação de operações já existentes. Uma classe base proporciona a funcionalidade que é comum a todas as suas classes derivadas, enquanto que uma classe derivada proporciona a funcionalidade adicional que especializa o seu comportamento [BUZ98]. Herança é o compartilhamento de atributos e operações entre classes com base em um relacionamento hierárquico [KHO90]. Uma classe pode ser definida de forma abrangente e depois refinada em sucessivas subclasses mais específicas. Cada subclasse incorpora, ou herda, todas as propriedades de sua(s) superclasse(s) e acrescenta suas próprias e exclusivas características.

Nesse contexto, cada classe é declarada como uma subclasse de uma ou mais superclasses. Quando existe mais de uma superclasse, a relação de herança é denominada herança múltipla. Podem existir problemas de conflito entre informações herdadas por uma subclasse proveniente de diferentes superclasses, como, por exemplo, serem herdadas variáveis com mesmo nome, mas de tipos diferentes.

Polimorfismo significa que a mesma operação pode atuar de modos diversos em classes diferentes. No contexto de orientação a objetos, polimorfismo significa que diferentes tipos de objetos podem responder a uma mesma mensagem de forma diferente [BUZ98]. Por exemplo, pode-se definir um método denominado **`imprime()`** em diversas classes diferentes, mas cada versão deste método é adaptada para cada tipo de objeto diferente que será impresso. Um objeto supostamente da classe **`Cheque`** irá responder a mensagem de uma maneira, um objeto da classe **`Relatorio`** irá responder de outra forma e um objeto da classe **`Fotografia`** responderá de uma outra forma ainda. O método **`imprime()`** é polimórfico, pois é implementado diferentemente por diferentes classes de objetos.

No mundo real uma operação é simplesmente uma abstração de um comportamento análogo entre diferentes tipos de objetos. Cada objeto "sabe como" executar suas próprias operações. Entretanto, uma linguagem de programação baseada em objetos seleciona automaticamente o método correto para implementar uma operação com base no nome da operação e na classe do objeto que esteja sendo operado. O usuário de uma operação não necessita saber quantos métodos existem para implementar uma determinada operação polimórfica. Novas classes podem ser adicionadas sem que se modifique o código existente [RUM94].

Uma mensagem enviada a um objeto provoca um determinado comportamento no objeto receptor e, se a mesma mensagem for enviada a diferentes objetos, o comportamento poderá ser distinto. A esta interpretação particular de cada objeto frente à mesma mensagem, dá-se o nome de polimorfismo [LIS95].

A programação orientada a objetos é freqüentemente tida como um novo paradigma[5] de programação. Outros paradigmas são a programação procedimental (linguagens como C e Pascal), a programação lógica (Prolog) e a programação funcional (FP ou Haskell). A orientação a objetos permite que *softwares* sejam construídos a partir de componentes de propósitos gerais reutilizáveis, o que demonstra um modelo promissor mais confiável e modular. A utilização de técnicas orientadas a objetos facilita o controle da complexidade do sistema, porque promove uma melhor

[5] Entende-se paradigma como um conjunto de teorias, métodos e padrões que juntos representam uma forma de organizar o conhecimento, isto é, uma forma de ver o mundo.

estruturação de seus componentes, e também permite que componentes já validados sejam reutilizados [BUZ98].

Foram observadas na linguagem Java as vantagens decorrentes da utilização do paradigma da orientação a objetos. Java foi criada para atender os requisitos do mundo real, possuindo importantes aspectos citados a seguir [NAU96]:

- **simples e poderosa**: os métodos para realizar uma determinada tarefa são claros e em número reduzido. Java propicia o poder de expressar cada idéia de forma clara e limpa, orientada a objetos, sem ter de expor todos os perigosos funcionamentos internos do sistema;
- **segura**: os programas executados na máquina virtual Java são submetidos a verificadores de código e gerenciadores de segurança, cujas normas podem ser específicas para cada aplicativo;
- **robusta**: a existência de um sistema de tratamento de exceções bem estruturado, aliado aos verificadores dinâmicos de código e às bibliotecas de componentes robustos que não permitem ao usuário ignorar a ocorrência de erros, tornam Java uma linguagem atraente para o desenvolvimento de aplicativos confiáveis;
- **interativa**: foi criada para atender a requisitos do mundo real, entre eles a criação de programas interativos e em rede;
- **neutra em relação à arquitetura**: a linguagem Java foi desenvolvida com preocupações em garantir a longevidade e portabilidade de código entre plataformas diferentes;
- **interpretada e de alto desempenho**: a independência de plataforma é alcançada em Java através de uma representação intermediária chamada *bytecode*, que foi cuidadosamente projetada para que seja fácil traduzi-la diretamente para o código da máquina nativa e tenha um alto desempenho.

O desempenho de Java é um fator até o momento preocupante para os programadores e tem pesado negativamente sobre a linguagem, mas pesquisas estão sendo feitas no sentido de melhorar esta situação. Estas pesquisas mostram que o desempenho de Java poderá exceder o desempenho de linguagens como C++, pois a compilação dinâmica dá ao compilador Java acesso em tempo de execução a informações que não estão disponíveis a um compilador C++ [REI2000].

As considerações descritas nesta seção motivaram a utilização da orientação a objetos e, em particular, o uso da linguagem Java para a implementação da biblioteca de *checkpointing* e recuperação nesta dissertação.

2.3 Persistência

A persistência, em linguagens de programação orientada a objetos, é a habilidade dos objetos existirem além do tempo de vida do programa, no qual eles foram criados. O tempo de vida de um objeto inicia quando ele é criado pelo operador *new*, e subsiste até ser destruído pelo *garbage collector* da JVM – *Java Virtual Machine* [MAT97]. O

coletor de lixo (*garbage collector*) é responsável pela liberação de memória que não está mais sendo referenciada [THO97].

Os princípios de persistência de objetos são independentes de linguagem e podem ser utilizados em qualquer linguagem específica [JOR98]. Em resumo, eles são como segue:

- o tempo de vida dos objetos é determinado pela referência de outros objetos persistentes. O salvamento de um objeto no armazenamento estável também deve assegurar o salvamento de todos os objetos que são referenciados por ele. Assim, um objeto torna-se persistente se é explicitamente salvo no armazenamento estável, ou se é referenciado por outro objeto persistente;
- durante a execução normal de um programa com objetos persistentes e não-persistentes, deveria ser indistinguível se o código está operando em um objeto persistente ou em um objeto não persistente.

Geralmente, a persistência é implementada para preservar o estado de um objeto. Neste contexto, preservar o estado significa converter o objeto para uma seqüência de *bytes* e armazená-los em um meio que prolongue sua vida útil. Um objeto persistente pode ser armazenado em um arquivo para posterior uso ou ser transmitido para outra máquina [BER2000].

A persistência consiste em armazenar o estado de um objeto, ou conjunto de objetos. Por exemplo, o programa P1 armazena em disco os objetos *obj1* e *obj2*; assim que houver necessidade, em uma futura execução, será possível restaurar o estado de execução de P1, utilizando os dados persistentes de *obj1* e *obj2* [BER2000].

Os arquivos são freqüentemente utilizados para armazenar informações, algumas vezes simples como um arquivo-texto; outras, complexas como um diagrama de circuitos. Os arquivos constituem a base da persistência em Java, pois neles é possível salvar o estado de um objeto de forma simples, utilizando a API de serialização Java [BER2000].

2.4 Serialização

A capacidade de armazenar objetos em memória estável e, posteriormente, recuperá-los e trazê-los para a memória principal, ou transmiti-los de um computador para outro, é uma característica essencial em muitas aplicações distribuídas e orientadas a objetos. Em Java, essa habilidade de persistência pode ser conseguida através da serialização dos objetos [AMA99].

A serialização compreende a transformação da representação de um objeto em uma seqüência de *bytes*, conhecida como forma serial, o que posteriormente pode ser revertida para a representação original, de modo a reconstituir o objeto. A representação serial possibilita que um objeto possa ser gravado em arquivos para torná-lo persistente, bem como transportado por uma rede de comunicação, para se tornar disponível em outra plataforma. No momento que ocorre o envio de mensagens entre objetos remotos, a serialização de objetos é necessária para o transporte dos argumentos e dos valores de retorno, ou seja, para a comunicação dos objetos [ECK98, COK97].

Através da serialização é possível criar um objeto em uma certa plataforma, serializá-lo, e enviá-lo através da rede para uma máquina da mesma ou de outra plataforma, onde será corretamente construído. Além disso, o programador não precisa se preocupar com a representação dos dados nas diferentes máquinas, com a ordem dos *bytes* ou com quaisquer outros detalhes. A serialização de objetos possibilita a escrita de objetos em uma seqüência de *bytes* (*stream*) e o armazenamento persistente de objetos. A escrita de objetos em seqüências de *bytes*, geralmente, é utilizada para enviar fluxo de *bytes* (estado do objeto) pela rede, enquanto o armazenamento persistente possibilita salvar o estado do objeto em um meio permanente.

O processo de serialização de objetos compreende os seguintes passos:

- converter a representação de um objeto na memória para uma seqüência de *bytes*;
- a forma serial pode então ser enviada para um arquivo em disco ou para uma conexão de rede.

O processo de de-serialização ocorre quando é necessário ter novamente o objeto representado na memória: ao ser lido um objeto a partir de um arquivo ou quando o objeto chegou à sua plataforma de destino, compreendendo os seguintes passos:

- transformação da representação serial para a representação de memória;
- instalação do objeto na memória e inicialização de seu *handle* (objeto manipulador).

Por exemplo, para serializar um objeto da classe *Hashtable*, definida no pacote *java.util*, armazenando seu conteúdo em um arquivo em disco, usa-se a seqüência de passos mostrada na figura 2.1 [RIC2001].

```
 1 // criar/manipular o objeto
 2 Hashtable dados = new Hashtable();
 3 ...
 4 // definir o nome do arquivo
 5 String nomearquivo = ...;
 6 // abrir o arquivo de saída
 7 File arquivo = new File(nomearquivo);
 8 if(!arquivo.exists()){
 9   arquivo.createNewFile();
10 }
11 FileOutputStream out = new FileOutputStream(arquivo);
12 // associar ao arquivo o ObjectOutputStream
13 ObjectOutputStream s = new ObjectOutputStream(out);
14 // serializar o objeto
15 s.writeObject(dados);
```

FIGURA 2.1 – Seqüência de passos para serializar um objeto [RIC2001].

O processo inverso, a de-serialização, mostrado na figura 2.2, permite ler essa representação de um objeto a partir de um **`ObjectInputStream`** usando o método **`readObject()`**.

```
 1 FileInputStream in = new FileInputStream(arquivo);
 2 ObjectInputStream s = new ObjectInputStream(in);
 3 dados = (Hashtable)s.readObject();
```

FIGURA 2.2 – Exemplo de de-serialização de um objeto.

Se os valores dos campos incluem referências para outros objetos, então estes objetos são também serializados recursivamente. Referências múltiplas para o mesmo objeto e referências circulares devem ser manipuladas corretamente pela atividade de serialização. Quando de-serializar uma instância de classe, os campos transientes (campos não-persistentes) serão estabelecidos para seus valores-padrão (por exemplo, 0 para *int*, e *null* para tipos de referência) [HAN99].

A serialização de uma instância de classe, isto é, a serialização de um objeto, além de capturar as propriedades particulares do objeto, também inclui informações a respeito da classe do objeto. A de-serialização de uma instância de classe, utiliza estas informações, em tempo de execução do sistema, para selecionar a classe correspondente do objeto [HAN99].

Para objetos de um vetor (*array objects*), a semântica de serialização é capturar o tamanho do vetor e os valores dos componentes. Se estes valores são referências a outros objetos, esses objetos também são serializados como instâncias de classe referenciando outros objetos [HAN99].

2.4.1 Serialização e de-serialização de objetos com a API de serialização Java

Serialização é um processo transitivo, ou seja, subclasses serializáveis de classes serializáveis são automaticamente incorporadas à representação serializada do objeto raiz. Para que o processo de de-serialização possa operar corretamente, todas as classes envolvidas no processo devem ter um construtor padrão (*default*) sem argumentos [RIC2001].

Os objetivos propostos pela serialização de objetos, oferecidos pela linguagem Java são [JAV98]:

- possuir um mecanismo extensível, ainda que simples;
- manter o tipo de objeto Java, bem como as propriedades de segurança, na forma serializada;
- prover um mecanismo extensível para suporte às operações de serialização e de-serialização, úteis para objetos remotos;
- prover um mecanismo extensível para suporte à persistência de objetos Java;
- permitir ao objeto a definição de seu formato externo.

Em Java, os objetos são armazenados numa seqüência de *bytes* (*stream*), suportando duas interfaces: **`Externalizable`** e **`Serializable`**. Para objetos utilizando a interface **`Serializable`**, a seqüência de caracteres inclui informações suficientes para restituir os campos ali contidos em uma versão compatível da classe [JAV98].

Utilizando-se a API de serialização da linguagem Java, é possível serializar e de-serializar objetos de duas formas. Na primeira, a classe implementa a interface **`Serializable`**; na segunda, a classe implementa a interface **`Externalizable`**. A diferença entre as duas está nos métodos que as mesmas utilizam e nas informações que serializam [ECK98]. Classes que não implementam essas interfaces não podem ter o

estado de seus objetos serializado e ou de-serializado. A classe implementa a interface **`Serializable`** ou **`Externalizable`** para indicar se é possível ou não que as instâncias da classe possam ser serializadas no armazenamento persistente ou ser enviadas pela rede. Para a implementação deste trabalho foi utilizada a interface **`Serializable`**. Esta interface é implementada pelo objeto contêiner que compõe a estrutura física do arquivo de *checkpoint* (encontra-se na subseção 4.2.2 informações sobre a estrutura do arquivo de *checkpoint* utilizada nesta dissertação). A interface **`Serializable`** é descrita a seguir.

2.4.1.1 Interface Serializable

Objetos de classes para os quais são previstas serializações e de-serializações devem implementar a interface **`Serializable`**, do pacote **`java.io`**. A interface **`Serializable`** não define nenhum método ou campo. Ela é apenas uma indicação de que a classe que a implementa é compatível com o mecanismo de serialização. Quando se está utilizando a interface **`Serializable`**, algumas considerações devem ser observadas:

- as subclasses de classes não-serializáveis devem assumir a responsabilidade de armazenar e restaurar o estado das variáveis herdadas (*public* e *protected*). Isto será possível somente, se a superclasse possuir um construtor não parametrizado (sem argumentos) que inicialize o estado da classe, caso contrário, uma exceção (*NotSerializableException*) em tempo de execução ocorrerá;
- esta interface indica que devem ser armazenadas as seguintes informações: a classe do objeto, a assinatura da classe e o valor de todas as variáveis não transientes e não estáticas, incluindo membros que referenciam outros objetos. Somente os dados dos objetos e a declaração das classes são codificados na seqüência de *bytes*; os *bytecodes* que possuem a implementação dos métodos, não são armazenados quando o objeto é serializado. No momento de restaurar o estado do objeto, a declaração da classe é lida e os mecanismos triviais para carregar a classe são usados [MAT97];
- o método **`writeObject()`**, definido na classe **`ObjectOutputStream`**, é responsável pela serialização dos objetos, ou seja, com este método é possível escrever o estado de um objeto de uma classe particular. O estado compreende a assinatura de sua classe e os valores de qualquer atributo herdado, além dos atributos não-transientes e não-estáticos [JAV98]. O método **`writeObjetc()`** não realiza sincronização no objeto que está sendo serializado. Caso existam várias *threads* usando um mesmo objeto, uma *thread* pode estar serializando um objeto, enquanto outra está manipulando atributos deste mesmo objeto. É necessário realizar a sincronização explícita [MAT97];
- a de-serialização é realizada pelo método **`readObject()`**, definido em **`ObjectInputStream`**. Com este método é possível restaurar o estado do objeto serializado, ou seja, desfazer todas as serializações, incluindo a recuperação das referências de objetos múltiplos [JAV98].

2.5 Java RMI

As informações desta seção e suas subseções foram obtidas em Ricarte [RIC2001], excetuando-se as frases ou parágrafos onde aparecer outra referência.

RMI – *Remote Method Invocation* é um recurso do ambiente de desenvolvimento Java que permite a comunicação de funções entre duas máquinas virtuais Java (JVM) residentes em dois equipamentos distintos. Um equipamento servidor implementa uma funcionalidade, que pode ser utilizada remotamente, por outra aplicação, o cliente. Em termos de projeto, o que vai garantir a compatibilidade entre a aplicação cliente e a aplicação servidora é uma interface comum. A interface é pré-compilada pelo pré-compilador RMI onde são gerados módulos *stub* para a máquina cliente e *skeleton* para a máquina servidora.

RMI é uma das abordagens da tecnologia Java para prover as funcionalidades de uma plataforma de objetos distribuídos. Esse sistema de objetos distribuídos faz parte do núcleo básico de Java desde a versão *jdk1.1*, com sua API sendo especificada através do pacote **`java.rmi`** e seus sub-pacotes. Através da utilização da arquitetura RMI, é possível que um objeto ativo em uma máquina virtual Java possa interagir com objetos de outras máquinas virtuais Java, independentemente da localização dessas máquinas virtuais.

No modelo de objetos distribuídos da linguagem Java, um objeto remoto possibilita que seus métodos possam ser invocados de uma máquina virtual Java, possivelmente, localizada em outro servidor. Como o próprio nome diz, RMI habilita a chamada remota de métodos. Também possibilita que o método de um objeto, em uma máquina virtual, chame um método de outro objeto, em outra máquina virtual. Isto é feito com a mesma sintaxe e facilidade de uma invocação local [BUR97]. Com RMI, um programa cliente pode invocar um método em um objeto remoto, da mesma forma que faz com um método de um objeto local. Todos os detalhes de conexão de rede estão ocultos, permitindo, assim, que o modelo de objetos tenha sua interface pública mantida através da rede, sem necessitar expor detalhes irrelevantes como conexões, portas ou endereços [ROC98].

O mecanismo de RMI suporta comunicação entre processos em um alto nível de abstração. Fornece suporte para novas capacidades [BUR97]:

1. carga dinâmica de classes;
2. *callbacks* para *applets*;
3. modelo de objetos distribuídos.

Em sistemas distribuídos, a comunicação ocorre entre computadores em diferentes espaços de endereçamento, ou seja, a linguagem Java utiliza o recurso de *sockets* para efetuar a comunicação. Apesar de mostrarem-se flexíveis, os *sockets* requerem que cliente e servidor entrem em acordo quanto ao protocolo em nível de aplicação, de maneira a codificar e decodificar a mensagem que está sendo trocada. Porém, os protocolos podem ser projetados de forma incorreta. Uma alternativa é utilizar RPC – *Remote Procedure Call*, a qual abstrai a interface de comunicação para o nível de um procedimento local, ou seja, ao invés do programador trabalhar com *sockets*, ele tem a ilusão de chamar um procedimento local [RMI97].

Entretanto, o mecanismo de RPC não tem se mostrado eficiente em sistemas de objetos distribuídos, onde acontece a comunicação entre objetos em nível de programa. Dessa forma, os sistemas de objetos distribuídos requerem invocação de métodos remotos, ou RMI, onde o *stub* do objeto local gerencia a invocação de um objeto remoto [RMI97].

2.5.1 Arquitetura do Sistema RMI

Um sistema RMI está dividido em três camadas: camada *stub/skeleton*; camada de referência remota; e camada de transporte. A divisão entre cada par de camadas é definida por uma interface e protocolos específicos. A arquitetura RMI (figura 2.3) oferece a transparência de localização através da organização de três camadas entre os objetos cliente e servidor.

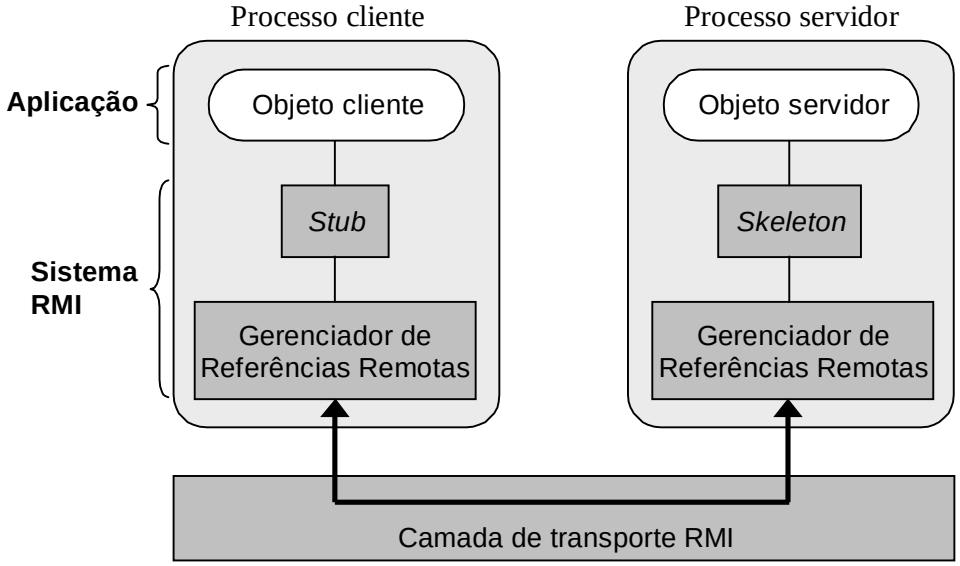

FIGURA 2.3 – Organização das três camadas da arquitetura RMI.

2.5.1.1 Camada *stub/skeleton*

A camada *stub/skeleton* oferece as interfaces que os objetos da aplicação usam para interagir entre si. Ela transmite dados para a camada de referência remota, usando a abstração de *streams* ordenados. Esses *streams* empregam o mecanismo de serialização de objetos para habilitar a transmissão dos objetos entre os diferentes espaços de endereçamento. Os objetos transmitidos usando o processo de serialização de objetos são passados por cópia ao espaço de endereçamento remoto, exceto quando se trata de objetos remotos, onde são passados por referência [BER2000].

Um *stub* para um objeto remoto é um *proxy* do lado do cliente. A camada *stub* é responsável por diversas características como:

- inicialização de uma chamada a um objeto remoto;

- montagem de argumentos para um *stream* ordenado;
- notificação para a camada inferior de que a chamada deve ser realizada;
- desmontagem de um valor de retorno ou exceção de um *stream* ordenado;
- informação à camada de referência que a chamada foi finalizada.

Um *skeleton* para um objeto remoto é uma entidade do lado do servidor, que contém um método que despacha chamadas para a implementação do objeto remoto. A camada *skeleton* é responsável por diversas características como:

- desmontagem de argumentos de um *stream* ordenado;
- criação de uma chamada para a implementação de objeto remoto;
- montagem do valor de retorno da chamada ou exceção em um *stream* ordenado.

2.5.1.2 Camada de referência remota

A camada de referência remota é o *middleware* entre a camada de *stub/skeleton* e o protocolo de transporte. Nesta camada são criadas e gerenciadas as referências remotas aos objetos. Ela interage com a camada de mais baixo nível, a de transporte. É responsável por carregar um protocolo de referência remota específico, que é independente do *stub* do cliente, ou do *skeleton* do servidor.

Cada implementação do objeto remoto escolhe sua própria subclasse de referência remota, que opera no seu comportamento. Diversos protocolos de invocação são carregados nesta camada, tais como *unicast* (ponto-a-ponto), e a invocação para grupos de objetos replicados.

Essa camada possui dois componentes que cooperam: os componentes do lado do servidor e os do lado do cliente. Os últimos contêm informação específica do servidor remoto e comunicam-se via camada de transporte do componente do lado do servidor. Durante cada invocação de método, os componentes do lado cliente e do servidor desenvolvem semânticas específicas de referência remota. Da forma semelhante, o componente do lado do servidor implementa as semânticas de referência remota para enviar uma invocação de um método remoto ao *skeleton* [SUN98].

2.5.1.3 Camada de transporte

A camada do protocolo de transporte oferece o protocolo de dados binários que envia as solicitações aos objetos remotos pela rede. A camada de transporte de um sistema RMI é responsável por diversas tarefas, tais como [BER2000]:

- ajuste de conexões para espaços de endereçamento remoto;
- gerenciamento de conexões em execução;
- espera por chamadas;

- manutenção de uma tabela de objetos remotos, que residem em um espaço de endereçamento;
- configuração de conexão para chamada.

2.5.2 Desenvolvimento da aplicação RMI

No desenvolvimento de uma aplicação cliente-servidor usando Java RMI, como para qualquer plataforma de objetos distribuídos, é essencial que seja definida a interface de serviços que será oferecida pelo objeto servidor. A especificação de uma interface remota é equivalente à definição de qualquer interface em Java, a não ser pelos seguintes detalhes: a interface deverá, direta ou indiretamente, estender a interface **`Remote`**; e todo método da interface deverá declarar que a exceção *RemoteException* (ou uma de suas superclasses) pode ser gerada na execução do método.

Os serviços especificados pela interface RMI deverão ser implementados através de uma classe Java. Nessa implementação dos serviços é preciso indicar que objetos dessa classe poderão ser acessados remotamente. A implementação do serviço dá-se através da definição de uma classe que implementa a interface especificada. No entanto, além de implementar a interface especificada, é preciso incluir as funcionalidades para que um objeto dessa classe possa ser acessado remotamente como um servidor. A implementação da interface remota dá-se da mesma forma que para qualquer classe implementando uma interface Java, ou seja, a classe fornece implementação para cada um dos métodos especificados na interface.

As funcionalidades de um servidor remoto são especificadas na classe abstrata **`RemoteServer`**, do pacote **`java.rmi.server`**. Um objeto servidor RMI deverá estender essa classe ou, mais especificamente, uma de suas subclasses. Uma subclasse concreta de **`RemoteServer`** oferecida no mesmo pacote é **`UnicastRemoteObject`**, que permite representar um objeto que tem uma única implementação em um servidor (ou seja, não é replicado em vários servidores) e mantém uma conexão ponto-a-ponto com cada cliente que o referencia.

Uma vez que a interface remota esteja definida e a classe que implementa o serviço remoto tenha sido criada, o próximo passo no desenvolvimento da aplicação distribuída é desenvolver o servidor RMI, uma classe que crie o objeto que implementa o serviço e cadastre esse serviço na plataforma de objetos distribuídos.

Um objeto servidor RMI simples deve realizar as seguintes tarefas:

- criar uma instância do objeto que implementa o serviço;
- disponibilizar o serviço através do mecanismo de registro.

O desenvolvimento de um cliente RMI requer essencialmente a obtenção de uma referência remota para o objeto que implementa o serviço, o que ocorre através do cadastro realizado pelo servidor. Uma vez obtida essa referência, as operações com objetos remotos ou locais são indistinguíveis.

2.5.3 Definindo *stubs* e *skeletons*

Para que um serviço oferecido por um objeto possa ser acessado remotamente através de RMI, é preciso também as classes auxiliares internas de *stubs* e *skeletons*, responsáveis pela comunicação entre o objeto cliente e o objeto que implementa o serviço, conforme descrito na apresentação da arquitetura RMI (figura 2.3).

Uma classe *stub* oferece implementações dos métodos do serviço remoto que são invocados no lado do cliente. Internamente, esses métodos empacotam os argumentos para o método **`stub`** e os enviam ao servidor. A implementação correspondente no lado servidor, no *skeleton*, desempacota os dados e invoca o método do serviço. Obtido o valor de retorno do serviço, o método **`skeleton`** empacota e envia esse valor para o método **`stub`**, que ainda estava aguardando esse retorno. Obtido o valor de retorno no *stub*, esse é desempacotado e retornado à aplicação cliente como resultado da invocação remota. Os argumentos e valores de retorno de métodos remotos invocados através de RMI são restritos a tipos primitivos de Java e a objetos de classes que implementam **`Serializable`**.

2.5.4 Execução de uma aplicação RMI

Na execução da uma aplicação RMI, pode-se ter vários processos em execução simultânea, conforme ilustrado na figura 2.4.

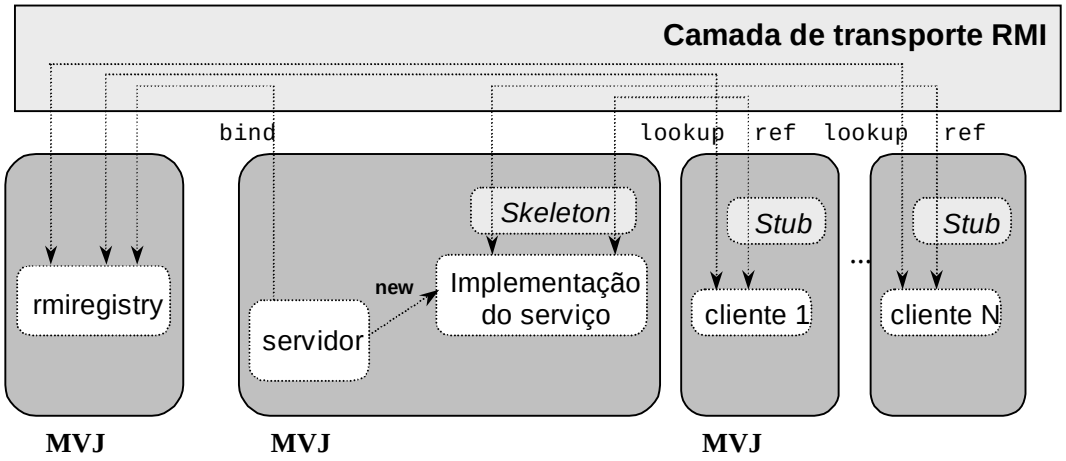

FIGURA 2.4 – Esquema da execução de uma aplicação RMI.

O registro RMI (*rmiregistry*) executa isoladamente em uma máquina virtual Java. O servidor da aplicação e a implementação do serviço são executados em outra máquina virtual Java; sua interação com o registro (ao invocar o método **`bind()`**) dá-se através de uma referência remota. Da mesma forma, cada aplicação cliente pode ser executada em sua própria máquina virtual Java; suas interações com o registro (método **`lookup()`**) e com a implementação do serviço (usando os correspondentes *stub* e *skeleton*) dão-se também através de referências remotas.

Para executar uma aplicação RMI é preciso inicialmente disponibilizar o serviço de registro RMI. Para tanto, o aplicativo *rmiregistry* deve ser executado. Tipicamente, esse aplicativo é executado como um processo de fundo (em *background*) que fica aguardando solicitações em uma porta, que pode ser especificada como argumento na linha de comando. Se nenhum argumento for especificado, a porta 1099 é usada como padrão. O aplicativo *rmiregistry* é uma implementação de um serviço de nomes para RMI. O serviço de nomes é uma espécie de diretório, onde cada serviço disponibilizado na plataforma é registrado através de um nome do serviço, correspondendo um *string* único para cada objeto que implementa serviços em RMI.

Com o *rmiregistry* disponível, o servidor pode ser executado. Para tanto, a máquina virtual Java deverá ser capaz de localizar e carregar as classes do servidor, da implementação do serviço e do *skeleton*. Com o servidor já habilitado para responder às solicitações, o código cliente pode ser executado. Essa máquina virtual deverá ser capaz de localizar e carregar as classes com a aplicação cliente, a interface do serviço e o *stub* para a implementação do serviço.

2.5.5 Coletor de lixo de objetos distribuídos

O processo de remoção de objetos remotamente não-referenciados ocorre de maneira automática. Cada servidor com objetos exportados mantém uma lista de referências remotas aos objetos que ele oferece. Através de comunicação com o cliente, ele é notificado quando a referência é liberada na aplicação remota.

Quando não existem mais referências para um objeto remoto, o sistema RMI indica que existe uma referência inadequada. Isto permite que o coletor de lixo da máquina virtual descarte o objeto. O algoritmo do coletor de lixo distribuído atua em conjunto com o coletor de lixo da máquina virtual Java [RMI97].

3 Características da biblioteca *Libcjp*

A biblioteca *Libcjp – Library of checkpoints in Java programs*, assim nomeada, foi projetada com o propósito de trabalhar principalmente com aplicações computacionais orientadas a objetos escritas na linguagem de programação Java. Foi desenvolvida com funcionalidades para capturar o estado de um ou mais objetos de uma aplicação em execução e representá-los no armazenamento estável para posterior recuperação, na ocorrência de falhas. Esta representação dos estados dos objetos da aplicação foi alcançada utilizando os mecanismos de persistência e serialização (seções 2.3 e 2.4) presentes na linguagem Java através da API de serialização Java [SUN98].

A biblioteca *Libcjp* tem o intuito de auxiliar a tarefa dos programadores para prover tolerância a falhas às aplicações computacionais, no que se refere a propiciar maior disponibilidade para estas aplicações. *Libcjp* é uma biblioteca desenvolvida para ser utilizada em nível de usuário, projetada para uso em situações onde se deseja minimizar o tempo de execução total de uma aplicação, na presença de falhas.

Como exemplo de uma situação de uso, considere-se uma aplicação escrita em Java que levará algum tempo para executar, por hipótese dez horas. Após cinco horas de computação, falha o processador no qual a aplicação está executando. Se o programador não planejou a ocorrência deste evento, a única coisa a ser feita é reiniciar a aplicação e perder cinco horas de trabalho, e ainda será necessário um tempo de dez horas contínuas de processador livre de falhas para completar o trabalho. Usando a biblioteca, existe apenas a necessidade de reiniciar a aplicação e continuar a computação a partir do último *checkpoint* armazenado, restando cerca de cinco horas para completar a execução total da aplicação, dependendo a freqüência dos salvamentos intermediários realizados.

3.1 Uso da biblioteca

A seguir são apresentados os benefícios fornecidos pelo enfoque proposto para a biblioteca *Libcjp*:

- fazendo o uso da biblioteca, consegue-se prover maior disponibilidade para muitas aplicações computacionais orientadas a objetos escritas em Java, com a finalidade de aprimorar o salvamento de estados e a retomada da execução, diante da ocorrência de falhas;
- diferentes formas de utilização da biblioteca poderão ser feitas pelo programador. Em alguns casos, a biblioteca adiciona somente uma pequena sobrecarga de processamento para salvamento dos estados dos objetos sobre o tempo de execução da aplicação. Estes estados são serializados e salvos nos arquivos de *checkpoint*. As diferentes formas de utilização poderão ser vistas na seção 5.2;
- uma das principais vantagens de uma aplicação em Java é a independência de plataforma: um programa Java pode ser executado em qualquer plataforma que possua uma Máquina Virtual Java e deverá apresentar o mesmo comportamento em cada uma destas plataformas;

- os *checkpoints* são independentes de plataforma, ou seja, uma dada aplicação Java poderá ser reiniciada em uma outra Máquina Virtual Java de outra plataforma, sem levar em consideração a plataforma na qual os *checkpoints* foram estabelecidos. Mas, para que isto ocorra, a aplicação em questão não poderá ser desenvolvida utilizando classes que pertençam a apenas uma plataforma específica.

Para prover os benefícios acima mencionados a biblioteca fornece alguns custos para sua utilização, descritos a seguir:

- *checkpointing* com *Libcjp* não é completamente transparente ao programador da aplicação, deverá requerer algum esforço de programação a ser acrescido para poder ser utilizada. Tal esforço poderá ser medido de acordo com a aplicação desenvolvida e de acordo com o uso da biblioteca, mas na maioria dos casos deverá ser pequeno. Diz-se que o *checkpointing* é transparente quando nenhuma modificação precisa ser feita no código-fonte da aplicação;
- o tamanho do código-fonte da aplicação deverá aumentar somente com o acréscimo de algumas linhas de código para especificar à biblioteca os objetos que deverão estar presentes no arquivo de *checkpoint* e também linhas de código para definir os procedimentos de recuperação.

3.2 Funcionalidades propostas para a biblioteca

Como visto anteriormente, a biblioteca *Libcjp* foi desenvolvida com o propósito de efetuar o salvamento dos estados dos objetos previamente à ocorrência de falhas, e de definir a retomada da execução da aplicação, restaurando os estados dos objetos a partir dos arquivos de *checkpoints* salvos. Sendo assim, as principais funcionalidades propostas e implementadas para a biblioteca são os mecanismos de *checkpointing* não incremental e *checkpointing* incremental com ou sem controle de tempo (*checkpointing* programado), mecanismo de recuperação, controle do número de arquivos de *checkpoint* salvos no disco e registros de informações (tempos decorridos para o estabelecimento de cada *checkpoint* e o tempo total de execução da aplicação). As funcionalidades da biblioteca são descritas mais detalhadamente nos próximos parágrafos.

Mecanismo de *checkpointing* não incremental: é o método mais direto para salvar um arquivo de *checkpoint*. A execução da aplicação é suspensa, enquanto os estados dos objetos da aplicação são serializados e salvos no arquivo de *checkpoint*. Este mecanismo também é denominado de *checkpointing* seqüencial, porque transferências da memória (*stream* contendo os estados dos objetos serializados) para o disco (arquivo de *checkpoint*) são intercaladas com a execução da aplicação. Vale ressaltar que apenas serão gravadas no arquivo as informações desejadas pelo programador da aplicação, mediante a utilização do método **`save(Object arg)`** da classe **`Checkpointing`** (subseção 4.2.2).

Mecanismo de *checkpointing* incremental: este mecanismo é semelhante ao descrito anteriormente, principalmente no que se refere à execução seqüencial para salvamento dos estados dos objetos serializados no arquivo de *checkpoint*. A diferença básica é que somente os estados dos objetos modificados desde o *checkpoint* prévio

necessitam ser salvos no arquivo de *checkpoint*. A base do *checkpointing* incremental é não precisar repetidamente salvar no disco os objetos que não foram modificados.

Em geral, o tamanho de um *checkpoint* não incremental cresce muito lentamente com o passar do tempo. Além disso, apenas o arquivo mais recente de *checkpoint* necessita ser mantido para recuperação; os arquivos mais antigos podem ser apagados [PLA95]. Em contraste, quando se emprega o mecanismo de *checkpointing* incremental, arquivos de *checkpoint* antigos não podem ser apagados, porque os estados dos objetos da aplicação são espalhados para muitos arquivos de *checkpoint*. Os estados inalterados dos objetos são restabelecidos a partir de *checkpoints* prévios. Salvando apenas os objetos cujos estados foram modificados, reduz-se o tamanho de cada arquivo de *checkpoint*.

O mecanismo de *checkpointing* incremental é ativado ou desativado através do parâmetro **`incremental`** do arquivo de parâmetros **`paramckpt.dat`** (subseção 4.2.5).

***Checkpointing* programado**: esta é uma técnica que emprega ambos os mecanismos, ou seja, usa o *checkpointing* não incremental e o *checkpointing* incremental. Para utilizar a biblioteca, o programador terá que especificar pontos no código da aplicação onde é vantajoso e ou necessária a ocorrência do *checkpointing*. Conforme os parâmetros **`intervalo`** e **`tipo_intervalo`** do arquivo de parâmetros **`paramckpt.dat`** (subseção 4.2.5), o arquivo de *checkpoint* pode ser estabelecido de duas formas:

- **executado sempre**: ocorre todas as vezes que a linha de execução da aplicação passar por uma chamada ao método **`checkpoint_here()`** – classe **`Checkpointing`** (subseção 4.2.2), o *checkpoint* será salvo. Isto acontece quando o valor do parâmetro **`tipo_intervalo`** do arquivo de parâmetros for *V*, o que desabilita o intervalo mínimo entre um *checkpointing* e outro;
- **inibido por controle de tempo**: neste caso, o *checkpoint* apenas será estabelecido quando um período mínimo de tempo decorrer desde o último *checkpoint*. Este período mínimo de tempo deve ser especificado no parâmetro **`intervalo`** do arquivo de parâmetros. Quando a linha de execução da aplicação passar por uma chamada ao método **`checkpoint_here()`** – classe **`Checkpointing`**, se o intervalo de tempo for insuficiente, esta será ignorada.

Mecanismo de recuperação: a técnica tradicional para recuperação baseia-se na recomposição de um objeto ou vários objetos a partir dos estados serializados de um arquivo de *checkpoint* para um estado operacional normal. Após a ocorrência desta atividade, a execução da aplicação é retomada.

Quando os arquivos de *checkpoint* foram salvos usando o mecanismo de *checkpointing* incremental, ao proceder à atividade de recuperação, haverá a necessidade de realizar a unificação (*merge*) dos arquivos velhos de *checkpoint*. A atividade de unificação consiste em buscar nos arquivos de *checkpoint* os estados mais atuais dos objetos da aplicação. Isto é feito com a leitura de cada arquivo de *checkpoint* prévio, durante a execução da aplicação sem a ocorrência de falhas, até que se obtenha os estados mais atuais dos objetos.

Controle do número de arquivos de *checkpoint*: esta funcionalidade trata de manter no disco apenas a quantidade de arquivos estipulada no parâmetro **`num_max_arquivos`** do arquivo de parâmetros **`paramckpt.dat`** (subseção 4.2.5).

Esta funcionalidade pode ser utilizada tanto para o mecanismo de *checkpointing* incremental como *checkpointing* não incremental. Após *n* arquivos de *checkpoint* terem sido criados, os estados mais atuais dos objetos da aplicação, presentes na memória, são salvos no próximo arquivo de *checkpoint* e os arquivos velhos de *checkpoint* são excluídos.

Registros de informações (tempos): esta funcionalidade tem por objetivo propiciar a obtenção de informações de tempos referentes ao *checkpointing* e à execução da aplicação. Os tempos decorrentes para a atividade de *checkpointing* e o tempo total de execução da aplicação serão salvos em um arquivo-texto específico (subseção 4.2.3).

4 Implementação da biblioteca (Classes)

Este capítulo tem como objetivo a descrição das classes propostas para a implementação das funcionalidades da biblioteca *Libcjp*. Nestas classes estão codificados os mecanismos de *checkpointing* não incremental e *checkpointing* incremental com ou sem controle de tempo, assim como o mecanismo de recuperação. Estas classes foram inicialmente modeladas através da **Linguagem de Modelagem Unificada** (UML – *Unified Modeling Language*) e, após, implementadas utilizando a linguagem orientada a objetos Java (pacote *jdk1.3.1_01*).

O diagrama das classes definidas para a biblioteca é apresentado na seção 4.1; nas seções subseqüentes deste capítulo são feitas descrições do conteúdo de cada uma das classes implementadas. Nestas descrições, são mostrados trechos de cada classe, demonstrando as variáveis declaradas e os métodos implementados para facilitar a compreensão das mesmas.

4.1 Modelagem das classes da biblioteca em UML

No processo de desenvolvimento de sistemas orientados a objetos, é importante ter-se uma notação de especificação padronizada e realmente eficaz nas fases de análise de requisitos, análise de sistemas e de projeto. Especificar significa construir modelos precisos, sem ambigüidades e completos. Em particular, a UML – *Unified Modeling Language* [ERI98] dá suporte às necessidades geradas pela análise, projeto e implementação, que devem ser tomadas para o desenvolvimento e implementação de sistemas complexos de *software*.

A UML é uma linguagem padrão para especificar, visualizar, documentar e construir artefatos de um sistema e pode ser utilizada com todos os processos ao longo do ciclo de desenvolvimento e através de diferentes tecnologias de implementação [FUR98]. A UML é adequada para a modelagem de sistemas, cuja abrangência poderá incluir sistemas de informação corporativos a serem distribuídos a aplicações baseadas na *Web* e até sistemas complexos embutidos de tempo real [BOO2000]. A notação UML passou por um processo de padronização pelo OMG (*Object Management Group*) e se tornou um padrão do OMG [FOW2000].

As classes da biblioteca proposta por esta dissertação foram modeladas tendo por base o padrão proposto pela linguagem UML.

As classes que compõem a biblioteca *Libcjp* são as seguintes: classe **`Checkpointing`** (implementa os mecanismos de *checkpointing* e recuperação); classe **`ArquivoCkpt`** (manipula as características dos arquivos de *checkpoint*); classe **`Tempo`** (efetua controle de tempo para o estabelecimento dos *checkpoints*); classe **`EstatisticaCkpt`** (faz o registro dos tempos de *checkpointing* e de execução da aplicação); interface **`Definicoes`** (contém constantes utilizadas pela biblioteca); classes **`ParametrosCkpt`** e **`InterParamCkpt`** (gerencia e manipula um arquivo de parâmetros).

Primeiramente, na figura 4.1, é mostrado um diagrama contendo um modelo simplificado de classes da biblioteca *Libcjp*. Posteriormente na figura 4.2 é ilustrado o modelo de classes propostas para a construção da biblioteca *Libcjp* de uma forma mais detalhada, mostrando todas as variáveis de instância e métodos utilizados.

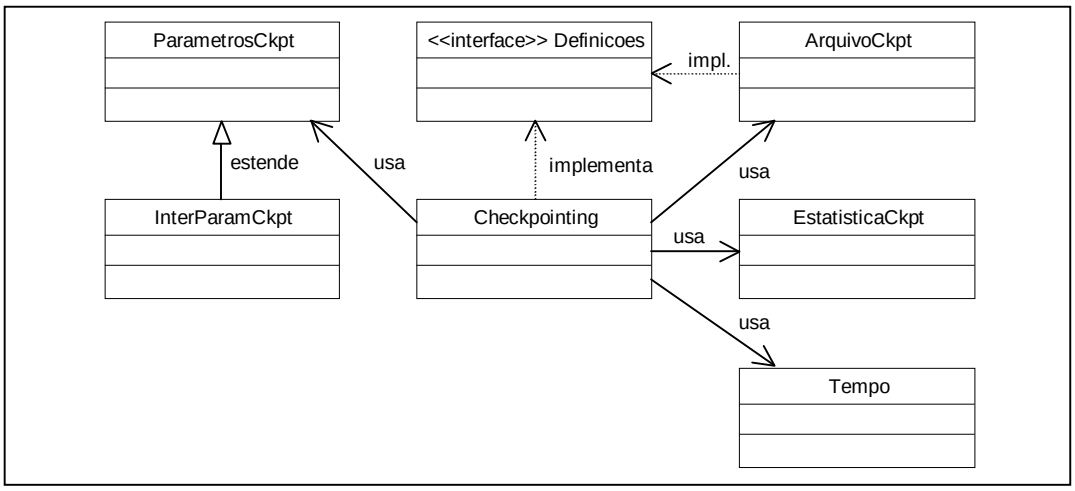

FIGURA 4.1 – Modelo simplificado de classes da biblioteca *Libcjp*.

4.2 Descrição das Classes propostas

Nas próximas seções, são descritas, individualmente, cada uma das classes presentes no diagrama mostrado na figura 4.2. Nesse diagrama, a classe **`Checkpointing`** é a principal, pois nela estão implementados os mecanismos para prover o estabelecimento de *checkpoints* e realizar as ações necessárias à recuperação, além de todo o controle operacional da biblioteca. As classes **`ArquivoCkpt`** e **`Tempo`** dão apoio funcional a classe **`Checkpointing`**. A classe **`ArquivoCkpt`** contém funcionalidades para tratamento dos arquivos de *checkpoint* no que se refere à geração de nomes de novos arquivos, classificação e obtenção do nome do arquivo de *checkpoint* mais recente para ser utilizado na atividade de recuperação. A classe **`Tempo`** tem como tarefa fazer o controle de tempo para o estabelecimento dos *checkpoints*.

A classe **`EstatisticaCkpt`** faz o registro dos tempos gastos para o estabelecimento de cada *checkpoint* e do tempo total de execução da aplicação, em um arquivo-texto cujo nome é especificado pelo programador da aplicação. A interface **`Definicoes`** contém as constantes utilizadas pelas classes **`Checkpointing`** e **`ArquivoCkpt`**.

As classes restantes **`ParametrosCkpt`** e **`InterParamCkpt`** são utilizadas como apoio à biblioteca, provendo um arquivo de parâmetros. Estes parâmetros têm a finalidade de configurar a biblioteca para alguns tipos diferentes de utilização.

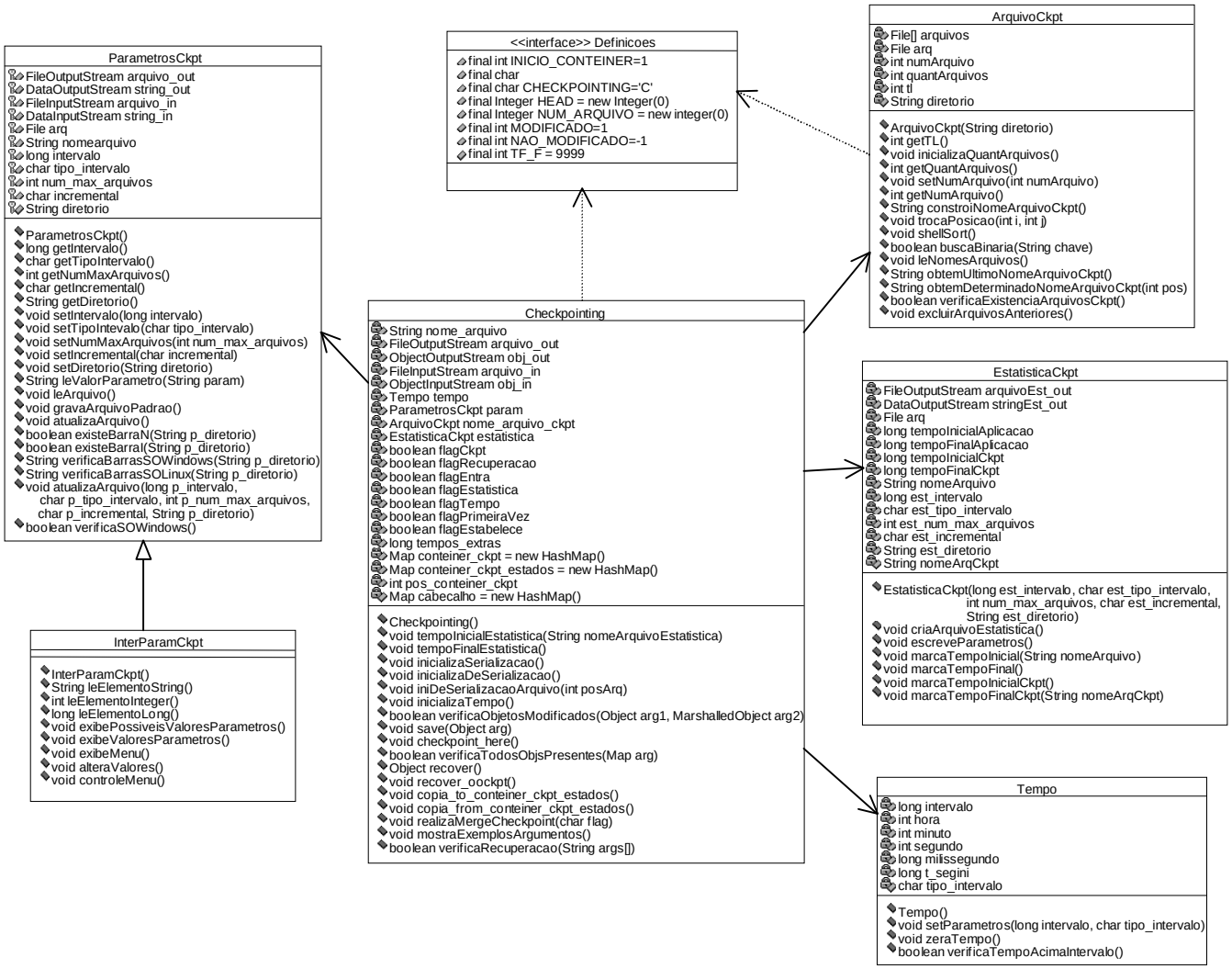

FIGURA 4.2 – Modelo detalhado de classes da biblioteca *Libcjp*.

4.2.1 Classe ArquivoCkpt

A classe **ArquivoCkpt** (figura 4.3) contém funcionalidades para tratamento e manipulação dos arquivos de *checkpoint*, dando apoio à classe **Checkpointing**. A classe armazena uma lista de nomes de arquivos em um vetor de objetos da classe **File** – **arquivos** (linha 3), denominado **arquivos**, referente aos arquivos de *checkpoint* salvos. A classe implementa a interface **Definicoes** (subseção 4.2.7) para ter acesso à constante **TF_F** que é usada como tamanho físico do vetor **arquivos**. Para ler os nomes dos arquivos de *checkpoint* do disco e atribuir ao vetor **arquivos**, é utilizado o método **leNomesArquivos()** (linha 50); o objeto **arq** da classe **File** (linha 4), através do método **arq.listFiles()**, é usado internamente neste método para ler as informações dos arquivos do diretório estipulado pela variável do tipo **String** – **diretorio** (linha 8). A variável **tl** (linha 7) armazena o tamanho lógico do vetor **arquivos**, que representa a quantidade de arquivos de *checkpoint* salvos. A variável **numArquivo** (linha 5) armazena um número seqüencial que é usado para compor o nome do arquivo de *checkpoint*; o método responsável por construir este nome é o método **constroiNomeArquivoCkpt()** (linha 34). A variável **quantArquivos** (linha 6) armazena o número de *checkpoints* salvos durante a execução da aplicação. Isto é importante para controlar o número máximo de arquivos que a aplicação deve manter durante a sua execução. Todas as variáveis descritas são inicializadas no construtor (linha 10); este recebe por parâmetro a variável **diretorio**, que representa o caminho do diretório onde serão salvos os arquivos de *checkpoint*.

O método **verificaExistenciaArquivosCkpt()** (linha 62) é usado na atividade de recuperação, onde o mecanismo de recuperação (subseção 4.2.2) necessita saber se existe arquivos de *checkpoint* para então efetuar a recuperação dos estados da aplicação. O método **obtemUltimoNomeArquivoCkpt()** (linha 54) extrai do vetor **arquivos** o nome do último arquivo de *checkpoint*. Os nomes dos arquivos no vetor **arquivos** são classificados em uma ordem crescente, com o intuito de facilitar a obtenção do nome do último arquivo de *checkpoint*, para isto existe o método **shellSort()** (linha 42).

Durante a atividade de *checkpointing* (subseção 4.2.2), quando a variável **quantArquivos** for igual ao valor do parâmetro **num_max_arquivos** do arquivo de parâmetros **paramckpt.dat** (subseção 4.2.5), o próximo *checkpoint* a ser estabelecido será composto pelos estados mais atuais dos objetos da aplicação, e os arquivos antigos de *checkpoint* serão descartados. Para efetuar a atividade de exclusão dos arquivos após o *checkpointing*, é invocado o método **excluirArquivosAnteriores()** (linha 66).

Quando os *checkpoints* forem estabelecidos através do mecanismo de *checkpointing* incremental, na fase de recuperação, haverá a necessidade de ser realizada a unificação dos arquivos em busca dos estados mais atuais dos objetos da aplicação. Na atividade de unificação, cada arquivo de *checkpoint* é visitado (os objetos contidos no arquivo de *checkpoint* são de-serializados para que se possa obter seus estados). Para obter o nome de um determinado arquivo é usado o método **obtemDeterminadoNomeArquivoCkpt(int pos)** (linha 58), onde **pos** representa a posição desejada no vetor **arquivos**.

```
 1 public class ArquivoCkpt implements Definicoes
 2 {
 3  private File[] arquivos;
 4  private File arq;
 5  private int numArquivo;
 6  private int quantArquivos;
 7  private int tl;
 8  private String diretorio;
 9
10  public ArquivoCkpt(String diretorio)
11  { ...
12  }
13
14  public int getTL()
15  { ...
16  }
17
18  public void inicializaQuantArquivos()
19  { ...
20  }
21
22  public int getQuantArquivos()
23  { ...
24  }
25
26  public void setNumArquivo(int numArquivo)
27  { ...
28  }
29
30  public int getNumArquivo()
31  { ...
32  }
33
34  public String constroiNomeArquivoCkpt()
35  { ...
36  }
37
38  public void trocaPosicao(int i, int j)
39  { ...
40  }
41
42  public void shellSort()
43  { ...
44  }
45
46  public boolean buscaBinaria(String chave)
47  { ...
48  }
49
50  public void leNomesArquivos()
51  { ...
52  }
53
54  public String obtemUltimoNomeArquivoCkpt()
55  { ...
56  }
57
58  public String obtemDeterminadoNomeArquivoCkpt(int pos)
59  { ...
60  }
```

FIGURA 4.3 – Classe **`ArquivoCkpt`**.

```
61
62  public boolean verificaExistenciaArquivosCkpt()
63  { ...
64  }
65
66  public void excluirArquivosAnteriores()
67  { ...
68  }
69 }
```

FIGURA 4.3 – Classe **`ArquivoCkpt`** (continuação).

4.2.2 Classe *Checkpointing*

Nesta subseção, é apresentada a descrição da classe **`Checkpointing`**, suas variáveis de instância, métodos e funcionalidades. Para melhor compreensão desta classe, ao final da presente subseção são apresentados diagramas contendo as trocas de mensagens realizadas entre os métodos dos objetos que dão apoio funcional.

A classe **`Checkpointing`** (figura 4.4) é a principal, pois implementa os mecanismos para prover estabelecimento de *checkpoints* e recuperação. Todo o controle operacional da biblioteca também está implementado nesta classe.

Para realizar a gravação do arquivo de *checkpoint*, a classe possui um objeto da classe **`ObjectOutputStream`** – **`obj_out`** (linha 5) para serializar os estados dos objetos da aplicação para um *stream* (seqüência de *bytes*) e um objeto da classe **`FileOutputStream`** – **`arquivo_out`** (linha 4) que grava a seqüência de *bytes* para um arquivo de *checkpoint*. Para fazer a leitura do arquivo de *checkpoint*, foram definidos um objeto da classe **`FileInputStream`** – **`arquivo_in`** (linha 6) e um objeto da classe **`ObjectInputStream`** – **`obj_in`** (linha 7) para realizar a de-serialização. O campo **`nome_arquivo`** (linha 3) armazena o nome do último arquivo de *checkpoint*, o mais atual.

O objeto **`tempo`** (linha 8), instanciado da classe **`Tempo`** (subseção 4.2.6), fornece um serviço de controle de tempo para o estabelecimento dos *checkpoints*. O objeto **`param`** (linha 9), instanciado da classe **`ParametrosCkpt`** (subseção 4.2.5), fornece acesso aos valores dos parâmetros salvos no arquivo de parâmetros **`paramckpt.dat`** para os métodos implementados da classe **`Checkpointing`**. O objeto **`nome_arquivo_ckpt`** (linha 10), instanciado da classe **`ArquivoCkpt`** (subseção 4.2.1), fornece métodos com funcionalidades para o tratamento dos arquivos de *checkpoint*. O objeto **`estatistica`** (linha 11), instanciado da classe **`EstatisticaCkpt`** (subseção 4.2.3), fornece um serviço de registro dos tempos decorridos para o *checkpointing* e o tempo total da execução da aplicação.

Para alguns controles de execução da biblioteca, a classe possui algumas variáveis. A variável **`flagCkpt`** (linha 13), quando possuir valor **`false`**, indicará que durante a execução de uma aplicação nenhum *checkpoint* foi estabelecido. A variável **`flagRecuperacao`** (linha 14), quando possuir valor **`true`**, indicará que a recuperação não iniciará a partir do arquivo de *checkpoint* mais atual, e sim de um arquivo de *checkpoint* específico. O nome deste arquivo de *checkpoint* específico é fornecido como argumento na linha de comando de execução da aplicação. A variável **`flagEntra`**

(linha 15) indica ao mecanismo de *checkpointing* que este deverá invocar o método **`inicializaSerializacao()`** (linha 42), pois não se trata mais do primeiro arquivo de *checkpoint* a ser salvo. A variável **`flagEstatistica`** (linha 16) indica para a biblioteca que o programador está usando o mecanismo de registro de informações de tempos. A variável **`flagTempo`** (linha 17), quando possuir valor **`true`**, indicará ao mecanismo de *checkpointing* que este deverá estabelecer o arquivo de *checkpoint* sem restrições de tempo, pois trata-se do primeiro. A variável **`flagPrimeiraVez`** (linha 18) é usada para indicar que, no primeiro *checkpoint* após ser efetuada a recuperação, a variável **`pos_conteiner_ckpt`** (linha 25) deve ser inicializada.

Um arquivo de *checkpoint* é composto por uma representação dos estados dos objetos da aplicação num dado instante da execução da aplicação. Os objetos são armazenados no disco segundo o conceito de persistência (seção 2.3). Os objetos são convertidos para uma forma serializada (seqüência de *bytes*) e então salvos no arquivo de *checkpoint*.

A estrutura física do arquivo de *checkpoint* é composta por um objeto contêiner (onde os estados dos objetos da aplicação são reunidos) e um cabeçalho que também é um objeto contêiner.

Os objetos que compõem o estado da aplicação são reunidos em um contêiner denominado de **`conteiner_ckpt`** (linha 23), declarado da classe **`Map`** e instanciado da classe **`HashMap`** antes de serem serializados, e salvos no arquivo de *checkpoint*. O contêiner **`conteiner_ckpt_estados`** (linha 24) armazena, durante a execução da aplicação, os estados mais atuais dos objetos numa forma serializada para posteriores comparações com outros estados dos objetos.

Um arquivo de *checkpoint* também é composto por um cabeçalho que indica quais objetos estão presentes, e no caso do *checkpointing* incremental, também indica quais objetos não foram modificados com relação aos estados mais atuais da aplicação, por isso não serão salvos no arquivo de *checkpoint*. O cabeçalho **`cabecalho`** (linha 27) também é um contêiner, e além das informações dos objetos que foram modificados ou não modificados, armazena também o número que compõe o nome do arquivo de *checkpoint*. A variável **`pos_conteiner_ckpt`** (linha 25) indica a posição chave (*key*) que cada objeto da aplicação irá ocupar no contêiner, também indica a posição chave no cabeçalho, onde será gravada a informação de objeto modificado ou não modificado.

O construtor da classe (linha 29) inicializa as variáveis de controle de execução da biblioteca assim como instancia os objetos que fornecem alguns serviços de apoio.

O método **`tempoInicialEstatistica(...)`** (linha 33) é invocado na aplicação do usuário, recebe por parâmetro o nome do arquivo para registrar as informações de tempo e invoca o método **`marcaTempoInicial(...)`** da classe **`EstatisticaCkpt`** passando por parâmetro a variável **`nomeArquivoEstatistica`**. O método **`tempoFinalEstatistica()`** (linha 38) também é invocado na aplicação e internamente invoca o método **`marcaTempoFinal()`** da classe **`EstatisticaCkpt`**, que calcula e registra o tempo total de execução da aplicação.

Criar um arquivo de *checkpoint* no disco é tarefa do método **`inicializaSerializacao()`** (linha 42). O nome do arquivo de *checkpoint* é gerado invocando o método **`constroiNomeArquivoCkpt()`** da classe **`ArquivoCkpt`**. O método **`inicializaDeSerializacao()`** (linha 46) faz o

processo inverso, lê o nome do último arquivo de *checkpoint* através do método **`obtemUltimoNomeArquivoCkpt()`** da classe **`ArquivoCkpt`** e abre o arquivo para efetuar a recuperação. O método **`iniDeSerializacaoArquivo(int posArq)`** (linha 50) é invocado pelo mecanismo de unificação dos arquivos de *checkpoint* para poder ler determinados arquivos (de-serializar).

O método **`inicializaTempo()`** (linha 54) obtém valores do arquivo de parâmetros **`paramckpt.dat`** e estabelece os parâmetros do controle de tempo ao objeto **`tempo`** – classe **`Tempo`** – através da invocação do método **`setParametros(...)`**.

O método **`save(Object arg)`** (linha 63) é invocado na aplicação do usuário para enviar por parâmetro os objetos da aplicação, cujos estados serão serializados no arquivo de *checkpoint*. Quando o mecanismo de *checkpointing* incremental estiver sendo utilizado, este método é responsável por verificar se os estados dos objetos foram modificados com relação aos estados anteriores, um de cada vez. Para comparar os estados dos objetos é utilizado o método **`verificaObjetosModificados(...)`** (linha 58), onde é comparado o estado do objeto **`arg`** com o estado anterior do mesmo objeto armazenado no contêiner **`conteiner_ckpt_estados`** numa forma serializada.

O mecanismo de *checkpointing* está implementado no método **`checkpoint_here()`** (linha 67). O método **`recover_oockpt()`** (linha 79) implementa o mecanismo de recuperação; quando o mecanismo de *checkpointing* incremental estiver sendo utilizado, o mecanismo de unificação dos arquivos velhos de *checkpoint* será invocado por este método. Após a invocação do método **`recover_oockpt()`** na aplicação, o método **`recover()`** (linha 75) deverá ser invocado também pela aplicação para obter os estados mais atuais dos objetos, recuperados do arquivo de *checkpoint* mais recente.

O método **`verificaTodosObjsPresentes(Map arg)`** (linha 71) é utilizado pelo mecanismo de unificação dos arquivos velhos de *checkpoint*. Este método analisa o cabeçalho do contêiner **`conteiner_ckpt`** para verificar se todos os objetos estão presentes neste contêiner.

O método **`copia_to_conteiner_ckpt_estados()`** (linha 83) copia os objetos que foram modificados desde o último *checkpoint* do contêiner **`conteiner_ckpt`** para o contêiner **`conteiner_ckpt_estados`**. No mecanismo de unificação dos arquivos de *checkpoint*, é utilizado o método **`copia_from_conteiner_ckpt_estados()`** (linha 87), que faz a cópia do contêiner **`conteiner_ckpt_estados`** para o contêiner **`conteiner_ckpt`** após ter reunido os estados mais atuais dos objetos da aplicação, a partir dos arquivos velhos de *checkpoint*.

Quando é feita a recuperação, se os arquivos de *checkpoint* foram salvos seguindo a filosofia do mecanismo de *checkpointing* incremental, o método **`realizaMergeCheckpoint(char flag)`** (linha 91) é invocado para executar o mecanismo de unificação dos arquivos velhos de *checkpoint*, com o valor do parâmetro **`flag`** sendo a constante **`RECUPERACAO`** (subseção 4.2.7).

Para prover a recuperação, o usuário deverá acrescentar na linha de comando de execução da aplicação, no *prompt* do sistema operacional, o argumento "**`recover`**", ou ainda especificar um arquivo para realizar a recuperação acrescentando o argumento

"**recover –a checkpoint_????.ckp**", onde **checkpoint_????.ckp** indica o nome do arquivo para efetuar a recuperação. Verificar se a linha de comando de execução da aplicação está correta ou se a aplicação é para restabelecer a sua execução a partir dos arquivos de *checkpoints* (ativação do mecanismo de recuperação), é tarefa do método **verificaRecuperacao(String args[])** (linha 99), que recebe por parâmetros os argumentos de linha de comando de execução da aplicação através da variável **args[]**. Quando os argumentos estão incorretos, a execução da aplicação será interrompida e o método **mostraExemplosArgumentos()** (linha 95) será invocado para mostrar na tela do usuário uma mensagem com um exemplo dos argumentos válidos na recuperação, conforme mostrado na figura 4.5.

```
 1 public class Checkpointing implements Definicoes
 2 {
 3  private transient String nome_arquivo;
 4  private transient FileOutputStream arquivo_out;
 5  private transient ObjectOutputStream obj_out;
 6  private transient FileInputStream arquivo_in;
 7  private transient ObjectInputStream obj_in;
 8  private transient Tempo tempo;
 9  private transient ParametrosCkpt param;
10  private transient ArquivoCkpt nome_arquivo_ckpt;
11  private transient EstatisticaCkpt estatistica;
12
13  private transient boolean flagCkpt;
14  private transient boolean flagRecuperacao;
15  private transient boolean flagEntra;
16  private transient boolean flagEstatistica;
17  private transient boolean flagTempo;
18  private transient boolean flagPrimeiraVez;
19  private transient boolean flagEstabelece;
20
21  private transient long tempos_extras;
22
23  private Map conteiner_ckpt = new HashMap();
24  private Map conteiner_ckpt_estados = new HashMap();
25  private transient int pos_conteiner_ckpt;
26
27  private Map cabecalho = new HashMap();
28
29  public Checkpointing()
30  { ...
31  }
32
33  public void tempoInicialEstatistica(
34                                  String nomeArquivoEstatistica)
35  { ...
36  }
37
38  public void tempoFinalEstatistica()
39  { ...
40  }
41
42  public void inicializaSerializacao()
43  { ...
44  }
45
46  public void inicializaDeSerializacao()
```

FIGURA 4.4 – Classe **Checkpointing**.

```
 47  { ...
 48  }
 49
 50  public void iniDeSerializacaoArquivo(int posArq)
 51  { ...
 52  }
 53
 54  public void inicializaTempo()
 55  { ...
 56  }
 57
 58  public boolean verificaObjetosModificados(Object arg1,
 59                                             MarshalledObject arg2)
 60  { ...
 61  }
 62
 63  public void save(Object arg)
 64  { ...
 65  }
 66
 67  public void checkpoint_here()
 68  { ...
 69  }
 70
 71  public boolean verificaTodosObjsPresentes(Map arg)
 72  { ...
 73  }
 74
 75  public Object recover()
 76  { ...
 77  }
 78
 79  public void recover_oockpt()
 80  { ...
 81  }
 82
 83  public void copia_to_conteiner_ckpt_estados()
 84  { ...
 85  }
 86
 87  public void copia_from_conteiner_ckpt_estados()
 88  { ...
 89  }
 90
 91  public void realizaMergeCheckpoint(char flag)
 92  { ...
 93  }
 94
 95  public void mostraExemplosArgumentos()
 96  { ...
 97  }
 98
 99  public boolean verificaRecuperacao(String args[])
100  { ...
101  }
102 }
```

FIGURA 4.4 – Classe **`Checkpointing`** (continuação).

```
+-----------------------------------------------------------+
| Argumentos validos na recuperacao                         |
|  a) java <<Aplicacao>> recover                            |
|  b) java <<Aplicacao>> recover -a checkpoint_????.ckp     |
+-----------------------------------------------------------+
```

FIGURA 4.5 – Argumentos válidos na recuperação.

Nesta subseção foi apresentada a descrição da classe **`Checkpointing`**. Como visto, esta classe centraliza todo o controle operacional da biblioteca, o que a torna um pouco complexa. Para facilitar o entendimento das trocas de mensagens que ocorrem com os objetos que dão apoio funcional a esta classe, recorreu-se novamente a UML para buscar uma melhor forma de representar esta questão. Uma alternativa adotada para isto foi utilizar os diagramas de interação.

Diagramas de interação são modelos que descrevem como grupos de objetos colaboram em algum comportamento. Tipicamente, um diagrama de interação captura o comportamento de um único **caso de uso**. Um caso de uso é um conjunto de cenários[6] amarrados por um objetivo comum de um usuário. O diagrama pode mostrar vários objetos e as mensagens que são passadas entre estes objetos em um caso de uso [FOW2000]. Existem dois tipos de diagramas de interação: diagramas de seqüência (dá maior ênfase à seqüência, facilita a observação da ordem de como as coisas acontecem) e diagramas de colaboração (fazem o uso de *layout* para indicar como os objetos estão estaticamente conectados).

Uma das coisas mais difíceis de compreender em um programa orientado a objetos é o fluxo global de controle. Um bom projeto pode apresentar muitos métodos em classes diferentes, e, às vezes, pode ser difícil entender a seqüência global do comportamento pretendido. É neste sentido que foi usado o diagrama de seqüência na classe **`Checkpointing`** para procurar expressar melhor o entendimento desta. A figura 4.6 mostra o diagrama de seqüência para representar o mecanismo de *checkpointing*. Este diagrama mostra a interação de uma aplicação hipotética com a classe **`Checkpointing`** e as principais interações desta classe com as outras classes que compõem a biblioteca para o salvamento do arquivo de *checkpoint.*

[6] Cenário é uma seqüência de passos que descreve uma interação entre um usuário e um sistema [FOW2000].

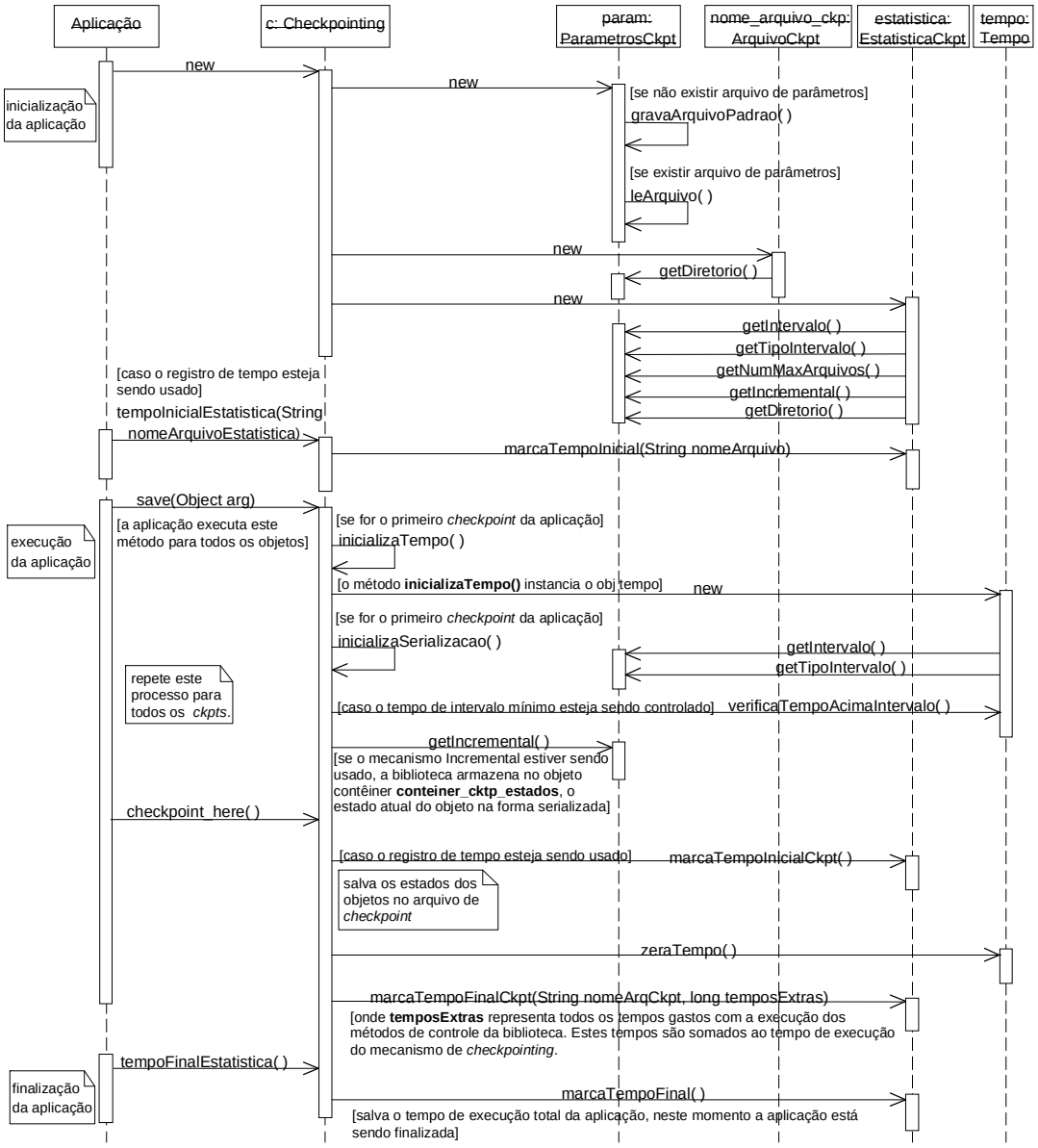

FIGURA 4.6 – Diagrama de seqüência da classe **`Checkpointing`** representando o mecanismo de *checkpointing*.

A figura 4.7 mostra o diagrama de seqüência para representar o mecanismo de recuperação. Este diagrama mostra a interação de uma aplicação hipotética com a classe **`Checkpointing`** e as principais interações desta classe com as outras classes que compõem a biblioteca para a recuperação dos estados dos objetos e a retomada da execução da aplicação.

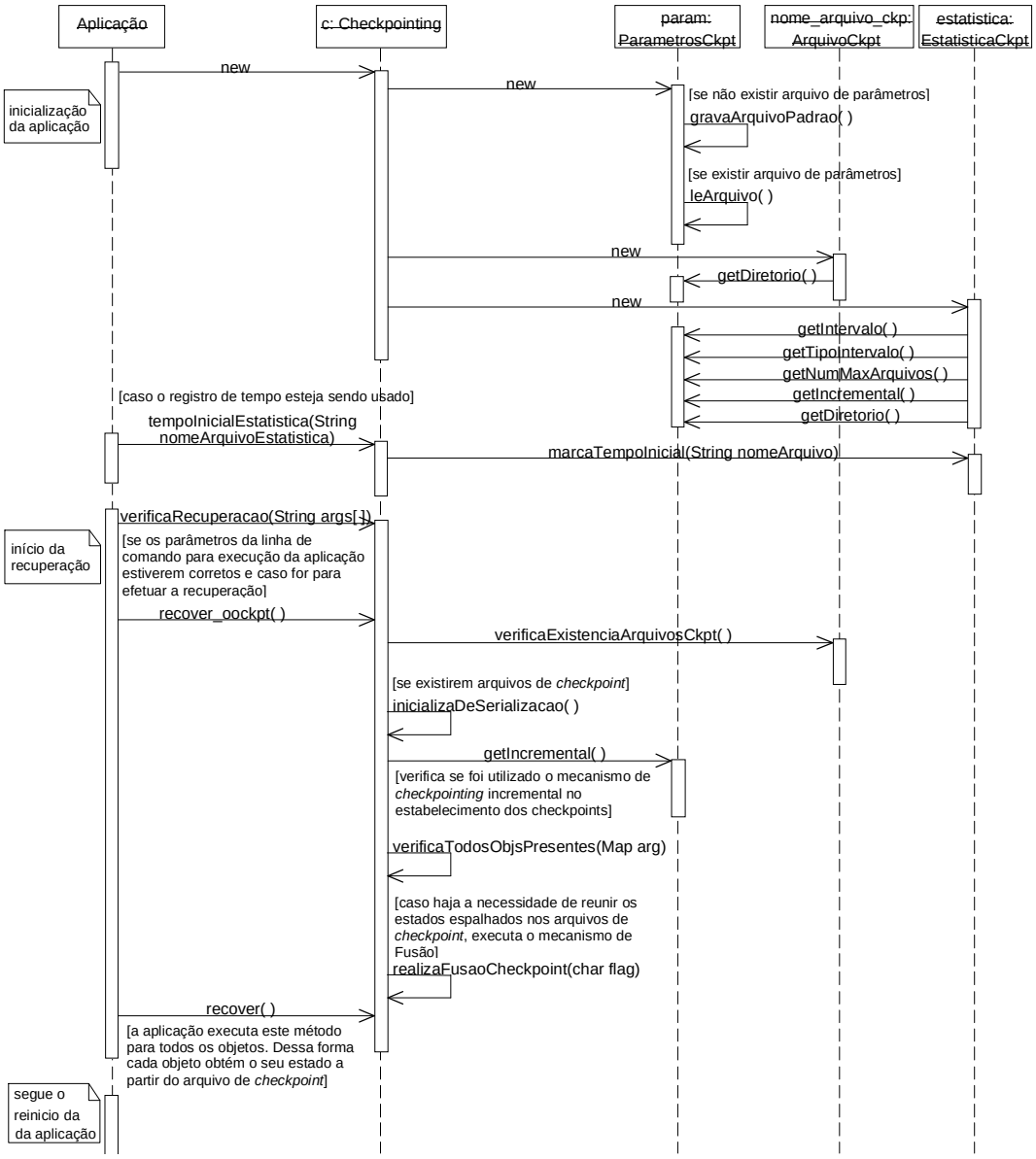

FIGURA 4.7 – Diagrama de seqüência da classe **`Checkpointing`** representando o mecanismo de recuperação.

4.2.3 Classe EstatisticaCkpt

A classe **`EstatisticaCkpt`** (figura 4.8) é utilizada para registrar os tempos decorridos no estabelecimento de cada *checkpoint*, assim como registrar o tempo total de execução da aplicação. Este sistema de controle de tempos, implementado pela classe, apenas calcula diferenças de relógio entre atividades referentes ao *checkpointing* e execução total da aplicação. Estes valores de tempos são escritos em um arquivo-texto juntamente o nome de cada *checkpoint* salvo, assim como a configuração no momento da execução dos parâmetros do arquivo **`paramckpt.dat`**. A escrita dos parâmetros

neste arquivo é de suma importância quando se deseja fazer uma análise estatística de utilização da biblioteca com as diferentes possibilidades de utilização que podem ser configuradas no arquivo de parâmetros, onde a aplicação é executada várias vezes. O nome do arquivo a ser gravado no diretório corrente da aplicação é escolhido pelo programador.

A classe possui um objeto para conectar um *stream* (seqüência de *bytes*) de saída **`stringEst_out`** (linha 4) a um objeto **`arquivoEst_out`** (linha 3) que faz acesso de escrita ao arquivo. O objeto **`arq`** (linha 5) faz referência ao arquivo em disco dado pela variável **`nomeArquivo`** (linha 10).

A classe armazena informações relativas ao tempo de execução da aplicação nas variáveis: **`tempoInicialAplicacao`** (linha 6) e **`tempoFinalAplicacao`** (linha 7). Os métodos para a atividade de registro do tempo de execução da aplicação são: **`marcaTempoInicial(String nomeArquivo)`** (linha 32), que além de registrar o tempo inicial da aplicação, recebe por parâmetro a variável **`nomeArquivo`** contendo o nome do arquivo-texto que será criado; e **`marcaTempoFinal()`** (linha 36). Com relação ao tempo decorrido para o estabelecimento de cada *checkpoint*, foram definidas as variáveis: **`tempoInicialCkpt`** (linha 8) e **`tempoFinalCkpt`** (linha 9). O método **`marcaTempoInicialCkpt()`** (linha 40), escreve na variável **`tempoInicialCkpt`** a hora atual do sistema em milissegundos correspondente ao início da atividade de *checkpointing*. Para registrar os nomes de cada *checkpoint* e seus respectivos tempos de execução, foi criado o método **`marcaTempoFinalCkpt(String nomeArqCkpt)`** (linha 44), que grava o tempo decorrido, associando-o ao nome do *checkpoint* respectivo.

As variáveis **`est_intervalo`** (linha 11), **`est_tipo_intervalo`** (linha 12), **`est_num_max_arquivos`** (linha 13), **`est_incremental`** (linha 14) e **`est_diretorio`** (linha 15) armazenam os valores correspondentes aos valores dos parâmetros do arquivo de parâmetros. As variáveis recebem estes valores via parâmetro pelo construtor da classe (linha 18).

Criar o arquivo de estatística no disco é atividade do método **`criaArquivoEstatistica()`** (linha 24). O método **`escreveParametros()`** (linha 28) registra, no arquivo-texto, os valores provenientes das variáveis que representam os dados do arquivo de parâmetros **`paramckpt.dat`**.

```
 1 public class EstatisticaCkpt
 2 {
 3  private FileOutputStream arquivoEst_out;
 4  private DataOutputStream stringEst_out;
 5  private File arq;
 6  private long tempoInicialAplicacao;
 7  private long tempoFinalAplicacao;
 8  private long tempoInicialCkpt;
 9  private long tempoFinalCkpt;
10  private String nomeArquivo;
11  private long est_intervalo;
12  private char est_tipo_intervalo;
13  private int est_num_max_arquivos;
14  private char est_incremental;
15  private String est_diretorio;
16  private String nomeArqCkpt;
```

FIGURA 4.8 – Classe **`EstatisticaCkpt`**.

```
17
18  public EstatisticaCkpt(long est_intervalo,
19                  char est_tipo_intervalo, int num_max_arquivos,
20                  char est_incremental, String est_diretorio)
21  { ...
22  }
23
24  public void criaArquivoEstatistica()
25  { ...
26  }
27
28  public void escreveParametros()
29  { ...
30  }
31
32  public void marcaTempoInicial(String nomeArquivo)
33  { ...
34  }
35
36  public void marcaTempoFinal()
37  { ...
38  }
39
40  public void marcaTempoInicialCkpt()
41  { ...
42  }
43
44  public void marcaTempoFinalCkpt(String nomeArqCkpt,
45                                                  long temposExtras)
46  { ...
47  }
48 }
```

FIGURA 4.8 – Classe **`EstatisticaCkpt`** (continuação).

A figura 4.9 ilustra um exemplo do arquivo de registro de informações (tempos) provido por esta classe.

```
 [parametros do arquivo "paramckpt.dat"]
 intervalo=20
 tipo_intervalo=L
 num_max_arquivos=0
 incremental=N
 diretorio=.

 [tempo de cada checkpointing]
 checkpoint_0001.ckp -> tempo = 510
 checkpoint_0002.ckp -> tempo = 502
 checkpoint_0003.ckp -> tempo = 442
 checkpoint_0004.ckp -> tempo = 436
 checkpoint_0005.ckp -> tempo = 490
 checkpoint_0006.ckp -> tempo = 495
 checkpoint_0007.ckp -> tempo = 391
 checkpoint_0008.ckp -> tempo = 560

 [execucao da aplicacao]
 tempo execucao total = 5630
 -> tempos em milissegundos
```

FIGURA 4.9 – Arquivo exemplo de registro de informações (tempos).

4.2.4 Classe InterParamCkpt

A classe **`InterParamCkpt`** (figura 4.10) estende a classe **`ParametrosCkpt`** (subseção 4.2.5), e fornece ao programador uma maneira mais fácil de configurar o arquivo de parâmetros **`paramckpt.dat`**. Como a classe herda as propriedades de **`ParametrosCkpt`**, o construtor (linha 3) invoca o método **`leArquivo()`** (classe – **`ParametrosCkpt`**) para fazer a leitura do arquivo de parâmetros e atribuição dos valores de cada parâmetro às variáveis correspondentes.

Os métodos **`leElementoString()`** (linha 7), **`leElementoInteger()`** (linha 11) e **`leElementoLong()`** (linha 15) são invocados no método **`alteraValores()`** (linha 31), para permitir ao programador ler os novos valores dos parâmetros que serão gravados no arquivo. Exibir os valores atuais dos parâmetros é tarefa do método **`exibeValoresParametros()`** (linha 23). Uma ajuda também pode ser requisitada com relação aos possíveis valores dos parâmetros, através do método **`exibePossiveisValoresParametros()`** (linha 19).

A interface de tela para o programador configurar o arquivo de parâmetros fica a cargo do método **`controleMenu()`** (linha 35) invocado pelo método principal **`main`** (linha 39).

```
 1 public class InterParamCkpt extends ParametrosCkpt
 2 {
 3  public InterParamCkpt()
 4  { ...
 5  }
 6
 7  public String leElementoString()
 8  { ...
 9  }
10
11  public int leElementoInteger()
12  { ...
13  }
14
15  public long leElementoLong()
16  { ...
17  }
18
19  public void exibePossiveisValoresParametros()
20  { ...
21  }
22
23  public void exibeValoresParametros()
24  { ...
25  }
26
27  public void exibeMenu()
28  { ...
29  }
30
31  public void alteraValores()
32  { ...
33  }
```

FIGURA 4.10 – Classe **`InterParamCkpt`**.

```
34
35  public void controleMenu()
36  { ...
37  }
38
39  public static void main(String args[])
40  { ...
41  }
42 }
```

FIGURA 4.10 – Classe **InterParamCkpt** (continuação).

4.2.5 Classe ParametrosCkpt

A classe **ParametrosCkpt** (figura 4.11) é utilizada como apoio à biblioteca *Libcjp*, provendo um arquivo de parâmetros para configuração. Essa configuração tem a finalidade de propiciar ao programador alguns tipos de utilização da biblioteca, como por exemplo: definir o intervalo mínimo entre *checkpointings* sucessivos, especificar um número máximo de arquivos de *checkpoint* que devem ser salvos no diretório escolhido pelo programador e ativar ou desativar o mecanismo de *checkpointing* incremental. Seus métodos são explicados nos próximos parágrafos.

Para fazer escrita dos valores dos parâmetros em um arquivo, a classe **ParametrosCkpt** possui um objeto para conectar um *stream* (seqüência de *bytes*) de saída **string_out** (linha 4) a um objeto **arquivo_out** (linha 3) que faz acesso de escrita ao arquivo. Para fazer leitura dos valores, foi definido um objeto para conectar um *stream* de leitura **string_in** (linha 6) a um objeto **arquivo_in** (linha 5) que faz acesso de leitura ao arquivo. O objeto **arq** (linha 7) faz referência ao arquivo em disco dado pela variável **nomearquivo** (linha 8).

Para armazenar os valores dos parâmetros, a classe possui as seguintes variáveis correspondentes, presentes no arquivo de parâmetros: **intervalo** (linha 10), **tipo_intervalo** (linha 11), **num_max_arquivos** (linha 12), **incremental** (linha 13), **diretorio** (linha 14), os quais serão descritos mais adiante nesta mesma subseção.

O construtor (linha 16) verifica se o arquivo de parâmetros existe; caso não exista, invoca o método **gravaArquivoPadrao()** (linha 68) para criar o arquivo de parâmetros com os valores-padrão. A leitura do arquivo e atribuição dos valores existentes de cada parâmetro às variáveis da classe é tarefa do método **leArquivo()** (linha 64).

O método **atualizaArquivo()** (linha 72) é invocado para inserir os valores dos parâmetros em vigor no arquivo. Já o método **atualizaArquivo(...)** (linha 92) é invocado pelo método **alteraValores()** da classe **InterParamCkpt** (subseção 4.2.4), onde recebe, por passagem de parâmetros, os valores para atualizar as variáveis previstas na classe e em seguida invoca o método **atualizaArquivo()** (linha 72) para corrigir os valores presentes no arquivo de parâmetros.

O método **verificaSOWindows()** (linha 98) verifica qual sistema operacional está em uso; caso for o *Windows 9.x*, o método retorna um valor **true**, caso contrário, retorna **false** (neste caso, o sistema operacional em uso poderá ser o *Linux*). Os

métodos: **existeBarraN(String p_diretorio)** (linha 76), **existeBarraI(String p_diretorio)** (linha 80), **verificaBarrasSOWindows(String p_diretorio)** (linha 84), **verificaBarrasSOLinux(String p_diretorio)** (linha 88) dão apoio à classe, para que a variável **diretorio** sempre esteja correta com relação ao caminho (*path*) que for atribuído a ela, esteja o programador usando o sistema operacional *Windows* ou *Linux*.

```
 1 public class ParametrosCkpt
 2 {
 3  protected FileOutputStream arquivo_out;
 4  protected DataOutputStream string_out;
 5  protected FileInputStream arquivo_in;
 6  protected DataInputStream string_in;
 7  protected File arq;
 8  protected String nomearquivo;
 9  //parametros
10  protected long intervalo;
11  protected char tipo_intervalo;
12  protected int num_max_arquivos;
13  protected char incremental;
14  protected String diretorio;
15
16  public ParametrosCkpt()
17  { ...
18  }
19
20  public long getIntervalo()
21  { ...
22  }
23
24  public char getTipoIntervalo()
25  { ...
26  }
27
28  public int getNumMaxArquivos()
29  { ...
30  }
31
32  public char getIncremental()
33  { ...
34  }
35
36  public String getDiretorio()
37  { ...
38  }
39
40  public void setIntervalo(long intervalo)
41  { ...
42  }
43
44  public void setTipoIntevalo(char tipo_intervalo)
45  { ...
46  }
47
48  public void setNumMaxArquivos(int num_max_arquivos)
49  { ...
50  }
```

FIGURA 4.11 – Classe **ParametrosCkpt**.

```
 51
 52  public void setIncremental(char incremental)
 53  { ...
 54  }
 55
 56  public void setDiretorio(String diretorio)
 57  { ...
 58  }
 59
 60  public String leValorParametro(String param)
 61  { ...
 62  }
 63
 64  public void leArquivo()
 65  { ...
 66  }
 67
 68  public void gravaArquivoPadrao()
 69  { ...
 70  }
 71
 72  public void atualizaArquivo()
 73  { ...
 74  }
 75
 76  public boolean existeBarraN(String p_diretorio)
 77  { ...
 78  }
 79
 80  public boolean existeBarraI(String p_diretorio)
 81  { ...
 82  }
 83
 84  public String verificaBarrasSOWindows(String p_diretorio)
 85  { ...
 86  }
 87
 88  public String verificaBarrasSOLinux(String p_diretorio)
 89  { ...
 90  }
 91
 92  public void atualizaArquivo(long p_intervalo,
 93                   char p_tipo_intervalo, int p_num_max_arquivos,
 94                   char p_incremental, String p_diretorio)
 95  { ...
 96  }
 97
 98  public boolean verificaSOWindows()
 99  { ...
100  }
101 }
```

FIGURA 4.11 – Classe **`ParametrosCkpt`** (continuação).

Os parâmetros disponíveis para configurar a operação da *Libcjp* devem ser escritos em um arquivo de parâmetros próprio chamado **`paramckpt.dat`**, o qual deve estar no diretório corrente da aplicação. Quando uma certa aplicação computacional escrita em Java fizer o uso da biblioteca, automaticamente ao ser executada, se o arquivo de parâmetros ainda não existir, este será criado automaticamente pela biblioteca com os valores-padrão para cada parâmetro. O programador da aplicação

também pode, antes de executar a aplicação, compilar e executar a classe **InterParamCkpt** (subseção 4.2.4), que serve de apoio ao programador para criar e configurar o arquivo de parâmetros. Os possíveis parâmetros do arquivo **paramckpt.dat**, seus possíveis-valores e seus valores-padrão são listados na figura 4.12.

Valores-padrão	**possíveis valores**
intervalo=1000	<**1**... (número correspondente ao tempo)>
tipo_intervalo=L	<**H** (**H**ora) **M** (**M**inuto) **S** (**S**egundo) **L** (mi**L**issegundo) **V** (sem inter**V**alo)>
num_max_arquivos=0	<**0**,**1** (sem efeito) **2**... (número máximo de arquivos de *checkpoint*)>
incremental=N	<**S**/**N**>
diretorio=.	<**.** (diretório desejado)>

FIGURA 4.12 – Valores possíveis dos parâmetros do arquivo "**paramckpt.dat**".

O parâmetro "**intervalo** = <*n*>" define um valor correspondente a um período mínimo de tempo que deve decorrer entre um *checkpointing* e outro. O valor- padrão é *n* = 1000. A interpretação do valor desse intervalo de tempo depende da grandeza definida pelo parâmetro **tipo_intervalo**. Se o tempo definido no parâmetro intervalo não transcorreu desde o *checkpoint* prévio, então as chamadas feitas pela aplicação do método **checkpoint_here()** – classe **Checkpointing** (subseção 4.2.2) são ignoradas.

O parâmetro "**tipo_intervalo** = <*H / M / S / L / V*>" especifica a unidade empregada para especificar o parâmetro **intervalo**. Este intervalo pode ser em horas *H*, minutos *M*, segundos *S*, milissegundos *L* ou sem intervalo *V*. O valor-padrão é = *L*. Quando o valor for *V*, ele desabilita o intervalo mínimo entre um *checkpointing* e outro, ou seja, as chamadas do método **checkpoint_here()** – classe **Checkpointing** (subseção 4.2.2) não serão ignoradas mesmo que o tempo decorrido seja inferior ao pré-estabelecido no parâmetro **intervalo**, e o *checkpointing* será executado sempre.

O parâmetro "**num_max_arquivos** = <*n*>" identifica o número máximo de arquivos de *checkpoint* para *n*. Este parâmetro pode ser usado tanto para o *checkpointing* incremental ou *checkpointing* não incremental. No caso específico do uso do mecanismo de *checkpointing* incremental, depois de *n* arquivos de *checkpoint* terem sido criados, os estados mais atuais dos objetos da aplicação são salvos no próximo arquivo de *checkpoint* e os arquivos velhos de *checkpoint* são descartados. Se *n* = 0 ou *n* = 1, o parâmetro não terá efeito. O valor-padrão é *n* = 0. Valores de *n* maiores do que 1 permitem ao usuário impor um equilíbrio entre o número de arquivos de *checkpoint* e a quantidade de espaço utilizado no disco.

Quando é executado um *checkpointing*, somente os estados dos objetos que foram modificados desde o *checkpoint* prévio mantido no disco, necessitam ser salvos. Os estados inalterados podem ser restabelecidos de *checkpoints* prévios. Isto caracteriza o mecanismo de *checkpointing* incremental.

O parâmetro <**incremental** = *S/N*> ativa ou desativa o mecanismo de *checkpointing* incremental. O padrão é *N*, (inativo).

Se o parâmetro **num_max_arquivos** for utilizado, quando o número de arquivos de *checkpoint* criados for igual ao valor deste parâmetro, o próximo *checkpoint* a ser salvo conterá os estados mais atuais dos objetos da aplicação. Para isso *Libcjp* invoca o método **realizaMergeCheckpoint(char flag)** – classe **Checkpointing** (subseção 4.2.2) não para fundir os arquivos velhos de *checkpoint*, mas para salvar os estados mais atuais dos objetos da aplicação em um novo arquivo *checkpoint*. Após o salvamento do arquivo de *checkpoint*, os arquivos velhos de *checkpoint* serão descartados. O valor do parâmetro **flag** do método neste caso será a constante **CHECKPOINTING** (subseção 4.2.7).

O parâmetro "**diretorio** = <**.**>" especifica o diretório no qual os arquivos de *checkpoint* serão gravados. O padrão corresponde ao diretório corrente.

4.2.6 Classe Tempo

A classe **Tempo** (figura 4.13) fornece um serviço de controle do tempo para o estabelecimento dos *checkpoints* utilizado na classe **Checkpointing** (subseção 4.2.2). Para que os *checkpointings* possam ser efetuados com um intervalo de tempo mínimo pré-estabelecido, a classe possui as variáveis: **intervalo** (linha 3) e **tipo_intervalo** (linha 9) para armazenarem informações sobre qual tempo deve decorrer entre um *checkpointing* e outro e qual o tipo de intervalo (hora, minuto, segundo ou milissegundo). Estas informações são atribuídas às variáveis **intervalo** e **tipo_intervalo** pelo método **setParametros(long intervalo, char tipo_intervalo)** (linha 15), e obtidas a partir dos valores dos parâmetros **intervalo** e **tipo_intervalo** do arquivo de parâmetros **paramckpt.dat** (subseção 4.2.5).

O método **verificaTempoAcimaIntervalo()** (linha 23) é usado no mecanismo de *checkpointing* (subseção 4.2.2) para verificar se já decorreu o intervalo de tempo pré-estabelecido. A cada arquivo de *checkpoint* salvo, o método **zeraTempo()** (linha 19) é invocado para inicializar as variáveis: **hora** (linha 4), **minuto** (linha 5), **segundo** (linha 6), **milissegundo** (linha 7), e **t_segini** (linha 8) responsáveis por armazenar o tempo exato no momento de invocação deste método.

```
 1 public class Tempo
 2 {
 3  private long intervalo;
 4  private int hora;
 5  private int minuto;
 6  private int segundo;
 7  private long milissegundo;
 8  private long t_segini;
 9  private char tipo_intervalo;
10
11  public Tempo()
12  { ...
```

FIGURA 4.13 – Classe **Tempo**.

```
13  }
14
15  public void setParametros(long intervalo, char tipo_intervalo)
16  { ...
17  }
18
19  public void zeraTempo()
20  { ...
21  }
22
23  public boolean verificaTempoAcimaIntervalo()
24  { ...
25  }
26 }
```

FIGURA 4.13 – Classe **Tempo** (continuação).

4.2.7 Interface Definicoes

A interface **Definicoes** contém as constantes utilizadas pelas classes **Checkpointing** e **ArquivoCkpt** (figura 4.14).

```
 1 public interface Definicoes
 2 {
 3  public final int INICIO_CONTEINER = 1;
 4  public final char RECUPERACAO = 'R';
 5  public final char CHECKPOINTING = 'C';
 6  public final Integer HEAD = new Integer(0);
 7  public final Integer NUM_ARQUIVO = new Integer(0);
 8  public final int MODIFICADO = 1;
 9  public final int NAO_MODIFICADO = -1;
10  public final int TF_F = 9999;
11 }
```

FIGURA 4.14 – Interface **Definições**.

Com exceção da constante **TF_F** – classe **ArquivoCkpt** (subseção 4.2.1), todas as outras constantes são utilizadas na classe **Checkpointing** (subseção 4.2.2). A seguir, estão listadas todas as constantes e as respectivas descrições do seu significado de utilização:

- **INICIO_CONTEINER**: utilizada para atribuir um valor inicial à variável **pos_conteiner_ckpt** – classe **Checkpointing**, que é responsável por servir de posição chave (*key*) para os objetos que serão atribuídos aos contêiners de objetos da classe **Checkpointing**: **conteiner_ckpt** (este contêiner armazena os estados dos objetos da aplicação, posteriormente é serializado e salvo no arquivo de *checkpoint*) e **conteiner_ckpt_estados** (este contêiner armazena os estados mais atuais dos objetos durante a execução da aplicação);
- **RECUPERACAO**: se o mecanismo de recuperação for executado com esta finalidade, e se os *checkpoints* forem gerados pelo mecanismo incremental, esta constante indicará ao método responsável por realizar a unificação dos arquivos de *checkpoint* (**realizaMergeCheckpoint(char flag)** –

classe **`Checkpointing`**), que este deverá se comportar para realizar a recuperação de estados dos objetos da aplicação;

- **`CHECKPOINTING`**: quando os arquivos de *checkpoint* forem gerados com o mecanismo incremental ativado e, se o parâmetro **`num_max_arquivos`** do arquivo de parâmetros **`paramckpt.dat`** (subseção 4.2.5) for maior ou igual a uma constante, indicará ao método **`realizaMergeCheckpoint(char flag)`** – classe **`Checkpointing`**, que este deverá salvar os estados mais atuais dos objetos da aplicação no próximo arquivo de *checkpoint*;
- **`HEAD`**: constante para indicar a posição chave (*key*) para o cabeçalho (variável **`cabecalho`** – classe **`Checkpointing`**) ser atribuído ao contêiner **`conteiner_ckpt`** – classe **`Checkpointing`**;
- **`NUM_ARQUIVO`**: esta constante tem a finalidade de indicar a posição chave (*key*) na variável **`cabecalho`** – classe **`Checkpointing`**, para que possa ser atribuído um objeto do tipo da classe **`Integer`** contendo o número seqüencial correspondente ao arquivo de *checkpoint* salvo no disco;
- **`MODIFICADO`**: constante para indicar que o estado do objeto a ser serializado foi modificado. Esta constante é importante quando o mecanismo de *checkpointing* incremental for utilizado para prover os arquivos de *checkpoints*, nestes apenas conterão os objetos cujos estados forem modificados e diferem portanto dos estados anteriores;
- **`NAO_MODIFICADO`**: constante para indicar que o estado do objeto não foi modificado. Este objeto não estará presente no arquivo de *checkpoint* quando o mecanismo utilizado for o *checkpointing* incremental;
- **`TF_F`**: constante utilizada na classe **`ArquivoCkpt`** para definir um tamanho físico a um vetor (denominado de **`arquivos`**) de objetos do tipo da classe **`File`**, responsável por armazenar informações dos arquivos de *checkpoint*.

5 Avaliação funcional e testes

Este capítulo tem como um dos objetivos mostrar o desempenho funcional da biblioteca através da execução de casos-exemplo. Para estes casos-exemplo, foram utilizadas algumas aplicações computacionais escritas em Java, que possuem comportamentos distintos. Também estão dentre os objetivos deste capítulo: demonstrar como a biblioteca pode ser usada; como pode ser feita a utilização do arquivo de parâmetros; e qual é o impacto que o uso da biblioteca causa com relação à execução das aplicações e nas atividades de programação – ou seja, o que o programador precisa alterar no código-fonte da aplicação para inclusão das chamadas aos métodos da biblioteca para ativar os mecanismos desejados.

Como esta dissertação tem um caráter mais prático, onde os algoritmos correspondentes aos mecanismos propostos foram implementados, realizou-se a verificação referente à correção funcional das aplicações somente através de testes funcionais para problemas pré-calculados, comparando-se as respostas obtidas na execução computacional e cálculo “manual” ou resultados obtidos nas referências consultadas. Em seqüência, como testes de uso da biblioteca, foram realizados experimentos com a execução das aplicações e a tomada de *checkpoints* .

Os testes para comprovação funcional do mecanismo de recuperação foram efetivados com a interrupção das aplicações (estas aplicações são listadas na tabela 5.1 da seção 5.1) de uma forma forçada, prematura. Como não se pensou em desenvolver ou adaptar um mecanismo pré-existente de inserção de falhas, esta técnica *ad-hoc* foi adotada para que se pudesse realizar os testes funcionais da recuperação. As aplicações após serem interrompidas, foram executadas novamente, mas não a partir do início, e sim a partir do último *checkpoint* armazenado e prosseguiram a computação até o término. Os resultados obtidos a partir das aplicações com o uso do mecanismo de recuperação foram então comparados aos obtidos com execuções das mesmas aplicações sem fazer uso da biblioteca, apresentando-se idênticos aos das execuções sem a interferência da biblioteca.

A demonstração de uso é uma conseqüência da exploração das aplicações e da forma pela qual foram inseridas as chamadas no código original. A inclusão de trechos de código e da montagem do arquivo de parâmetros permitem observar o uso dos mecanismos da biblioteca.

Finalmente, são mostrados resultados preliminares da interferência da biblioteca e da inserção dos mecanismos de *checkpointing* sobre a execução normal das aplicações (sem a ocorrência de falhas). Este é basicamente o objetivo principal ao qual está dedicado este capítulo de avaliação. Como o enfoque principal deste trabalho diferencia-se dos trabalhos investigados e apresentados na seção 1.2, este capítulo não compara a implementação dos mecanismos com os resultados obtidos em outros trabalhos. Fica a critério do usuário o ajuste de parâmetros ao seu uso particular como forma de reduzir as conseqüências no uso da biblioteca, em termos de impacto que ela trará às aplicações, a partir da compreensão dos seus mecanismos.

Na seqüência deste capítulo, são apresentadas as aplicações empregadas para testes e avaliação funcional da biblioteca *Libcjp*, juntamente com as suas diferentes formas de utilização. Ao final, são apresentados os resultados da avaliação de desempenho e os resultados dos experimentos realizados.

Os experimentos destinados à avaliação de desempenho foram realizados em um microcomputador com processador *Duron* 850 Mhz, com 128 *Mbytes* de memória RAM, executando o sistema operacional *Linux Mandrake release* 8.0 com *kernel* 2.4.3-20mdk.

5.1 Aplicações-exemplo

Nesta seção, são descritas as aplicações que foram utilizadas para testar e avaliar o desempenho funcional da biblioteca *Libcjp*. Algumas destas aplicações são programas científicos típicos. Todas estas aplicações foram escritas na linguagem de programação Java e podem beneficiar-se do uso do *checkpointing* para prover tolerância a falhas.

Um resumo das aplicações-exemplo utilizadas nesta dissertação consta da tabela 5.1, especificando: uma descrição rápida da aplicação, o mnemônico correspondente, o cenário de execução e a origem da aplicação (se foi desenvolvida ou obtida de outra fonte). As aplicações são descritas mais detalhadamente nas próximas subseções. Os códigos-fonte das aplicações-exemplo são listados no anexo 2.

TABELA 5.1 – Lista das aplicações-exemplo utilizadas para teste e avaliação da biblioteca *Libcjp*.

Aplicação	**mnemônico**	**cenário**	**origem**
Multiplicação de Matrizes	MAT	objetos centralizados	desenvolvida
Cálculo da Transformada Discreta do Cosseno	TDC	objetos centralizados	desenvolvida
Método de ordenação *ShellSort*	SHELL	objetos centralizados	desenvolvida
Método de ordenação *HeapSort*	HEAP	objetos centralizados	desenvolvida
Método da Eliminação Gaussiana com Pivoteamento	GAUSPIV	objetos centralizados	desenvolvida
Aplicação RMI (Loja de Suprimentos Java)	LojaRMI	objetos distribuídos	obtida/ modificada

As aplicações desenvolvidas a partir de exemplos da matemática, como MAT, TDC, SHELL, HEAP e GAUSPIV, foram escolhidas por um critério de facilidade: o conhecimento prévio do autor auxiliaria no seu desenvolvimento. Adicionalmente, constatou-se que estas aplicações apresentam características diferenciadas, o que enriqueceria os experimentos realizados. Já a aplicação LojaRMI foi inicialmente obtida a partir de um exemplo proveniente da página da disciplina de Modelos de Linguagens de Programação (disponível em: http://inf.ufrgs.br/aulas/mlp) e posteriormente modificada.

5.1.1 Multiplicação de Matrizes (MAT)

Esta é uma aplicação que faz multiplicação de duas matrizes de 615 x 615 elementos do tipo inteiro. Para a realização dos testes, as matrizes-origem são geradas em memória com valores que seguem uma seqüência crescente de números inteiros, através da execução de métodos específicos; não há entrada de dados proveniente da leitura de arquivos. A primeira matriz *m1* é gerada com números crescentes no sentido das linhas da matriz; a segunda matriz *m2* é gerada com números crescentes no sentido das colunas da matriz. Estes valores foram escolhidos sem nenhum critério especial, mas apenas para se ter os mesmos dados em todas as execuções da aplicação. As matrizes *m1* e *m2* são multiplicadas produzindo acessos constantes de leitura e escrita na memória e, ao final da computação, o produto desta multiplicação é salvo em um arquivo-texto no meio de armazenamento. As chamadas de métodos da biblioteca para o salvamento dos arquivos de *checkpoint* foram introduzidas apenas durante a execução dos cálculos matemáticos. O código-fonte desta aplicação, contendo as chamadas de métodos para os mecanismos de *checkpointing* e recuperação, pode ser encontrado no anexo 2.1.

5.1.2 Cálculo da Transformada Discreta do Cosseno (TDC)

As informações que descrevem a Transformada Discreta do Cosseno foram obtidas em Nelson [NEL91].

A Transformada Discreta do Cosseno (TDC) é a transformada utilizada no processo de compressão do JPEG (*Joint Photographic Experts Group*), que converte um bloco de *pixels* em uma matriz de coeficientes, descorrelacionando a informação da imagem.

A transformada é a chave para o processo de compressão: ela toma um conjunto de pontos no domínio espacial e os transforma em uma representação equivalente no domínio da freqüência. Um dos objetivos da transformada é reescrever a imagem original de maneira que os novos dados não tenham qualquer correlação, eliminando-se portanto as redundâncias estatísticas. Outros aspectos relevantes impõem que os novos dados exijam menos espaço, do mesmo modo que o algoritmo para obtê-los seja o mais eficiente possível.

A Transformada Discreta do Cosseno (TDC) é definida pela seguinte equação:

$$C(i,j) = \alpha(i)\alpha(j)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\cos\left(\frac{(2x+1)i\pi}{2N}\right)\cos\left(\frac{(2y+1)j\pi}{2N}\right)$$

para i, j = 0, 1... , N-1, e

$$\alpha(x) = \begin{cases} \frac{1}{\sqrt{N}} & \text{se} \quad x = 0 \\ \sqrt{\frac{2}{N}} & \text{para} \quad x = 1, 2, \ldots, N-1 \end{cases}$$

onde :

C : Transformada Discreta do Cosseno
f (x,y): Valores da imagem original (tons de cinza)
N : Dimensão da imagem (ou bloco)
A Transformada Inversa do Cosseno (ITDC) é definida pela seguinte equação:

$$f(x,y) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \alpha(i)\alpha(j)C(i,j)\cos\left(\frac{(2x+1)i\pi}{2N}\right)\cos\left(\frac{(2y+1)j\pi}{2N}\right)$$

A aplicação desenvolvida como exemplo tem por objetivo efetuar o cálculo da TDC a partir de uma matriz 512 x 512, que representa as informações de cada *pixel* em tons de cinza de uma suposta imagem. Nesta aplicação, a imagem não provém de dados guardados no meio de armazenamento. Uma matriz é gerada em memória através de um algoritmo da própria aplicação com números crescentes no sentido da linha da matriz, correspondendo à imagem. Após ter sido gerada a matriz correspondente à imagem, é aplicada sobre ela o algoritmo da Transformada Discreta do Cosseno. Neste momento, durante os cálculos, a matriz é constantemente lida e uma outra matriz é gerada, o que representa um processamento constante. Após todos os cálculos terem sido efetuados, a matriz resultante, que contém as informações descorrelacionadas da imagem original, é salva em um arquivo-texto no meio de armazenamento. Foram introduzidas chamadas para a atividade de *checkpointing* no algoritmo que efetua os cálculos da transformada. Por fim, o algoritmo da Transformada Inversa do Cosseno é aplicado sobre a matriz resultante da transformada, retornando a matriz que corresponde a imagem original. Como último passo dessa aplicação, a matriz resultante da transformada inversa também é salva no mesmo arquivo-texto. O código-fonte desta aplicação, contendo as chamadas de métodos para os mecanismos de *checkpointing* e recuperação, pode ser encontrado no anexo 2.2.

5.1.3 Método de ordenação *ShellSort* (SHELL)

O método de ordenação por inserção de incrementos decrescentes (*ShellSort*) foi proposto por D. L. Shell, como sendo um refinamento do método de ordenação por Inserção Direta [WIR89]. A figura 5.1 mostra um exemplo da aplicação do método, cujo funcionamento é explicado a seguir.

Primeiramente, todos os elementos que estiverem a intervalos de quatro posições entre si na seqüência corrente são agrupados e ordenados separadamente. Este processo é denominado de ordenação de distância 4. No exemplo mostrado na figura 5.1, cada

grupo contém exatamente dois elementos. Após este primeiro passo, os elementos são reagrupados em grupos com elementos cujo intervalo é de duas posições, sendo então ordenados novamente. Este processo é denominado de ordenação de distância 2. Finalmente, em um terceiro passo, todos os elementos são ordenados através de uma ordenação simples, ou ordenação de distância 1.

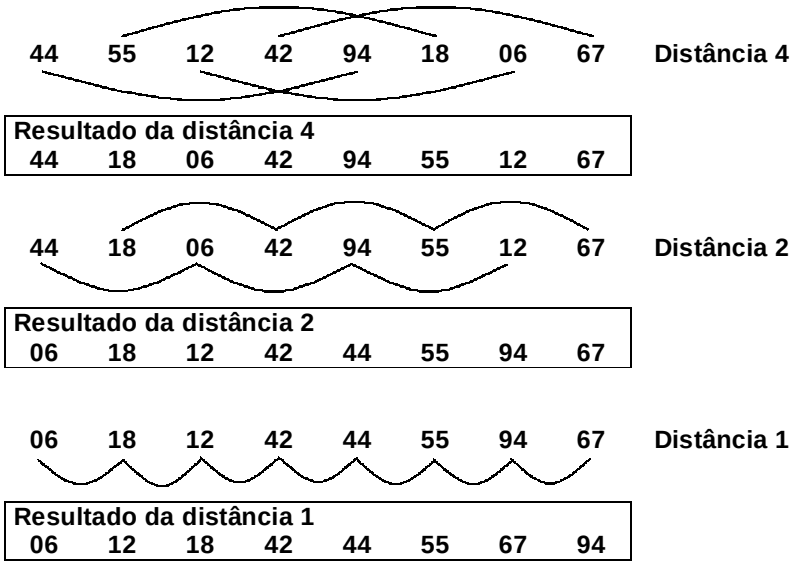

FIGURA 5.1 – Exemplo do método de ordenação *ShellSort*.

Esta aplicação realiza a ordenação de um vetor de elementos inteiros utilizando o método *ShellSort*. Primeiramente, o vetor é inicializado com uma seqüência de 100.000 elementos inteiros em ordem inversa, gerados pela própria aplicação. O único critério empregado na escolha desses dados foi o de ter-se os mesmos dados em todas as execuções da aplicação durante o experimento. Após a formação inicial, o método de ordenação é aplicado sobre este vetor. Esta aplicação apresenta processamento constante durante a execução do algoritmo de ordenação, onde também são feitas constantes leituras e escritas na memória. As chamadas aos métodos da biblioteca para *checkpointing* foram inseridas junto ao algoritmo básico de ordenação *Shell*. Ao final do processamento, o resultado produzido (o vetor ordenado) é exibido na tela. O código-fonte desta aplicação contendo as chamadas de métodos da biblioteca pode ser visualizado no anexo 2.3.

5.1.4 Método de ordenação *HeapSort* (HEAP)

O método de ordenação *HeapSort* é dividido em duas partes: primeiro monta-se uma árvore binária chamada *heap* para, em seguida, classificar através de seleção (na árvore). Mais detalhes sobre este método podem ser encontrados em Azeredo [AZE96], fonte da qual foram extraídas as informações aqui reproduzidas.

É considerado um vetor de chaves C_1, C_2, ...,C_n, como sendo a representação de uma árvore binária, usando a seguinte interpretação dos índices das chaves:

C_1 é raiz da árvore;

C_{2i} = subárvore da esquerda de C_i

C_{2i+1} = subárvore da direita de C_i

} para i = 1, n div 2

A representação da árvore de um vetor $C_{1..7}$ é mostrada na figura 5.2.

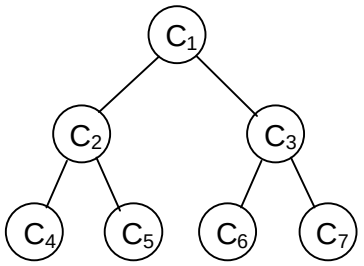

FIGURA 5.2 – Árvore representada pelo vetor $C_{1..7}$.

Os passos de ordenação consistem em trocar as chaves dentro do vetor, de tal forma que estas passem a formar uma hierarquia, na qual todas as raízes das subárvores sejam maiores ou iguais a qualquer uma das suas sucessoras ($C_i \geq C_{2i}$ e $C_i \geq C_{2i+1}$). Quando todas as raízes das subárvores satisfizerem essas condições, a árvore forma um *heap*.

São efetuadas as trocas de posições das chaves no vetor, de tal forma que a árvore representada passe a ser um *heap*. Este processo é feito testando-se cada uma das subárvores que possuam pelo menos um sucessor para verificar se elas satisfazem, separadamente, a condição do *heap*. O teste inicia-se pela última subárvore, cuja raiz está na posição *n div* 2 do vetor de chaves, prosseguindo-se, a partir daí, para as subárvores que antecedem esta, até testar sua raiz. Sempre que uma subárvore não formar um *heap*, seus componentes são rearranjados de modo a formá-lo.

Para exemplificar todo o processo, suponha-se o seguinte vetor de chaves e sua interpretação sob a forma de árvore mostrada na figura 5.3.

Vetor de chaves:

1	2	3	4	5	6	7
12	**09**	**13**	**25**	**18**	**10**	**22**

Interpretação sob a forma de árvore:

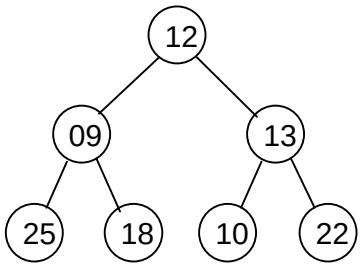

FIGURA 5.3 – Vetor-exemplo e sua interpretação em forma de árvore.

A transformação desta árvore em *heap* inicia-se pela subárvore cuja raiz é 13 e prossegue até a subárvore cuja raiz é a raiz principal de toda a árvore. Uma vez que

todas as subárvores formam *heaps*, a árvore toda também é um *heap*. Depois de todo o processo, a chave que está na raiz é a maior de todas, então sua posição definitiva correta na ordem crescente é na última posição do vetor, onde ela é colocada, por troca com a chave que ocupa aquela posição. Este processo repete-se até que todas as chaves do vetor estejam ordenadas. O vetor de chaves e sua interpretação sob a forma de árvore resultante de todo o processo são mostrados na figura 5.4.

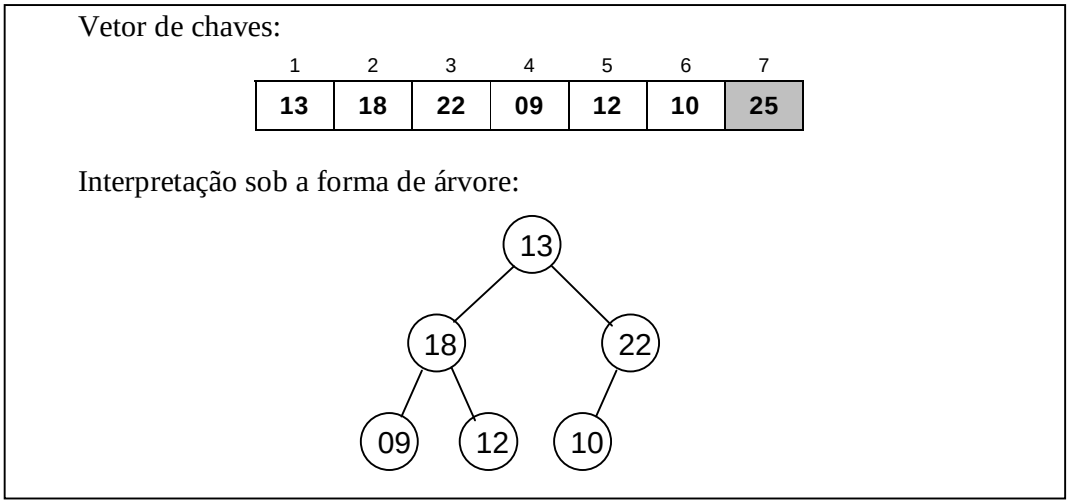

FIGURA 5.4 – Vetor-exemplo e sua interpretação em forma de árvore depois da ordenação de uma chave.

A aplicação desenvolvida a partir das informações descritas nesta subseção realiza a ordenação de um vetor de elementos inteiros utilizando o método *HeapSort*. Primeiramente o vetor é inicializado com uma seqüência de 100.000 elementos inteiros, em ordem inversa, gerados pela própria aplicação. Novamente, visou-se ter os mesmos dados em todas as execuções da aplicação durante o experimento. Após esta atividade, o método de ordenação é aplicado sobre este vetor. Esta aplicação apresenta processamento constante durante a execução do algoritmo de ordenação, onde também são feitas constantes leituras e escritas na memória. As chamadas aos métodos da biblioteca para *checkpointing* foram adicionadas ao código do algoritmo de ordenação *Heap*. Ao final do processamento, o resultado produzido (o vetor ordenado) é exibido na tela. O código-fonte desta aplicação contendo as chamadas de métodos da biblioteca pode ser visualizado no anexo 2.4.

5.1.5 Método da Eliminação Gaussiana com Pivoteamento (GAUSPIV)

O método da Eliminação de Gauss consiste em transformar convenientemente o sistema linear original num sistema linear equivalente com matriz dos coeficientes triangular superior, pois estes são de resolução imediata. Ruggiero [RUG88] é uma boa fonte de consulta sobre o método da Eliminação Gaussiana com Pivoteamento.

Seja o sistema linear $Ax = B$, onde A identifica a matriz $n \times n$, triangular superior, com elementos da diagonal diferentes de zero. Escrevendo as equações deste sistema, tem-se:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n = b_1 \\ a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n = b_2 \\ a_{33}x_3 + \ldots + a_{3n}x_n = b_3 \\ \vdots \\ a_{mn}x_n = b_m \end{cases}$$

$a_{ii} \neq 0 \quad i = 1, 2, \ldots, n$

De cada equação, iniciando pela última, obtém-se:

$x_n = b_n / a_{nn}$

$x_{n-1} = (b_{n-1} - a_{n-1,\ n}\, x_n) / a_{n-1,\ n-1}$

e assim sucessivamente até x_1:

$x_1 = (b_1 - a_{12}\, x_2 - a_{13}x_3 - \ldots - a_{1n}x_n) / a_{11}$

A restrição $a_{ii} \neq 0$ é feita pois, se algum $a_{ii} = 0$, então, haverá infinitas soluções ou o sistema não admite solução (sistema incompatível).

No processo de eliminação, os elementos da diagonal são chamados Pivôs. Para garantir a estabilidade do método da Eliminação Gaussiana, é necessário adotar uma Estratégia de Pivoteamento, trocando a ordem das equações não apenas quando o Pivô é nulo, mas também quando ele é muito próximo de zero. Isto ocorre porque, em qualquer calculadora ou computador, os cálculos são efetuados com aritméticas de precisão finita. Pivôs próximos de zero dão origem a multiplicadores bem maiores que a unidade, os quais, por sua vez, originam ampliações dos erros de arredondamento.

A aplicação desenvolvida neste exemplo implementa a Eliminação Gaussiana com Pivoteamento para resolver um sistema linear de 30 x 30. Como entrada de dados, a aplicação lê do meio de armazenamento um arquivo-texto contendo a matriz dos coeficientes *A* e o vetor dos termos independentes *B*. Durante a execução do algoritmo para resolver o sistema, há operações repetidas de leitura e escrita na memória, através das quais muitos dados são trabalhados. No algoritmo que efetua os cálculos da Eliminação Gaussiana, foram introduzidas chamadas para os métodos da biblioteca para realizar a atividade de *checkpointing*. Ao longo da computação, os dados resultantes dos cálculos efetuados para a resolução do sistema são gravados em um arquivo-texto denominado de "**`GausPiv_result.txt`**". No anexo 2.6, pode ser encontrada a aplicação **`ConstroiMatriz.java`**, responsável por gerar o sistema linear a ser resolvido pela Eliminação Gaussiana. A aplicação ConstroiMatriz gera a matriz dos coeficientes *A* e o vetor dos termos independentes *B*; estes são gravados num arquivo-texto denominado "**`mv.txt`**" que será lido pela aplicação da eliminação gaussiana no início de sua execução, para então efetuar os cálculos. O código-fonte da aplicação da eliminação gaussiana contendo as chamadas de métodos da biblioteca, pode ser encontrado no anexo 2.5.

5.1.6 Aplicação RMI (Loja de Suprimentos Java)

Esta é uma típica aplicação cliente-servidor implementada para trabalhar com objetos distribuídos usando a tecnologia Java RMI. A finalidade da aplicação resume-se

em controlar vendas de suprimentos através de pedidos de compra efetuados pelo cliente RMI da aplicação; a cada suprimento solicitado, é decrementado o estoque. O código-fonte desta aplicação pode ser encontrado no anexo 2.7.

A aplicação possui uma interface **`LojaInterface,`** a qual especifica métodos que compõem os serviços oferecidos e implementados pelo servidor da aplicação na classe **`LojaServer`**. Os serviços que compõem o servidor (figura 5.5) e seus respectivos métodos são listados a seguir:

- **efeturar compra de um item**: este serviço é implementado pelo método **`compraItem(...)`** (linha 8);
- **verificação de item no estoque**: este serviço é implementado pelo método **`existeQuantidade(...)`** (linha 13);
- **listar itens de suprimentos**: este serviço é implementado pelo método **`listItems()`** (linha 17);
- **listar quantidade em estoque dos suprimentos**: este serviço é implementado pelo método **`listQuantidades()`** (linha 21).

```
 1 public class LojaServer extends UnicastRemoteObject
 2                                         implements LojaInterface
 3 {
 4  public LojaServer(String[] args) throws RemoteException
 5  { ...
 6  }
 7
 8  public boolean compraItem(String item, int pos)
 9                                           throws RemoteException
10  { ...
11  }
12
13  public boolean existeQuantidade(int pos) throws RemoteException
14  { ...
15  }
16
17  public String[] listItems() throws RemoteException
18  { ...
19  }
20
21  public String[] listQuantidades() throws RemoteException
22  { ...
23  }
24
25  public void startUp(String[] args)
26  { ...
27  }
28
29  public static void main(String[] args)
30  { ...
31  }
32 }
```

FIGURA 5.5 – Classe do Servidor RMI da Loja de Suprimentos.

O Cliente RMI é implementado na classe **`LojaClient`**, a qual possui uma janela com duas listas, uma para mostrar na tela a descrição dos suprimentos e outra para as quantidades que cada suprimento possui em estoque. Também possui um botão para

efetuar a compra de um suprimento e um botão para atualizar a tela para o caso de múltiplos clientes estarem comprando suprimentos. A interface do Cliente RMI da aplicação é mostrada na figura 5.6.

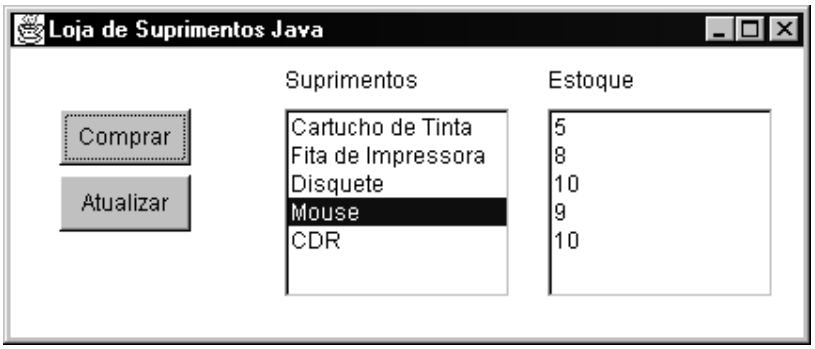

FIGURA 5.6 – Interface do Cliente RMI da aplicação Loja de Suprimentos.

5.2 Utilização da biblioteca

Para utilizar a biblioteca *Libcjp* e obter a funcionalidade e o desempenho esperado, deve-se recorrer à configuração dos valores dos parâmetros disponíveis no arquivo de parâmetros **`paramckpt.dat`** (subseção 4.2.5). As diferentes formas de utilização da biblioteca são alcançadas através de modificações neste arquivo.

Como um primeiro exemplo é utilizada a aplicação MAT para demonstrar as diferentes formas de utilização da biblioteca *Libcjp* e as funcionalidades oferecidas por ela.

No código-fonte da aplicação, apenas são necessárias inclusões de chamadas aos métodos da biblioteca para ativar os mecanismos de *checkpointing* e recuperação. Na figura 5.7, é mostrado um trecho do código da aplicação MAT correspondendo ao método **`multMatriz(...)`** (linha 6), sem fazer o uso da *Libcjp* (sem qualquer modificação). Este trecho foi escolhido para a inclusão das chamadas aos métodos de *checkpointing* e recuperação, pelo fato do método **`multMatriz(...)`** ser responsável pela manipulação de leitura e escrita em memória dos dados da aplicação. É neste trecho que ocorre a execução dos cálculos; sendo repetido *n* vezes durante a execução da aplicação. Inserir as chamadas para a biblioteca neste local significa que o procedimento vai ser efetivamente executado rotineiramente. Como a aplicação permanece na execução deste trecho a maior parte do tempo, estima-se que a probabilidade de falhas ali seja maior.

```
1 public class OperaMatriz
2 {
3  ...
4  ...
5
6  public Integer[][] multMatriz(Integer m1[][], Integer m2[][])
7  {
8    Integer m3[][] = new Integer[615][615];
9
```

FIGURA 5.7 – Aplicação MAT original.

```
10    for (int i=0  ; i<615 ; i++)
11    {
12      for (int j=0 ; j<615 ; j++)
13      {
14        int s=0;
15        for (int k=0 ; k<615 ; k++)
16          s+=(m1[i][k].intValue()*m2[k][j].intValue());
17        m3[i][j] = new Integer(s);
18      }
19    }
20    return m3;
21  }
22  ...
23 }
```

FIGURA 5.7 – Aplicação MAT original (continuação).

Na figura 5.8, são mostrados os trechos de código da aplicação MAT (os mesmos da figura 5.7), com as inserções e modificações necessárias ao uso da biblioteca *Libcjp*, destacadas em negrito.

```
 1 public class OperaMatriz
 2 {
 3  Checkpointing c = new Checkpointing();
 4  ...
 5
 6  public Integer[][] multMatriz(Integer m1[][], Integer m2[][],
 7                                                 String args[])
 8  {
 9    Integer m3[][] = new Integer[615][615];
10    int ini=0;
11
12    if (c.verificaRecuperacao(args))
13    {
14      c.recover_oockpt();
15      m3=(Integer[][])c.recover();
16      ini=(((Integer)c.recover()).intValue());
17    }
18
19    for (int i=ini  ; i<615 ; i++)
20    {
21      for (int j=0 ; j<615 ; j++)
22      {
23        int s=0;
24        for (int k=0 ; k<615 ; k++)
25          s+=(m1[i][k].intValue()*m2[k][j].intValue());
26        m3[i][j] = new Integer(s);
27      }
28      c.save(m3);
29      c.save(new Integer(i+1));
30      c.checkpoint_here();
31    }
32    return m3;
33  }
34  ...
35 }
```

FIGURA 5.8 – Aplicação MAT fazendo o uso da *Libcjp*.

Como modificações e inserções, pode-se observar os pontos a seguir relatados. O primeiro refere-se à instanciação do objeto **`c`** da classe **`Checkpointing`**

(subseção 4.2.2), que é responsável pela execução dos métodos que ativarão os mecanismos de *checkpointing* e recuperação da biblioteca (linha 3). O método **`multMatriz`** (linhas 6 e 7) possui um novo argumento passado como de parâmetro – **`String args[]`**, que contém os argumentos utilizados na linha de comando para a execução da aplicação. A variável **`ini`**, declarada na linha 10, é utilizada para servir de controladora de iterações para cada linha da matriz. Neste tipo de aplicação que possui uma repetição, faz-se necessário o uso de uma variável para ser utilizada como uma controladora das iterações. O método **`c.verificaRecuperacao(args)`** (linha 12) atua para definir a linha de comando de execução da aplicação, sendo responsável por ativar o mecanismo de recuperação quando estiver estipulado. Na linha 14, a execução do método **`c.recover_oockpt()`** ativa o mecanismo de recuperação, restabelecendo os estados mais atuais dos objetos da aplicação a partir dos arquivos de *checkpoint*.

Os estados dos objetos da aplicação são restabelecidos através do método **`c.recover()`** (linhas 15 e 16). Para salvar os arquivos dos *checkpoints*, é utilizado o método **`c.checkpoint_here()`** (linha 30). Mas, antes do mecanismo de *checkpointing* ser ativado, os estados atuais dos objetos da aplicação são passados via parâmetro para a biblioteca através do método **`c.save(...)`** (linha 28 e 29).

Como outro exemplo de utilização da *Libcjp* é utilizada a aplicação SHELL. Observa-se também poucas modificações no código-fonte. As inserções para invocar os métodos que estabelecerão os *checkpoints* e que proverão a recuperação, assim como as modificações feitas, são muito parecidas com as do exemplo anterior, demonstrado na aplicação MAT.

Para ilustrar o uso, neste caso da aplicação SHELL, acrescentou-se a utilização da funcionalidade para registrar os tempos decorrentes para a atividade de *checkpointing* e o tempo total de execução da aplicação. A figura 5.9 mostra trechos do código original da aplicação SHELL (sem fazer o uso da *Libcjp*).

```
 1 public class ShellSort
 2 {
 3  ...
 4  ...
 5
 6  public Integer[] shellSort(Integer vet[])
 7  {
 8    int dist,i,j,k,tl;
 9    tl=vet.length;
10    dist=4;
11    while (dist>=1)
12    {
13      for (i=1 ; i<=dist ; i++)
14      {
15        j=i-1;
16        while ((j+dist)<tl)
17        {
18          if (vet[j+dist].intValue()<vet[j].intValue())
19          {
20            k=j;
21            trocaPosicao(vet,j,j+dist);
22            if (k-dist>=0)
23            {
24              while ((k-dist>=0) &&
25                     (vet[k].intValue()<vet[k-dist].intValue()))
```

FIGURA 5.9 – Aplicação SHELL original.

```
26              {
27                if (k-dist>=0)
28                  trocaPosicao(vet,k,k-dist);
29                k-=dist;
30              }
31            }
32          }
33          j+=dist;
34        }
35      }
36      dist=(int)dist/2;
37    }
38  }
39 ...
40 }
```

FIGURA 5.9 – Aplicação SHELL original (continuação).

Na figura 5.10, são mostrados os mesmos trechos de código da aplicação SHELL, fazendo o uso da biblioteca *Libcjp*, destacando além das inserções para o *checkpointing* e recuperação, a funcionalidade para registrar os tempos da aplicação. No corpo do construtor da classe **`ShellSort`** (linha 7), é invocado o método **`c.tempoInicialEstatistica(...)`**; o nome do arquivo é passado como parâmetro para especificar onde serão registradas as informações de tempos da aplicação. Na linha 62, o método **`fim()`** foi criado para ser invocado como última instrução da aplicação (linha 71). No corpo deste método apenas é invocado o método **`c.tempoFinalEstatistica()`** (linha 64) que finaliza o arquivo de registro de tempos, após gravar o tempo total de execução da aplicação.

```
 1 public class ShellSort
 2 {
 3  Checkpointing c = new Checkpointing();
 4
 5  public ShellSort()
 6  {
 7    c.tempoInicialEstatistica("ShellSort.dat");
 8  }
 9  ...
10
11  public Integer[] shellSort(Integer vet[], String args[])
12  {
13    int dist,i,j,k,tl,ini=1;
14    tl=vet.length;
15    dist=4;
16
17    if (c.verificaRecuperacao(args))
18    {
19      //restaurando os estados
20      c.recover_oockpt();
21      ini=(((Integer)c.recover()).intValue());
22      dist=(((Integer)c.recover()).intValue());
23      vet=(Integer[])c.recover();
24    }
25    while (dist>=1)
26    {
27      for (i=ini ; i<=dist ; i++)
28      {
```

FIGURA 5.10 – Aplicação SHELL fazendo o uso da *Libcjp*.

```
29        j=i-1;
30        while ((j+dist)<tl)
31        {
32          if (vet[j+dist].intValue()<vet[j].intValue())
33          {
34            k=j;
35            trocaPosicao(vet,j,j+dist);
36            if (k-dist>=0)
37            {
38              while ((k-dist>=0) &&
39                       (vet[k].intValue()<vet[k-dist].intValue()))
40              {
41                if (k-dist>=0)
42                  trocaPosicao(vet,k,k-dist);
43                k-=dist;
44              }
45            }
46          }
47          j+=dist;
48          //salvando os estados
49          c.save(new Integer(i));
50          c.save(new Integer(dist));
51          c.save(vet);
52          c.checkpoint_here();
53        }
54      }
55      dist=(int)dist/2;
56      ini=1; //inicializa o ini para a proxima repeticao
57    }
58    return vet;
59  }
60  ...
61
62  public void fim()
63  {
64    c.tempoFinalEstatistica();
65  }
66
67  public static void main(String args[])
68  {
69    ShellSort s = new ShellSort();
70    ...
71    s.fim();
72  }
73 }
```

FIGURA 5.10 – Aplicação SHELL fazendo o uso da *Libcjp* (continuação).

Como pode ser observado nos exemplos correspondentes às aplicações MAT e SHELL, a forma de utilização da biblioteca *Libcjp* não apresenta muitas variações quanto ao uso, em termos de adições ou modificações no código prévio, mas isto depende do escopo de cada aplicação. As aplicações TDC, HEAP e GAUSPIV seguem o mesmo princípio de utilização da *Libcjp*. Vale ressaltar que a escolha mais apropriada dos pontos exatos onde se deve inserir as linhas de código para prover o salvamento dos arquivos de *checkpoint* e a recuperação pode apresentar diferenças em função do perfil e detalhes de implementação das aplicações. Havendo interesse em consultar detalhes adicionais, cabe lembrar que todos os códigos-fonte das aplicações utilizadas nesta dissertação para teste e avaliação, originais e modificados devido à utilização da biblioteca *Libcjp*, são mostrados no anexo 2.

O comportamento da *Libcjp* é configurado alterando os valores dos parâmetros no arquivo **`paramckpt.dat`**. Um exemplo de configuração do arquivo de parâmetros é demonstrado na figura 5.11. A aplicação que fizer o uso da *Libcjp* com esta configuração estabelecerá cada *checkpoint* com um tempo hipotético de 2 segundos (2000 milissegundos) como intervalo mínimo entre cada par de *checkpoints*, atendendo aos parâmetros **`intervalo`** (linha 1) e **`tipo_intervalo`** (linha 2). Os *checkpoints* serão efetuados sem empregar o mecanismo de *checkpointing* incremental (linha 4). O diretório-destino para salvar os arquivos de *checkpoint* é o diretório corrente da aplicação (linha 5).

```
1  intervalo=2000
2  tipo_intervalo=L
3  num_max_arquivos=0
4  incremental=N
5  diretorio=.
```

FIGURA5.11 – Configuração do arquivo de parâmetros com o mecanismo de *checkpointing* incremental desativado.

Um outro exemplo de configuração do arquivo de parâmetros é demonstrado na figura 5.12. Neste exemplo, a aplicação estabelecerá os *checkpoints* com um tempo hipotético de 3 minutos de intervalo mínimo entre dois *checkpoints* consecutivos (configurado nas linhas 1 e 2). A configuração da linha 3 faz com que o número máximo de arquivos de *checkpoint* seja = 5. Os *checkpoints* serão salvos utilizando o mecanismo incremental (linha 4). O diretório-destino para salvar os arquivos de *checkpoints* está definido na linha 5: `/usr/home/aplicacao` (exemplo de caminho de diretório no sistema operacional *Linux*).

```
1  intervalo=3
2  tipo_intervalo=M
3  num_max_arquivos=5
4  incremental=S
5  diretorio=/usr/home/aplicacao
```

FIGURA5.12 – Configuração do arquivo de parâmetros com o mecanismo de *checkpointing* incremental ativado.

5.2.1 Execução da aplicação RMI fazendo uso da biblioteca *Libcjp*

A execução da aplicação RMI (Loja de Suprimentos Java) foi feita utilizando dois microcomputadores rodando o sistema operacional *Windows* 95, numa rede local.

Para executar a aplicação RMI, após compilar as classes da interface (**`LojaInterface.java`**), do servidor (**`LojaServer.java`**) e do cliente (**`LojaClient.java`**), deve-se previamente gerar os módulos *stub* para a máquina cliente e *skeleton* para a máquina servidora, utilizando o pré-compilador RMI, através da linha de comando **`rmic LojaServer`**. Isto deverá gerar dois arquivos: **`LojaServer_Stub.class`** (módulo *stub*) e **`LojaServer_Skel.class`** (módulo *skeleton*).

Também é necessário, antes de executar a aplicação, criar um arquivo de permissão que deve estar acessível ao servidor no momento da execução. Este arquivo deve se chamar **`lojapolicy`** e ter o seguinte conteúdo:

```
grant{permission java.security.AllPermission;};
```

Para executar o servidor, deve-se disparar o serviço de nomes a partir do mesmo diretório onde se encontram as classes do servidor. Isto é feito em uma janela de *Prompt* do MS-DOS com a seguinte linha de execução:

```
start rmiregistry
```

Logo, em seguida, deve-se abrir uma outra janela de *Prompt* do MS-DOS e executar o servidor através da linha de comando:

```
java -Djava.security.policy=lojapolicy LojaServer
```

Para executar o cliente, deve-se abrir uma janela de *Prompt* do MS-DOS e digitar a seguinte linha de comando:

```
java -Djava.security.policy=lojapolicy LojaClient
```

Para fazer uso da biblioteca *Libcjp* nesta aplicação, algumas inserções de linhas devem ser feitas na classe do servidor (**`LojaServer.java`**) para invocar os métodos responsáveis por efetuar os *checkpointings* e prover a recuperação no caso da ocorrência de falhas no servidor. Estas modificações no código-fonte da classe do servidor são mostradas na figura 5.13. Nela, pode-se observar: a instanciação do objeto **`c`** da classe **`Checkpointing`** (subseção 4.2.2), que é responsável pela execução dos métodos que ativarão os mecanismos de *checkpointing* e recuperação (linha 4); os métodos **`c.save(...)`** e **`c.checkpoint_here()`** (linhas 21, 22 e 23), responsáveis por estabelecer os *checkpoints*, que foram adicionados ao método **`compraItem(...)`** (linha 10). Toda vez que o servidor receber uma chamada remota a este método para a compra de um suprimento, o estoque é decrementado e os estados dos objetos **`Items`** (lista de suprimentos – linha 7) e **`Quantidades`** (estoque – linha 8) são serializados para um arquivo de *checkpoint*.

Para prover uma possível recuperação, linhas de código foram adicionadas ao método **`startUp(String[] args)`** (linha 29). Na linha 36, observa-se a invocação do método **`c.recover_oockpt()`** para ativar o mecanismo de recuperação e com isso restaurar os estados dos objetos **`Items`** e **`Quantidades`**.

```
 1 public class LojaServer extends UnicastRemoteObject
 2                                          implements LojaInterface
 3 {
 4  Checkpointing c = new Checkpointing();
 5  ...
 6
 7  private Vector Items;
 8  private Vector Quantidades;
 9
10  public boolean compraItem(String item, int pos)
11                                            throws RemoteException
12  {
13    System.out.println("Recebendo pedido de compra: "+item+".\n");
14    if (Items.contains(item))
15    {
```

FIGURA 5.13 – Servidor da aplicação LojaRMI fazendo o uso da *Libcjp*.

```
16      int valor = Integer.parseInt((String)
17                                      Quantidades.elementAt(pos));
18      valor--;
19      Quantidades.removeElementAt(pos);
20      Quantidades.insertElementAt(Integer.toString(valor),pos);
21      c.save(Items);
22      c.save(Quantidades);
23      c.checkpoint_here();
24      return true;
25    }
26    return false;
27  }
28
29  public void startUp(String[] args)
30  {
31    try
32    {
33      if (c.verificaRecuperacao(args))
34      {
35        //restaurando os estados
36        c.recover_oockpt();
37        Items=(Vector)c.recover();
38        Quantidades=(Vector)c.recover();
39      }
40      else
41      {
42        ...
43      }
44      ...
45    }catch (Exception e) {e.printStackTrace();}
46  }
47  ...
48 }
```

FIGURA 5.13 – Servidor da aplicação LojaRMI fazendo o uso da *Libcjp* (continuação).

5.3 Avaliação de desempenho

Nesta seção, são apresentados os resultados da avaliação de desempenho através dos experimentos realizados com as aplicações centralizadas descritas na seção 5.1. As aplicações usadas para as questões de avaliação de desempenho da *Libcjp* são: MAT, TDC, SHELL, HEAP e GAUSPIV. Cada aplicação foi executada 100 vezes. A partir dos dados obtidos, foram calculados: a média aritmética (tempo médio para cada *checkpoint*, tamanho médio de cada *checkpoint*, tempo médio da execução da aplicação), o desvio padrão e o intervalo de confiança das informações extraídas para análise.

Como visto no início do capítulo 5, o ambiente de teste para avaliação de desempenho da *Libcjp* foi um microcomputador isolado, com processador *Duron* 850 Mhz, 128 *Mbytes* de RAM e sistema operacional *Linux Mandrake release* 8.0 com *kernel* 2.4.3-20mdk. O motivo da escolha do ambiente *Linux* para efetuar os testes de desempenho é pelo fato de que este sistema operacional se comportou melhor, e se apresentou mais confiável para a coleta dos tempos de execução do mecanismo de *checkpointing* e tempo de execução da aplicação; do que o sistema operacional *Windows*. Para realizar os experimentos, as aplicações utilizadas para teste foram

executadas de forma seqüencial (uma aplicação por vez). A biblioteca, assim como todas as aplicações usadas para os experimentos de avaliação desta, foram implementadas utilizando a linguagem orientada a objetos Java (pacote *jdk1.3.1_01*).

Todas as tabelas desta seção mostram os resultados estatísticos obtidos nos testes efetuados com o uso da biblioteca para todas as aplicações, através dos experimentos de *checkpointing*. A seguir são definidos os principais campos encontrados nestas tabelas:

- **aplicação**: identifica, através de um mnemônico, a aplicação em análise. Mais informações a respeito das aplicações foram apresentadas na seção 5.1;
- **tempo médio de execução (seg)**: média aritmética dos tempos de execução para cada aplicação (expressa em segundos). Cada aplicação foi executada 100 vezes;
- **σx**: exibe os dados referentes aos valores de desvio-padrão. Quando relacionado às aplicações, representa o desvio-padrão das amostras de tempo de execução da aplicação. Quando relacionado ao mecanismo de *checkpointing*, representa o desvio-padrão dos dados extraídos a partir dos tempos decorrentes do mecanismo de *checkpointing*;
- **IC 95%**: largura do intervalo de confiança para 95%. Quando relacionado às aplicações, representa a largura do intervalo de confiança da média das amostras de tempo de execução da aplicação. Quando relacionado ao mecanismo de *checkpointing*, representa a largura do intervalo de confiança da média dos dados extraídos a partir dos tempos decorrentes do mecanismo de *checkpointing*;
- **% sobrecarga**: representa a porcentagem de sobrecarga de processamento causada pela inserção do mecanismo de *checkpointing* (toma por base o tempo de processamento da aplicação original – sem *checkpointing*);
- **intervalo (seg)**: intervalo de tempo decorrido, em segundos, entre cada dois *checkpoints*;
- **Nº arq.**: corresponde ao número de arquivos de *checkpoint* salvos durante a execução da aplicação (cada *checkpoint* gera um arquivo);
- **tempo médio (seg)**: representa o tempo médio do estabelecimento de um *checkpoint* para cada aplicação, utilizando o mecanismo não incremental;
- **tamanho médio (*Kbytes*)**: tamanho médio dos arquivos de *checkpoint* expressos em *Kbytes*.

Na tabela 5.2, são apresentados os tempos de execução de cada aplicação sem fazer o uso da biblioteca. Os valores de tempo desta tabela foram obtidos a partir da execução de cada aplicação, em operação normal, sem a interferência de qualquer mecanismo de *checkpointing*.

TABELA 5.2 – Tempos de execução das aplicações sem mecanismo de *checkpointing*.

aplicação	tempo médio de execução (seg)	σ_x	IC 95%
MAT	106,5350	0,1849	0,0725
TDC	33,7787	0,6231	0,2443
SHELL	368,9958	3,8265	1,5000
HEAP	598,6225	4,1880	1,6417
GAUSPIV	21,0666	0,2772	0,1087

A tabela 5.3 reúne os principais dados referentes ao impacto do mecanismo de *checkpointing* não incremental, quando é utilizada a biblioteca em cada uma das aplicações, comparativamente aos tempos de execução normal (sem a ocorrência de falhas). O intuito não é o de comparar os resultados das diversas aplicações, mas o de analisar cada linha independentemente.

TABELA 5.3 – Impacto do mecanismo de *checkpointing* para as aplicações.

aplicação	TME s/ *checkpointing*	c/ *checkpointing*					
		TME	σ_x	IC 95%	% sobrecarga	intervalo (seg)	Nº arq.
MAT	106,5350	127,4311	1,2447	0,4879	19,6143	10	10
TDC	33,7787	57,7184	1,1622	0,4556	70,8720	0,5	5
SHELL	368,9958	390,8345	9,1609	3,5911	5,9184	30	12
HEAP	598,6225	636,0603	1,9628	0,7694	6,2540	30	20
GAUSPIV	21,0666	21,8360	0,3220	0,1262	3,6510	2	7

TME = tempo médio de execução (expresso em segundos)

O intervalo mínimo decorrido entre cada dois *checkpoints*, intervalo (seg), apresenta valores diferentes para as diversas aplicações. Isto é uma conseqüência da forma pela qual foram inseridas as chamadas para as atividades de *checkpointing*, normalmente internas aos laços de procedimentos iterativos, e dos tempos originais de execução dessas aplicações (estes tempos de execução são mostrados no campo: TME). Assim, a porcentagem de sobrecarga de processamento de cada aplicação é particular a esta, apresenta valores relacionados com o número de vezes que foram estabelecidos os *checkpoints* e também com o volume dos dados que foram salvos. Considerou-se que os valores obtidos para o desvio padrão (σ_x) e a largura do intervalo de confiança (IC 95%) das amostras de tempo de execução da aplicação apresentaram resultados baixos, que permitem obter conclusões estatísticas dos experimentos realizados. A aplicação TDC apresenta uma porcentagem de sobrecarga bastante elevada, devido ao custo computacional despendido para o salvamento, no meio estável, do grande volume de dados (estados dos objetos da aplicação) operado por ela. Este custo computacional é constituído principalmente pela atividade de serialização (converte a representação de um objeto na memória para uma seqüência de *bytes*) dos estados dos objetos da aplicação para posterior armazenamento.

Para demonstrar como é tratado pela *Libcjp* o intervalo mínimo decorrido entre cada dois *checkpoints* durante a execução da aplicação, a figura 5.14 apresenta um exemplo de uma suposta aplicação *X*, onde os *checkpoints* foram estabelecidos a cada 10 segundos e o tempo despendido para a atividade de *checkpointing* foi de aproximadamente 2 segundos. Este intervalo mínimo de tempo entre dois *checkpoints* consecutivos é medido apenas durante a execução útil da aplicação; cessa durante o *checkpointing* e só é retomado após a conclusão desta atividade.

Na suposta aplicação *X* da figura 5.14, observa-se que esta possui, ao início da execução, um gasto computacional para sua inicialização e, ao seu término, um gasto computacional para sua finalização. O processo inicial compreende, para muitas aplicações computacionais, a inicialização de variáveis e estruturas de controle, assim como a obtenção de dados externos provenientes da leitura de arquivos do meio de armazenamento. Já o processo de finalização da aplicação, compreende a exibição ou salvamento no meio de armazenamento dos resultados.

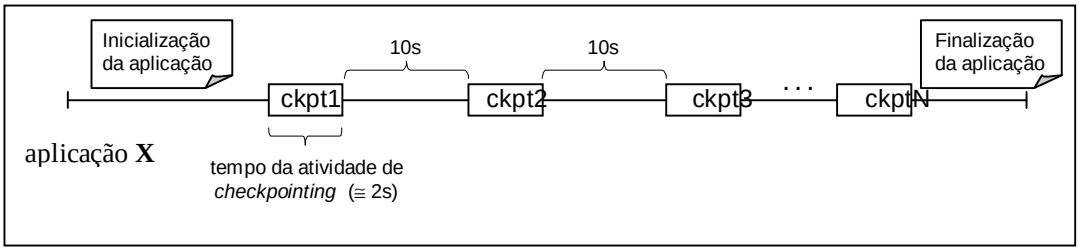

FIGURA 5.14 – Exemplo de como é medido o intervalo entre dois *checkpoints*.

O exemplo da figura 5.14 também pretende ilustrar que, no momento da atividade de inicialização, não é usada a biblioteca com a execução do mecanismo de *checkpointing*. Supostamente foram especificados pontos no código-fonte da aplicação onde é mais vantajosa a ocorrência do *checkpointing*. Esta é uma observação importante para não gerar interpretações erradas a respeito dos resultados apresentados. Por exemplo, sem estas informações, se fosse feita uma estimativa preliminar de quantos arquivos de *checkpoint* a aplicação GAUSPIV proveria durante a execução, possivelmente o número calculado seria aproximadamente de 10 arquivos, a partir do tempo de execução (21,0666 s) dividido pelo intervalo (2 s) . Entretanto, devido aos fatos mencionados anteriormente, ou seja, a aplicação GAUSPIV (a descrição das características desta aplicação consta da subseção 5.1.5) possui uma inicialização que demanda um determinado custo computacional, e conseqüentemente um determinado tempo para esta atividade, o que resultou nos 7 arquivos.

A tabela 5.4 mostra os tempos médios referentes às medidas da atividade e do volume de dados de cada *checkpointing*. Nesta tabela fica nítida a relação entre a quantidade média de dados a serem salvos e o tempo necessário para o *checkpointing*.

TABELA 5.4 – Resultado das medidas da atividade de cada *checkpointing*.

aplicação	tempo médio (seg)	σ_x	IC 95%	tamanho médio (*Kbytes*)
MAT	1,9877	1,2358	0,1532	2034,90
TDC	4,8379	0,9529	0,1670	5352,17
SHELL	1,6922	0,0909	0,0103	976,91
HEAP	1,4548	0,0612	0,00536	976,87
GAUSPIV	0,0350	0,0407	0,00564	13,87

Nas seções de análise individual de cada aplicação que seguem, serão mostrados os valores médios do tamanho dos *checkpoints*, diferenciados pela posição que ocuparam na seqüência de estabelecimento, em cada experimento (em algumas aplicações, o tamanho dos arquivos é variável, ao longo da execução). Estes valores possibilitarão observar o volume de dados salvos em cada arquivo de *checkpoint*, o que não foi possível na tabela 5.4.

A tabela 5.5, mostra a normalização dos dados referentes à sobrecarga de processamento. Os valores apresentados correspondem à porcentagem de sobrecarga por *checkpoint* em cada aplicação. Comparativamente, observa-se o acréscimo de esforço computacional causado pelo mecanismo de *checkpointing* imposto a cada aplicação.

TABELA 5.5 – Normalização da sobrecarga com a atividade de *checkpointing*.

aplicação	% sobrecarga de um *ckpt.*
MAT	1,9614
TDC	14,1744
SHELL	0,4932
HEAP	0,3127
GAUSPIV	0,5216

O valor de sobrecarga individual para o estabelecimento de *checkpoint* na aplicação TDC apresenta um valor expressivamente superior aos demais. O alto valor de sobrecarga de processamento para a aplicação também havia sido observado na tabela 5.3. Na subseção 5.3.2, que trata da análise da aplicação TDC, poderão ser encontrados maiores esclarecimentos sobre esta questão.

Estes tempos foram obtidos com o uso da própria *Libcjp*. A biblioteca possui uma classe denominada de **`EstatisticaCkpt`** (subseção 4.2.3), composta por métodos que são executados através da invocação pela própria biblioteca, e são responsáveis por marcar os tempos de execução de cada *checkpointing* e também da execução total da aplicação.

No anexo 3, poderão ser encontradas breves explicações da metodologia e dos cálculos estatísticos utilizados nas tabelas desta seção. As subseções que seguem mostram a análise estatística de cada aplicação.

5.3.1 Análise da aplicação MAT

Neste primeiro experimento utilizando a aplicação MAT, em cada arquivo de *checkpoint* foi salva apenas a matriz resultante do produto, conforme mostrado na figura 5.8. Como o objeto-matriz resultante *m3* do produto sempre é modificado, optou-se por utilizar para este experimento apenas o mecanismo de *checkpointing* não incremental. Ou seja, não haveria vantagem na escolha do mecanismo incremental. A configuração do arquivo de parâmetros da *Libcjp* utilizada para este experimento foi definida como mostrado na figura 5.15. Explicações referentes ao arquivo de parâmetros podem ser encontradas na subseção 4.2.5. Com os testes realizados, para salvar os 10 arquivos de *checkpoint* durante a execução da aplicação, foi medida uma sobrecarga de processamento de 19,614%.

```
1  intervalo=10
2  tipo_intervalo=S
3  num_max_arquivos=0
4  incremental=N
5  diretorio=.
```

FIGURA 5.15 – Configuração dos parâmetros para a aplicação MAT.

Vale ressaltar que a configuração da biblioteca pode influenciar muito no controle do tempo despendido para as atividades de execução do mecanismo de *checkpointing*. Quando se diminui o tempo entre dois *checkpoints* consecutivos, aumenta o número de arquivos salvos no meio de armazenamento. Aumentando este tempo, obtém-se o efeito contrário. É de conhecimento que, quanto mais intensa for a atividade de *checkpointing*, maior será a sobrecarga de processamento da aplicação. O volume de dados (estados

dos objetos) também é outro fator que influencia na sobrecarga de processamento na execução da aplicação.

A tabela 5.6 mostra o tamanho médio dos arquivos de *checkpoint* resultantes da utilização do mecanismo de *checkpointing* para as 100 execuções da aplicação MAT. Na primeira coluna da tabela, encontra-se a representação dos 10 arquivos de *checkpoint* (*C1*,..*C10*) salvos durante a execução da aplicação; na segunda coluna, o tamanho médio de cada arquivo de *checkpoint* considerando as 100 execuções realizadas neste experimento com o mecanismo de *checkpointing* não incremental.

TABELA 5.6 – Tamanho médio dos *checkpoints* para a aplicação MAT.

	tamanho médio *checkpoint* (*Kbytes*)
C1	378,87
C2	751,80
C3	1120,30
C4	1478,70
C5	1855,99
C6	2221,50
C7	2588,79
C8	2959,64
C9	3317,47
C10	3675,93

Os dados resultantes das execuções da aplicação MAT para este experimento são apresentados, a seguir, na forma de tabelas de freqüência (fi) e histogramas correspondentes a estas tabelas. Os diversos histogramas utilizam escalas e valores diferentes, em cada representação. Assim, não é recomendada uma comparação direta entre os seus dados.

Primeiramente, são apresentados os valores dos tempos decorrentes da atividade de salvamento dos arquivos de *checkpoint*. A figura 5.16 mostra a tabela de freqüência e o histograma dos tempos médios (em milisegundos) dos *checkpoints* referentes a cada uma das 100 execuções. Cada execução da aplicação produziu um valor, representando a média dos tempos despendidos para o salvamento dos *checkpoints*.

	Classes		fi
1	1870	a 1908	2
2	1908	a 1946	21
3	1946	a 1984	26
4	1984	a 2022	25
5	2022	a 2060	18
6	2060	a 2098	6
7	2098	a 2136	2
			Σ = 100

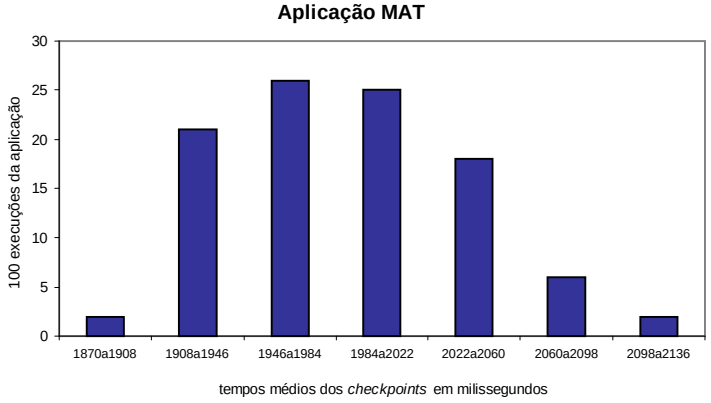

FIGURA 5.16 – Histograma dos tempos médios dos *checkpoints* (aplicação MAT).

Em seguida, são apresentados os dados correspondentes às amostras de tempo obtidas com as 100 execuções da aplicação sem *checkpointing* (sem fazer uso da *Libcjp*) e com *checkpointing*.

A figura 5.17 mostra a tabela de freqüência e o histograma dos tempos das execuções da aplicação MAT sem fazer uso da *Libcjp*.

	Classes	fi
1	106000 a 106190	1
2	106190 a 106380	14
3	106380 a 106570	53
4	106570 a 106760	21
5	106760 a 106950	8
6	106950 a 107140	2
7	107140 a 107330	1
	Σ =	100

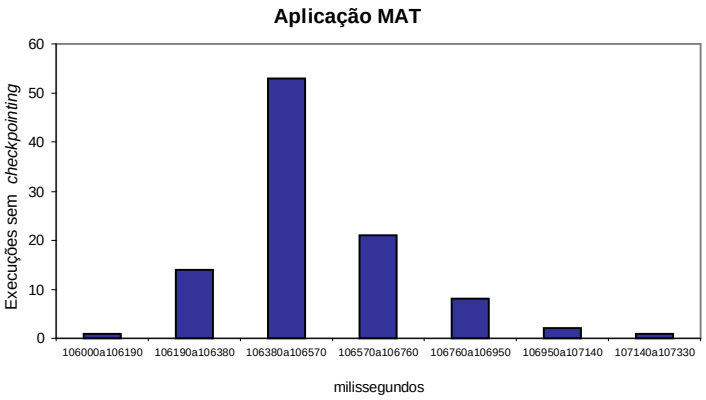

FIGURA 5.17 – Histograma dos tempos de execução de MAT sem *checkpointing*.

A figura 5.18 mostra a tabela de freqüência e o histograma dos tempos das execuções da aplicação MAT com o uso do mecanismo de *checkpointing* não incremental.

	Classes	fi
1	125000 a 126300	13
2	126300 a 127600	51
3	127600 a 128900	26
4	128900 a 130200	6
5	130200 a 131500	2
6	131500 a 132800	1
7	132800 a 134100	1
	Σ =	100

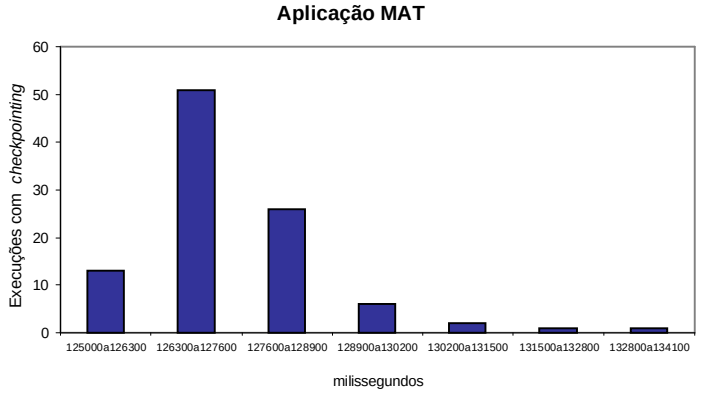

FIGURA 5.18 – Histograma dos tempos de execução de MAT com *checkpointing*.

5.3.2 Análise da aplicação TDC

Especificamente neste experimento envolvendo a aplicação TDC foram utilizados os dois mecanismos de *checkpointing*: não incremental e incremental. A aplicação TDC utiliza dois objetos do tipo matriz: uma matriz-origem (**`MatImgOriginal`**) e uma matriz resultante dos cálculos da transformada discreta do cosseno (**`MatTDC`**). A aplicação TDC modifica constantemente o objeto da matriz resultante durante a sua execução, já o objeto da matriz-origem permanece inalterado. Os arquivos de

checkpoint são compostos pelos estados dos objetos matrizes e por algumas informações de controle da aplicação. Observa-se que são salvos nos arquivos de *checkpoint* dados desnecessários provenientes da matriz-origem. Isto somente foi feito para se ter dados inalterados durante a execução da aplicação e assim poder testar o mecanismo de *checkpointing* incremental. A descrição desta aplicação poderá ser encontrada na subseção 5.1.2 e o código-fonte no anexo 2.2.

Foram feitas duas configurações diferentes para o arquivo de parâmetros da *Libcjp* utilizada para este experimento, uma para o mecanismo não incremental (figura 5.19) e outra para o mecanismo incremental (figura 5.20).

```
1  intervalo=500
2  tipo_intervalo=L
3  num_max_arquivos=0
4  incremental=N
5  diretorio=.
```

FIGURA 5.19 – Configuração dos parâmetros para TDC (modo não incremental).

```
1  intervalo=500
2  tipo_intervalo=L
3  num_max_arquivos=0
4  incremental=S
5  diretorio=.
```

FIGURA 5.20 – Configuração dos parâmetros para TDC (*checkpointing* incremental).

A tabela 5.7 demonstra comparativamente o impacto dos mecanismos de *checkpointing* não incremental e incremental, quando é utilizada a biblioteca, na aplicação TDC.

TABELA 5.7 – Impacto dos mecanismos de *checkpointing* para TDC.

TME s/ *checkpointing*	c/ *checkpointing*						
	checkpointing	TME	σ_x	IC 95%	% sobrecarga	intervalo (seg)	Nº arq.
33,7787	não incremental	57,7184	1,1622	0,4556	70,8720	0,5	5
	incremental	88,4417	1,1201	0,4391	161,8267	0,5	5

TME = tempo médio de execução (expresso em segundos)

Observa-se na tabela 5.7 uma porcentagem de sobrecarga de processamento alta, resultante do grande volume de dados salvos nos arquivos de *checkpoint* e da forma de implementação escolhida para o mecanismo incremental.

A tabela 5.8 mostra comparativamente os tempos médios referentes às medidas da atividade e do volume de dados de cada *checkpointing* nos dois mecanismos: não incremental e incremental. Além de permitir observar a relação entre a quantidade de dados a serem salvos e o tempo necessário para o *checkpointing*, esta tabela também possibilita a comparação entre os dados obtidos com a utilização dos dois mecanismos.

TABELA5.8 – Resultado das medidas da atividade de cada *checkpointing* nos dois mecanismos: não incremental e incremental (aplicação TDC)

checkpointing	tempo médio (seg)	σ_x	IC 95%	tamanho médio (*Kbytes*)
não incremental	4,8379	0,9529	0,1670	5352,17
incremental	11,2469	2,9033	0,5090	2462,31

Analisando a tabela 5.8, observa-se que, quando foi utilizado o mecanismo de *checkpointing* incremental, houve uma redução no tamanho médio dos arquivos de *checkpoint* se comparado ao mecanismo não incremental. Utilizando o mecanismo não incremental obteve-se um tamanho médio dos arquivos de *checkpoint* de 5352,17 *Kbytes*, já utilizando o mecanismo incremental obteve-se um tamanho médio de 2462,31 *Kbytes*, o que representa uma redução média de 2889,86 *Kbytes* (maior que 50%, portanto).

A tabela 5.9 mostra o tamanho médio dos arquivos de *checkpoint* para os dois mecanismos utilizados neste experimento. Na primeira coluna da tabela, encontra-se a identificação dos 5 arquivos de *checkpoint* (*C1*,..*C5*) salvos durante a execução da aplicação; na segunda coluna, figura o tamanho médio dos arquivos de *checkpoint*, considerando as 100 execuções realizadas para esta aplicação, utilizando o mecanismo de *checkpointing* não incremental; a terceira coluna é análoga à segunda, mas refere-se ao mecanismo incremental.

TABELA 5.9 – Tamanho médio dos *checkpoints* da aplicação TDC.

	não incremental (*Kbytes*)	incremental (*Kbytes*)
C1	3851,34	3851,34
C2	4490,21	896,24
C3	5276,00	1713,28
C4	6140,73	2550,53
C5	7002,58	3300,15

A figura 5.21 mostra um gráfico comparativo com os dados da tabela 5.9, construído a partir dos tamanhos dos arquivos de *checkpoint* obtidos com os dois mecanismos: não incremental e incremental. Observou-se que os tamanhos dos arquivos obtidos através do mecanismo não incremental mantiveram-se constantes e crescentes em virtude das características da aplicação. Entretanto, comparando o mecanismo incremental com o mecanismo não incremental, observou-se o mesmo tamanho para o *checkpoint C1*; uma grande diminuição de tamanho do *checkpoint C2* no mecanismo incremental; e a partir deste ponto, observou-se um aumento dos *checkpoints* no mecanismo incremental, mantendo-se aproximadamente constante a diferença com relação ao mecanismo não incremental. Essa diminuição de tamanho do *checkpoint C2* e dos seus sucessores foi por ser desnecessário o salvamento do objeto matriz **`MatImgOriginal`** (origem), que se manteve inalterado durante a execução da aplicação.

Com relação ao tempo de salvamento dos arquivos de *checkpoint*, observou-se um tempo maior quando é utilizado o mecanismo de *checkpointing* incremental do que com o uso do mecanismo não incremental.

A biblioteca *Libcjp* possui um objeto contêiner em memória, que armazena, durante a execução da aplicação, os estados mais atuais dos objetos, em forma serializada, para posteriores comparações com os próximos estados dos objetos (maiores detalhes foram comentados na subseção 4.2.2 – classe **`Checkpointing`**). Esta foi a forma adotada para determinar se o estado de um objeto foi modificado.

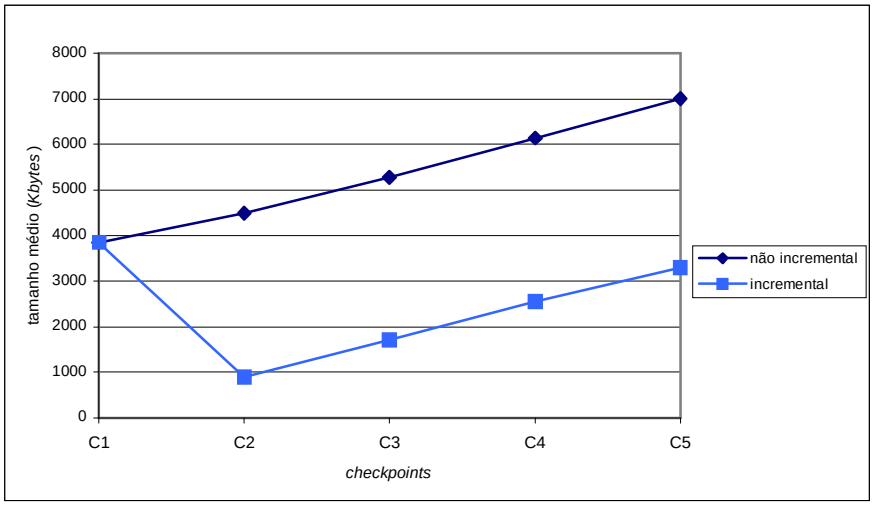

FIGURA 5.21 – Comparação dos tamanhos dos *checkpoints* nos dois mecanismos de *checkpointing*: não incremental e incremental para a aplicação TDC.

O estado do objeto candidato a ser salvo no arquivo de *checkpoint* é transformado para a forma serializada e comparado com o seu respectivo estado no objeto contêiner. Quando estes estados são diferentes, identifica-se que o objeto foi modificado e então este será salvo no arquivo de *checkpoint* atual. Estas atividades adicionais de serializar o estado do objeto e realizar a comparação com o estado anterior mantido em memória resultam em um aumento significativo de tempo do mecanismo de *checkpointing* incremental em relação ao não incremental.

A tabela 5.10 mostra o tempo médio gasto para a atividade de *checkpointing* utilizando os mecanismos não incremental e incremental para as 100 execuções da aplicação TDC. Na primeira coluna da tabela encontra-se a identificação dos 5 arquivos de *checkpoint* (*C1*,..*C5*) salvos durante a execução da aplicação; na segunda, o tamanho médio dos tempos dos *checkpoints* considerando as 100 execuções realizadas neste experimento, utilizando o mecanismo de *checkpointing* não incremental; e, na terceira, as mesmas informações da segunda, mas referentes ao mecanismo incremental.

TABELA 5.10 – Tempo médio dos *checkpoints* utilizando os mecanismos não incremental e incremental para as 100 execuções da aplicação TDC.

	não incremental (milissegundos)	incremental (milissegundos)
C1	3550,4600	12280,1200
C2	3806,5500	6584,1600
C3	5542,9400	9569,1400
C4	5633,2700	13077,1000
C5	5656,2200	14723,9000

A figura 5.22 apresenta um gráfico comparativo referente aos dados da tabela 5.10, para a aplicação TDC, mostrando os tempos gastos para o salvamento dos arquivos de *checkpoint* através dos dois mecanismos: não incremental e incremental.

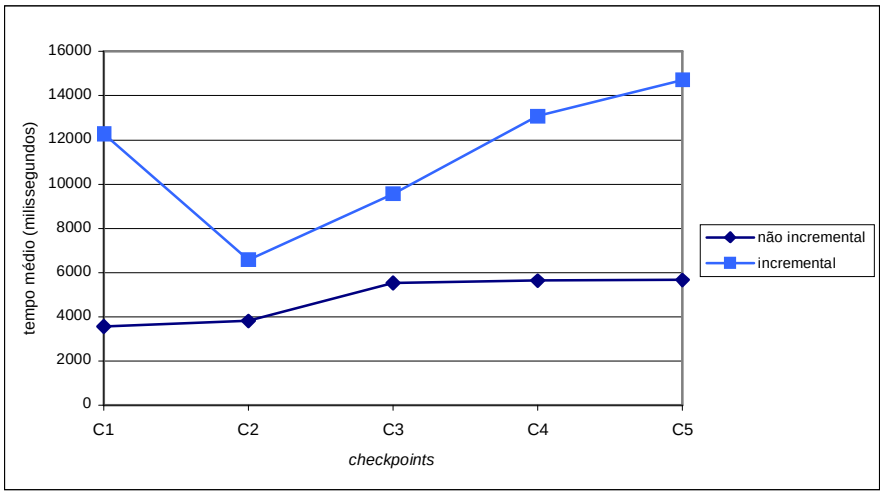

FIGURA 5.22 – Comparação dos tempos obtidos pelos mecanismos de *checkpointing* não incremental e incremental para a aplicação TDC.

Assim como ocorreu com a análise da aplicação MAT (subseção 5.3.1), os dados resultantes das execuções da aplicação TDC são apresentados na forma de tabelas de freqüência (fi) e histogramas correspondentes.

Nas duas figuras seguintes, são apresentados os valores dos tempos decorrentes do *checkpointing*, utilizando ambos mecanismos: não incremental e incremental. A figura 5.23 mostra a tabela de freqüência e o histograma dos tempos médios dos *checkpoints* referentes a cada uma das 100 execuções, utilizando o mecanismo não incremental. Cada execução da aplicação produziu um único valor, representando a média dos tempos despendidos para o salvamento dos *checkpoints*.

	Classes		fi
1	4750	a 4780	2
2	4780	a 4810	7
3	4810	a 4840	61
4	4840	a 4870	11
5	4870	a 4900	8
6	4900	a 4930	7
7	4930	a 4960	4

Σ = 100

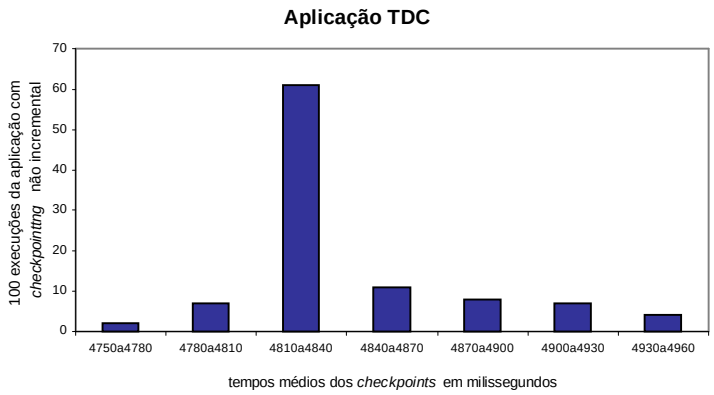

FIGURA 5.23 – Histograma dos tempos médios do *checkpointing* com o mecanismo não incremental (aplicação TDC).

A figura 5.24 mostra a tabela de freqüência e o histograma dos tempos médios dos *checkpoints* referentes a cada uma das 100 execuções, utilizando o mecanismo incremental.

	Classes	fi
1	10800 a 10940	12
2	10940 a 11080	13
3	11080 a 11220	20
4	11220 a 11360	21
5	11360 a 11500	21
6	11500 a 11640	9
7	11640 a 11780	4

Σ = 100

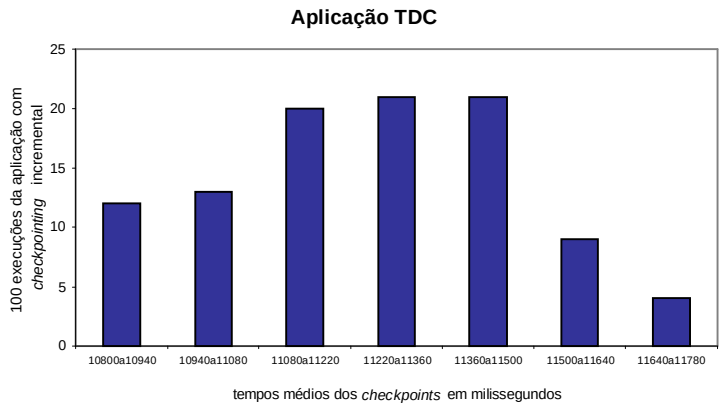

FIGURA 5.24 – Histograma dos tempos médios do *checkpointing* com o mecanismo incremental (aplicação TDC).

Nas três figuras restantes desta subseção, são apresentados os dados correspondentes às amostras de tempo obtidas com as 100 execuções da aplicação, sem fazer uso da *Libcjp*, e com o uso dos mecanismos de *checkpointing* não incremental e incremental.

A figura 5.25 mostra a tabela de freqüência e o histograma dos tempos das execuções da aplicação TDC, sem fazer o uso da *Libcjp*.

	Classes	fi
1	32500 a 32950	9
2	32950 a 33400	18
3	33400 a 33850	30
4	33850 a 34300	23
5	34300 a 34750	16
6	34750 a 35200	2
7	35200 a 35650	2

Σ = 100

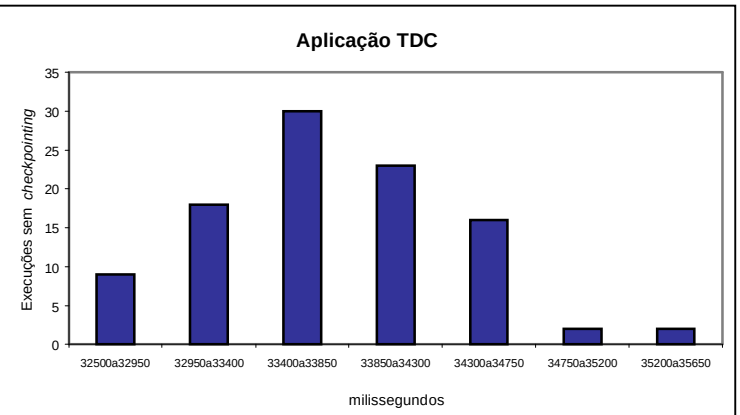

FIGURA 5.25 – Histograma dos tempos de execução da aplicação TDC sem *checkpointing*.

A figura 5.26 mostra a tabela de freqüência e o histograma dos tempos das execuções da aplicação TDC, com o uso do mecanismo de *checkpointing* não incremental.

	Classes	fi
1	55500 a 56240	13
2	56240 a 56980	14
3	56980 a 57720	19
4	57720 a 58460	24
5	58460 a 59200	22
6	59200 a 59940	6
7	59940 a 60680	2

Σ = 100

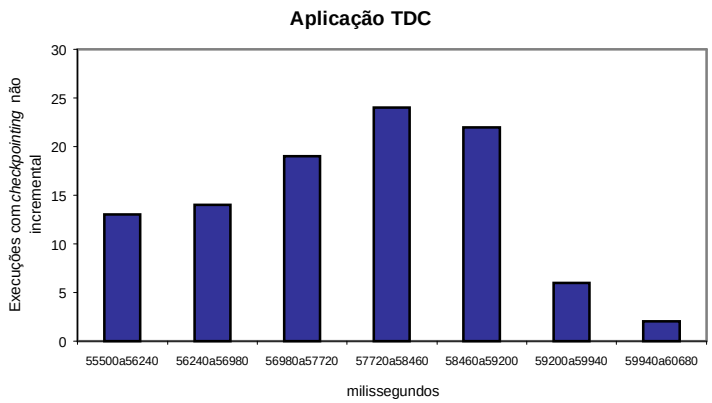

FIGURA 5.26 – Histograma dos tempos de execução da aplicação TDC com *checkpointing* não incremental.

A figura 5.27 mostra a tabela de freqüência e o histograma dos tempos das execuções da aplicação TDC com o uso do mecanismo de *checkpointing* incremental.

	Classes	fi
1	86000 a 86870	7
2	86870 a 87740	21
3	87740 a 88610	22
4	88610 a 89480	41
5	89480 a 90350	4
6	90350 a 91220	3
7	91220 a 92090	2

Σ = 100

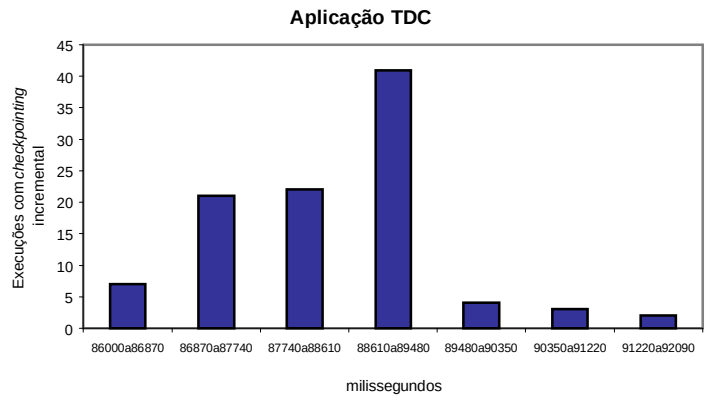

FIGURA 5.27 – Histograma dos tempos de execução da aplicação TDC com *checkpointing* incremental.

5.3.3 Análise da aplicação SHELL

Para esta aplicação, a tabela de freqüência e o histograma correspondente foram construídos levando em consideração todos os tempos despendidos para o salvamento dos arquivos de *checkpoint* e não a média dos tempos por arquivo de *checkpoint* com as 100 execuções da aplicação. Nesta aplicação, foram salvos 12 arquivos de *checkpoint,* o que resulta num número de 1200 amostras de tempo. Observou-se que os tamanhos dos arquivos de *checkpoint* permaneceram constantes em todas as execuções da aplicação. Cada arquivo de *checkpoint* tinha o tamanho correspondente a 976,91 *Kbytes*. Este fato ocorreu devido à característica própria da aplicação, onde apenas ocorre a troca de posição dos elementos em um vetor de elementos inteiros. Como não há geração ou exclusão de dados, seu volume permanece inalterado, o que explica a igualdade dos tamanhos dos arquivos de *checkpoint.*

A configuração do arquivo de parâmetros da *Libcjp* utilizada para este experimento da aplicação SHELL foi definida como mostrado na figura 5.28.

```
1  intervalo=30
2  tipo_intervalo=S
3  num_max_arquivos=0
4  incremental=N
5  diretorio=.
```

FIGURA 5.28 – Configuração da *Libcjp* para a aplicação SHELL.

A figura 5.29 mostra a tabela de freqüência e o histograma dos tempos das 1200 amostras de tempo dos *checkpoints* correspondentes às execuções da aplicação SHELL.

	Classes	fi
1	1500 a 1570	171
2	1570 a 1640	199
3	1640 a 1710	259
4	1710 a 1780	389
5	1780 a 1850	135
6	1850 a 1920	45
7	1920 a 1990	2

Σ = 1200

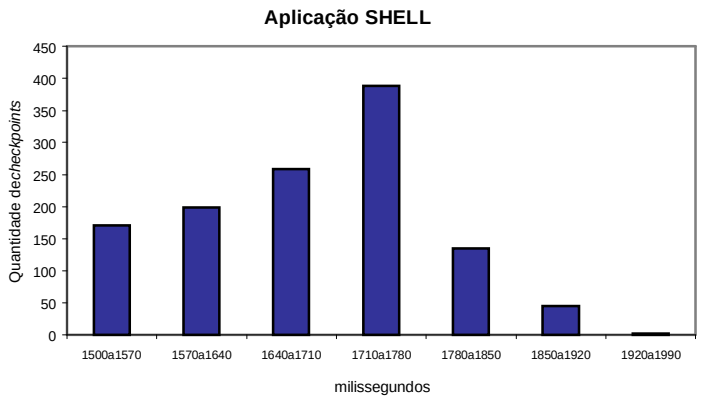

FIGURA 5.29 – Histograma dos tempos dos *checkpoints* (aplicação SHELL).

A seguir, são apresentados os dados correspondentes às amostras de tempo obtidas com as 100 execuções da aplicação sem *checkpointing* (sem fazer uso da *Libcjp*) e com *checkpointing*. A figura 5.30 mostra a tabela de freqüência e o histograma dos tempos das execuções da aplicação sem fazer o uso da *Libcjp*, enquanto a figura 5.31 mostra a tabela com os dados resultantes do uso do mecanismo de *checkpointing* não incremental.

	Classes	fi
1	353200 a 356600	1
2	356600 a 360000	2
3	360000 a 363400	9
4	363400 a 366800	14
5	366800 a 370200	15
6	370200 a 373600	58
7	373600 a 377000	1

Σ = 100

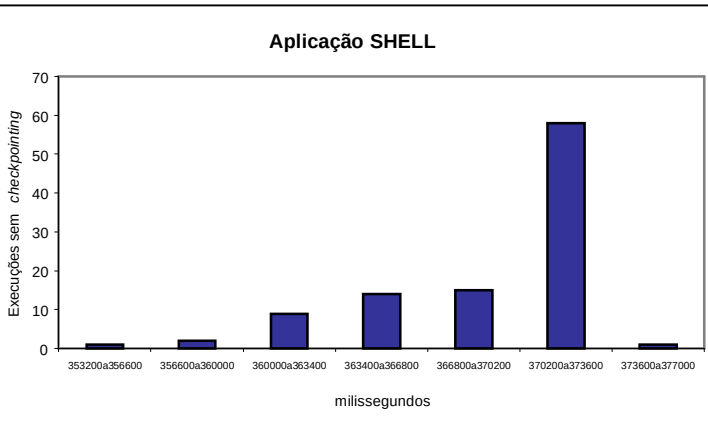

FIGURA 5.30 – Histograma de tempos de execução da aplicação SHELL sem *checkpoints*.

	Classes	fi
1	351000 a 361500	1
2	361500 a 372000	3
3	372000 a 382500	8
4	382500 a 393000	60
5	393000 a 403500	23
6	403500 a 414000	4
7	414000 a 424500	1
		Σ = 100

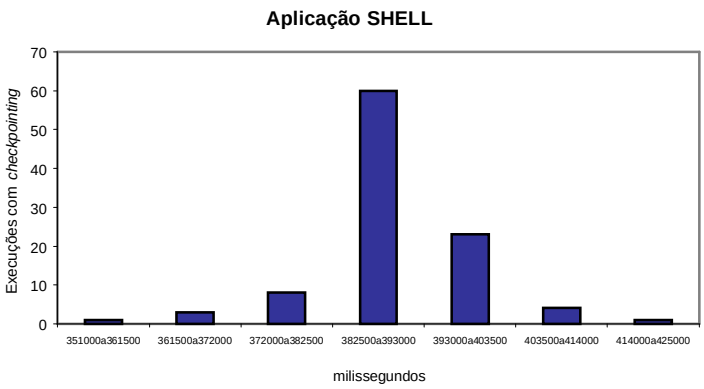

FIGURA 5.31 – Histograma de tempos de execução da aplicação SHELL c/ *checkpoints*.

5.3.4 Análise da aplicação HEAP

Para esta aplicação, de forma análoga à demonstração dos resultados da aplicação SHELL, a tabela de freqüência e o histograma correspondente foram construídos levando em consideração todos os tempos despendidos para o estabelecimento dos arquivos de *checkpoint* e não a média dos tempos por arquivo de *checkpoint*. Neste experimento também ocorreu a igualdade dos tamanhos dos arquivos de *checkpoint* em todas as execuções da aplicação. Nesta aplicação, foram salvos 20 arquivos de *checkpoint* por execução, o que resulta num número de 2000 amostras de tempo. O tamanho de cada arquivo de *checkpoint* foi de 976,87 *Kbytes*.

A configuração do arquivo de parâmetros da *Libcjp* definida para este experimento da aplicação HEAP é mostrada na figura 5.32.

```
1  intervalo=30
2  tipo_intervalo=S
3  num_max_arquivos=0
4  incremental=N
5  diretorio=.
```

FIGURA 5.32 – Configuração da *Libcjp* para a aplicação HEAP.

A figura 5.33 mostra a tabela de freqüência e o histograma dos tempos dos 2000 *checkpoints* correspondentes às execuções da aplicação HEAP.

	Classes	fi
1	1330 a 1375	77
2	1375 a 1420	532
3	1420 a 1465	695
4	1465 a 1510	364
5	1510 a 1555	152
6	1555 a 1600	104
7	1600 a 1645	76

Σ = 2000

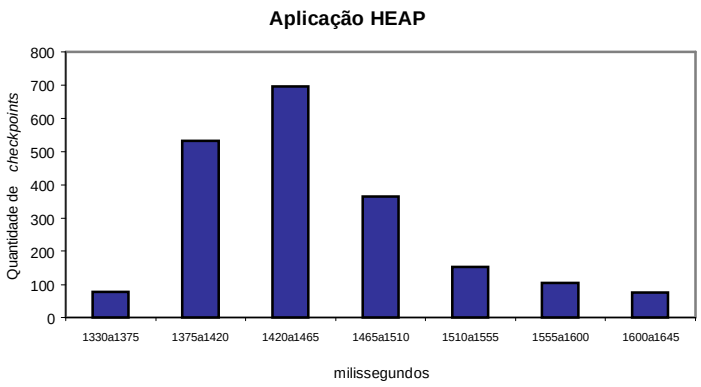

FIGURA 5.33 – Histograma dos tempos dos *checkpoints* (aplicação HEAP).

As figuras seguintes apresentam os dados correspondentes às amostras de tempo obtidas com as 100 execuções da aplicação sem e com *checkpointing*.

A figura 5.34 mostra a tabela de freqüência e o histograma dos tempos das execuções da aplicação sem fazer o uso da *Libcjp*, ou seja, sem estabelecer *checkpoints.*

	Classes	fi
1	587000 a 590500	3
2	590500 a 594000	14
3	594000 a 597500	18
4	597500 a 601000	46
5	601000 a 604500	12
6	604500 a 608000	4
7	608000 a 611500	3

Σ = 100

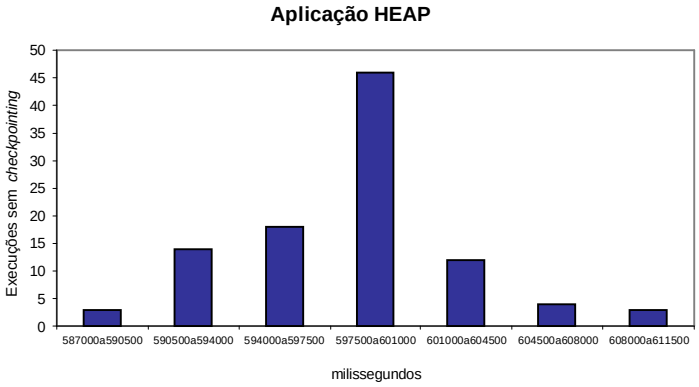

FIGURA 5.34 – Histograma dos tempos de execução de HEAP sem *checkpointing*.

A figura 5.35 mostra a tabela de freqüência e o histograma dos tempos das execuções da aplicação com o uso do mecanismo de *checkpointing* não incremental.

	Classes	fi
1	623000 a 626000	1
2	626000 a 629000	1
3	629000 a 632000	2
4	632000 a 635000	7
5	635000 a 638000	84
6	638000 a 641000	4
7	641000 a 644000	1

Σ = 100

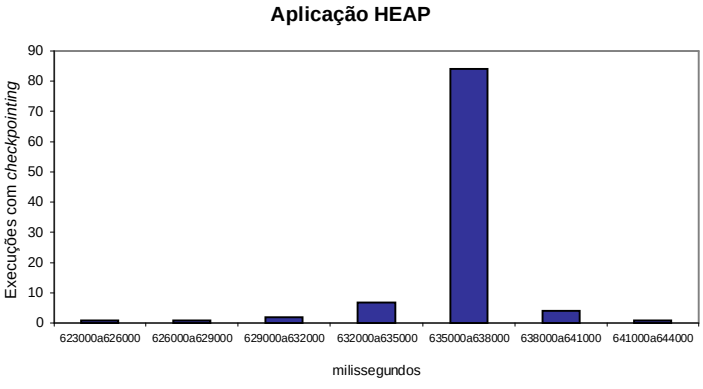

FIGURA 5.35 – Histograma dos tempos da aplicação HEAP com *checkpointing*.

5.3.5 Análise da aplicação GAUSPIV

Neste experimento, envolvendo a aplicação GAUSPIV, foram salvos em cada arquivo de *checkpoint*: a matriz dos coeficientes *A*; o vetor dos termos independentes *B*; e, o vetor *C* (resultado do sistema triangular). Observou-se, no experimento, que os tamanhos dos arquivos de *checkpoint* permaneceram constantes em todas as execuções da aplicação, tendo sido gerados 7 arquivos de *checkpoint* durante cada execução, com o tamanho de 13,87 *Kbytes* cada. A configuração do arquivo de parâmetros da *Libcjp* utilizada para este experimento foi definida como mostrado na figura 5.36.

```
1  intervalo=2
2  tipo_intervalo=S
3  num_max_arquivos=0
4  incremental=N
5  diretorio=.
```

FIGURA 5.36 – Configuração da *Libcjp* para a aplicação GAUSPIV.

Na figura 5.37, são apresentados os valores dos tempos decorrentes da atividade de *checkpointing*, através da tabela de freqüência e o histograma dos tempos médios dos *checkpoints* referentes a cada uma das 100 execuções. Cada execução da aplicação GAUSPIV produziu um valor, representando a média dos tempos despendidos para o salvamento dos *checkpoints*.

	Classes	fi
1	33,00 a 33,55	5
2	33,55 a 34,10	6
3	34,10 a 34,65	17
4	34,65 a 35,20	27
5	35,20 a 35,75	25
6	35,75 a 36,30	16
7	36,30 a 36,85	4
	Σ =	100

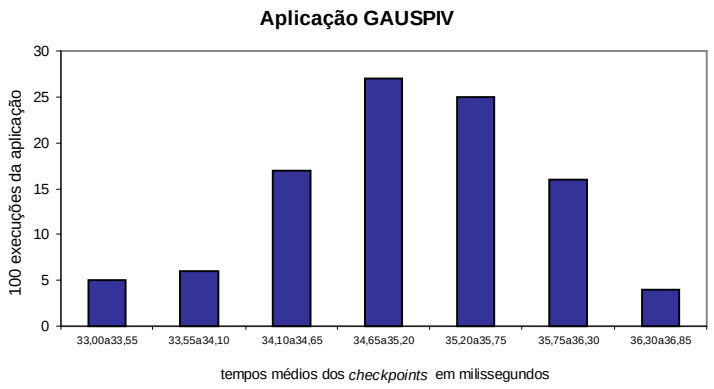

FIGURA 5.37 – Histograma dos tempos médios dos *checkpoints* (aplicação GAUSPIV).

A figura 5.38 mostra a tabela de freqüência e o histograma dos tempos das execuções da aplicação GAUSPIV sem fazer o uso da *Libcjp*.

	Classes	fi
1	20000 a 20270	2
2	20270 a 20540	5
3	20540 a 20810	7
4	20810 a 21080	50
5	21080 a 21350	26
6	21350 a 21620	6
7	21620 a 21890	4

Σ = 100

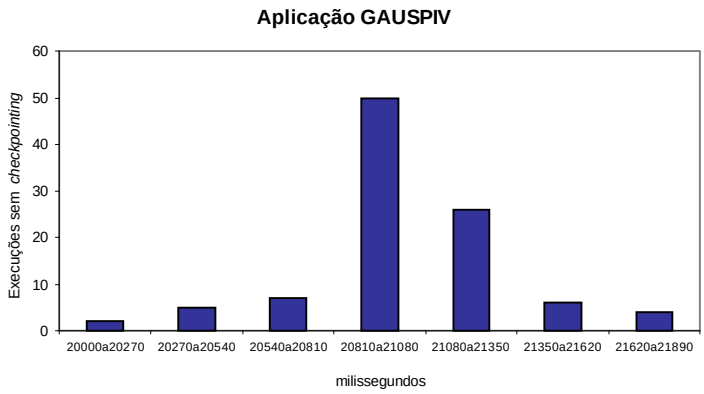

FIGURA 5.38 – Histograma dos tempos de execução de GAUSPIV sem *checkpointing*.

A figura 5.39 mostra a tabela de freqüência e o histograma dos tempos das execuções da aplicação GAUSPIV com o uso do mecanismo de *checkpointing* não incremental.

	Classes	fi
1	21100 a 21390	7
2	21390 a 21680	12
3	21680 a 21970	56
4	21970 a 22260	16
5	22260 a 22550	4
6	22550 a 22840	3
7	22840 a 23130	2

Σ = 100

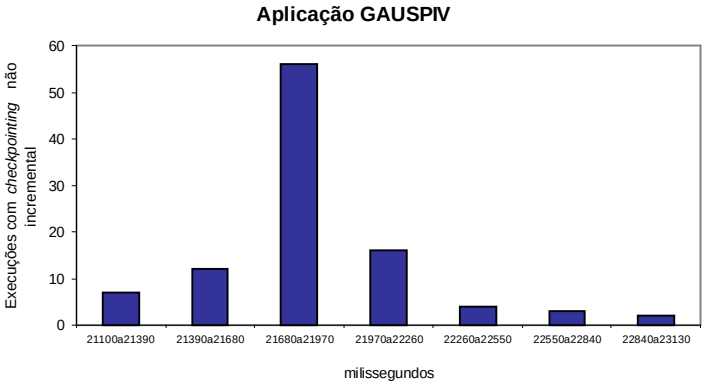

FIGURA 5.39 – Histograma dos tempos de execução da aplicação GAUSPIV com *checkpointing*.

6 Conclusões

Esta dissertação foi motivada pela necessidade de componentes básicos que dessem suporte à implementação de algoritmos de recuperação, que estavam sendo desenvolvidos no Grupo de Tolerância a Falhas da UFRGS. A disponibilidade de mecanismos pré-programados na forma de componentes de uma biblioteca viria a simplificar o trabalho dos programadores para desenvolver aplicações mais confiáveis. Estes mecanismos, se usados adequadamente, podem minimizar as perdas (correspondentes aos tempos de reprocessamento) decorrentes da interrupção de processamento devido a falhas no sistema, além de atenuar o impacto dos defeitos do sistema sobre a disponibilidade da aplicação. Encontra-se algum suporte pronto, na forma de bibliotecas para recuperação, desenvolvido por outros pesquisadores; entretanto, em trabalho realizado preliminarmente [SIL2001] identificaram-se dificuldades de utilização. A biblioteca citada por todas as fontes, quando se busca a funcionalidade de *checkpointing* e recuperação, é a *libckpt*, desenvolvida por Plank [PLA95]. Entretanto, em outro trabalho realizado no grupo [FER2002], testes demonstraram dificuldades ou problemas na execução de procedimentos de recuperação (no *Solaris*) e dificuldades bem mais graves, que iniciam nos procedimentos de *checkpointing*, na versão para *Linux*. Adicionalmente, esta biblioteca visa recuperação de processos independentes e demandaria um esforço expressivo para adequá-la à recuperação em processos concorrentes.

Outra motivação foi o uso de orientação a objetos em muitos dos trabalhos que vêm sendo realizados no Grupo. Ao dispor de mecanismos pré-implementados e testados que fornecem serviços de salvamento de estados e recuperação, e que podem ser utilizados na construção de aplicações, fica mais fácil a realização das tarefas associadas ao desenvolvimento. Esses benefícios são propiciados aos programadores sem obrigá-los a encarregarem-se de detalhes de implementação dos mecanismos.

Neste trabalho, a implementação destes mecanismos constituiu uma biblioteca de *checkpointing* e recuperação. Esta biblioteca, desenvolvida para ser empregada em nível de usuário, foi denominada de *Libcjp – Library of checkpoints in Java programs.* A biblioteca facilita a atividade de estabelecimento de *checkpoints* e retomada do processamento pós-falhas (ou recuperação) de aplicações orientadas a objetos escritas na linguagem de programação Java.

A linguagem Java foi escolhida para a implementação da biblioteca por ser, comparativamente às outras linguagens, mais vantajosa, principalmente devido à portabilidade, à independência de plataforma das aplicações Java, aos recursos disponíveis para persistência e serialização de objetos presentes na API de serialização Java e uso da tecnologia RMI para o cenário de aplicações distribuídas. A persistência e serialização constituíram os principais artifícios utilizados para o salvamento dos estados dos objetos da aplicação. Outro fator de peso na escolha por Java é que ela tem sido bastante utilizada no mercado de *software*. Esta característica particular permite garantir atualizações e correções sobre a linguagem.

Em contrapartida, um fator que tem pesado negativamente sobre a linguagem Java é quanto ao seu desempenho. Entretanto, isto tende a mudar: Reinholtz [REI2000] afirma que o desempenho de Java poderá comparativamente exceder o desempenho de linguagens como C++, através de uma mudança de paradigma do compilador. Esforços estão sendo dirigidos para que seja realizada compilação dinâmica.

As principais funcionalidades implementadas neste trabalho foram:

- o mecanismo de *checkpointing* não incremental, onde a execução da aplicação é intercalada com a atividade de salvamento dos arquivos de *checkpoint* para o meio de armazenamento estável;
- o mecanismo de *checkpointing* incremental, que constituiu uma idéia semelhante ao mecanismo anterior, mas cujo objetivo é de não repetir o salvamento dos objetos que não foram modificados;
- *checkpoint* programado, cuja idéia é a de controlar o tempo mínimo entre o estabelecimento de cada par de *checkpoints* consecutivos. Esta funcionalidade é válida tanto para o mecanismo não incremental como para o incremental;
- implementou-se a técnica tradicional para recuperação, visando efetuar a recomposição de um ou vários objetos, a partir dos estados de-serializados de um arquivo de *checkpoint*, para um estado operacional normal;
- e, finalmente, uma outra funcionalidade desenvolvida foi o controle do número máximo de arquivos de *checkpoint*, que cuida para manter armazenada apenas a quantidade de arquivos estipulada.

A biblioteca *Libcjp* foi desenvolvida com a implementação de um conjunto de classes para capturar o estado de um ou mais objetos de uma aplicação em execução e representá-los no meio de armazenamento estável através do mecanismo de *checkpointing*, como também para permitir o reinício da aplicação, quando da ocorrência de falhas, através do mecanismo de recuperação. Uma das classes é considerada a principal, pois nela estão implementados os mecanismos de *checkpointing* e recuperação, como também todo o controle operacional da biblioteca. Para dar apoio funcional a estes mecanismos, foram implementadas outras classes. Duas dessas classes destinam-se a realizar atividades no que se refere à manipulação das características dos arquivos de *checkpoint* e para fazer o controle de tempo para definir os momentos de execução dos procedimentos de salvamento. Uma outra classe foi desenvolvida para fazer o registro dos tempos gastos para o *checkpointing* e o tempo total de execução da aplicação. Por fim, foram desenvolvidas classes que controlam o comportamento da biblioteca: estas provêm de um arquivo de parâmetros, que é utilizado com a finalidade de configurar a biblioteca para alguns tipos diferentes de utilização.

A biblioteca não isenta o programador de conhecer os mecanismos básicos envolvidos na execução das atividades de *checkpointing* e recuperação, ele precisa configurar alguns parâmetros e introduzir algumas chamadas de métodos na aplicação. A biblioteca simplifica o restante da programação por disponibilizar os mecanismos necessários.

Foram implementadas e descritas aplicações que apresentam diferentes perfis para realizar os testes e avaliação funcional da biblioteca *Libcjp*. Com os resultados obtidos da avaliação de desempenho através dos experimentos realizados, observou-se o comportamento da biblioteca, onde se podem constatar diferentes valores quando os parâmetros de uso (tipo de mecanismo de *checkpointing*, volume de dados, etc...) variaram através da parametrização adequada. A má configuração de parâmetros ou a inserção de chamadas em pontos inadequados terá reflexos importantes sobre a sobrecarga causada e a qualidade dos resultados finais.

Os benefícios fornecidos ao programador trouxeram a constatação de um problema de desempenho com relação ao mecanismo de *checkpointing* incremental,

implementado neste trabalho, notadamente quando ele é comparado com o mecanismo não incremental. Este fato pôde ser observado na seção de avaliação de desempenho (seção 5.3), especialmente na figura 5.22, que mostra a comparação dos tempos de estabelecimento dos arquivos de *checkpoints* nos mecanismos não incremental e incremental. No caso em análise, ao ser utilizado o mecanismo incremental, o tempo médio para a atividade de *checkpointing* foi superior ao verificado com o mecanismo não incremental. Isto ocorreu devido ao enfoque utilizado para concepção do mecanismo incremental, que adiciona a este um esforço computacional para determinar se houve modificação, ou não, dos estados dos objetos da aplicação. Esperava-se que este esforço computacional fosse compensado pela redução da quantidade de dados a serem salvos nos arquivos de *checkpoint***,** devido à imutabilidade dos estados de alguns dos objetos da aplicação. Mas, na prática, isto não aconteceu: este acréscimo de computação traduz-se em atividades para serializar os estados dos objetos da aplicação e para realizar a comparação com os estados anteriores, mantidos em memória, também na forma serializada. Estas atividades, que são particulares ao mecanismo incremental, representam o acréscimo de tempo observado nos experimentos realizados.

Sem dúvida, o enfoque escolhido para o desenvolvimento do mecanismo de *checkpointing* incremental caracterizou o ponto mais negativo deste trabalho, pois não confirmou a idéia do mecanismo incremental, que deveria resultar em menor sobrecarga de processamento, ou seja, um tempo menor despendido para a atividade de *checkpointing*. Outra característica importante a respeito do mecanismo incremental é a redução da utilização do meio de armazenamento, onde somente os estados dos objetos modificados necessitam serem salvos. Esta última característica pode ser vista com eficiência neste trabalho. A figura 5.21 mostrou a comparação dos tamanhos dos arquivos de *checkpoints* nos dois mecanismos de *checkpointing*: não incremental e incremental. Esta comparação confirmou a maior utilização do meio de armazenamento quando é empregado o mecanismo não incremental do que quando é utilizado o mecanismo incremental. Mas isto depende da característica da aplicação e dos dados que são salvos no arquivo de *checkpoint*.

Espera-se que, com o estudo e resultados alcançados, este trabalho venha a somar-se às atividades do Grupo de Tolerância a Falhas da UFRGS, contribuindo para a programação de algoritmos de recuperação distribuída. A concepção deste trabalho foi muito importante para o aprendizado teórico-prático de recuperação e técnicas de *checkpointing*, principalmente no que se refere ao paradigma orientado a objetos. A estrutura desta biblioteca e alguns resultados parciais deste trabalho referentes à avaliação de desempenho foram apresentados durante o III Workshop de Testes e Tolerância a Falhas [SIL2002]. Esta apresentação foi bastante importante como instrumento de avaliação externa e determinante de alguns questionamentos e modificações.

Acredita-se que a descrição feita nesta dissertação possibilite ao programador adquirir os conhecimentos básicos necessários para a utilização da biblioteca proposta para a confecção de aplicações que apresentem maior resiliência e maior disponibilidade diante da possibilidade de ocorrência de falhas. Adicionalmente, extensões efetuadas sobre esta programação deverão permitir a programação mais fácil de algoritmos de recuperação para sistemas distribuídos.

É de grande interesse que o mecanismo de *checkpointing* incremental apresente um bom desempenho quando for utilizado em aplicações com características favoráveis. Mas para que isso ocorra, um novo enfoque deverá ser desenvolvido observando características de gastos de memória, uso de processador e escrita no meio de

armazenamento. Uma redução da utilização destes componentes poderia significar um aumento do desempenho deste mecanismo, mas isto tudo deve ser compensado com uma facilidade e uma razoável transparência para o programador. Algumas características podem ser consideradas para se buscar um mecanismo de *checkpointing* incremental mais eficiente:

- a utilização de um enfoque empregando reflexão computacional, onde é possível monitorar os objetos da aplicação e efetuar *checkpointing* somente dos atributos e ou objetos que foram modificados desde o último *checkpoint*;
- experimentar a técnica de *checkpointing* incremental proposta por Lawall e Muller, seguindo fielmente sua implementação: os autores utilizam uma variável de controle (*flag*) que, no caso do mecanismo incremental, indica se algum campo do objeto foi modificado desde o último *checkpoint*.

Um trabalho de grande interesse, também, seria o desenvolvimento de novas classes de *checkpointing* e recuperação para trabalhar com *threads* de programas em Java. Estas novas classes específicas poderiam estender as classes já desenvolvidas e com isso ampliar a abrangência da utilização da biblioteca para aplicações Java.

Anexo 1 Código-fonte da Biblioteca

Anexo 1.1 ArquivoCkpt.java

```
import java.io.*;
import java.text.*;

public class ArquivoCkpt implements Definicoes
{
  private File[] arquivos;
  private File arq;
  private int numArquivo;
  private int quantArquivos;
  private int tl;
  private String diretorio;

//---------------------------------------------------------------------------
  public ArquivoCkpt(String diretorio)
  {
    this.diretorio=diretorio;
    arquivos = new File[TF_F];
    arq = new File(diretorio);
    numArquivo=1;
    inicializaQuantArquivos();
    tl=-1;
  }//ArquivoCkpt constructor

//---------------------------------------------------------------------------
  public int getTL()
  {
    return tl;
  }//getTL

//---------------------------------------------------------------------------
  public void inicializaQuantArquivos()
  {
    quantArquivos=0;
  }//inicializaQuantArquivos

//---------------------------------------------------------------------------
  public int getQuantArquivos()
  {
    return quantArquivos;
  }//getQuantArquivos

//---------------------------------------------------------------------------
  public void setNumArquivo(int numArquivo)
  {
    this.numArquivo=numArquivo;
  }//setNumArquivo

//---------------------------------------------------------------------------
  public int getNumArquivo()
  {
    return numArquivo;
  }//getNumArquivo

//---------------------------------------------------------------------------
  public String constroiNomeArquivoCkpt()
  {
    String nome;
    String auxnum;
    Collator c = Collator.getInstance();
    auxnum=Integer.toString(numArquivo);
```

```
    if (auxnum.length()==1)
    {
      auxnum="000"+auxnum;
    }
    else
    if (auxnum.length()==2)
    {
      auxnum="00"+auxnum;
    }
    else
    if (auxnum.length()==3)
    {
      auxnum="0"+auxnum;
    }

    if (c.compare(diretorio,".")==0)
      nome="checkpoint_"+auxnum+".ckp";
    else
      nome=diretorio+"checkpoint_"+auxnum+".ckp";
    numArquivo++;
    quantArquivos++;
    return nome;
  }//constroiNomeArquivoCkpt

//---------------------------------------------------------------------------
  public void trocaPosicao(int i, int j)
  {
    File aux = new File(".");
    aux=arquivos[i];
    arquivos[i]=arquivos[j];
    arquivos[j]=aux;
  }//trocaPosicao

//---------------------------------------------------------------------------
  public void shellSort()
  {
    Collator c = Collator.getInstance();
    String aux_1 = new String();
    String aux_2 = new String();
    int dist,i,j,k;

    dist=4;
    while (dist>=1)
    {
      for (i=1 ; i<=dist ; i++)
      {
        j=i-1;
        while ((j+dist)<=tl)
        {
          aux_1=arquivos[j+dist].getName();
          aux_2=arquivos[j].getName();
          if (c.compare(aux_1,aux_2)<0)
          {
            k=j;
            trocaPosicao(j, j+dist);
            if (k-dist>=0)
            {
              aux_1=arquivos[k].getName();
              aux_2=arquivos[k-dist].getName();
              while ((k-dist>=0) && (c.compare(aux_1,aux_2)<0))
              {
                trocaPosicao(k, k-dist);
                k-=dist;
                if (k-dist>=0)
                {
                  aux_1=arquivos[k].getName();
                  aux_2=arquivos[k-dist].getName();
                }
```

```
            }
          }
        }
        j+=dist;
      }
    }
    dist=(int)dist/2;
  }
}//shellSort

//---------------------------------------------------------------------------
  public boolean buscaBinaria(String chave)
  {
    int inicio,fim,meio;
    Collator c = Collator.getInstance();
    inicio=0;
    fim=tl;
    meio=(int)(inicio+fim)/2;
    while ((inicio<fim) && (c.compare(chave,arquivos[meio].getName())!=0))
    {
      if (c.compare(chave,arquivos[meio].getName())>0)
        inicio=meio+1;
      else
        fim=meio;
      meio=(int)(inicio+fim)/2;
    }
    if (c.compare(chave,arquivos[meio].getName())==0)
      return true;
    else
      return false;
  }//buscaBinaria

//---------------------------------------------------------------------------
  public void leNomesArquivos()
  {
    int i;
    String ext = new String();
    File[] arquivos_aux = new File[TF_F];
    arquivos_aux=arq.listFiles();
    Collator c = Collator.getInstance();
    tl=-1;

    for (i=0 ; i<arquivos_aux.length ; i++)
    {
      ext=arquivos_aux[i].getName();
      if ((ext.length()==19) && (c.compare(ext.substring(15,19),".ckp")==0))
      {
        tl++;
        arquivos[tl]=arquivos_aux[i];
      }
    }
    shellSort();
    quantArquivos=tl+1;
  }//leNomesArquivos

//---------------------------------------------------------------------------
  public String obtemUltimoNomeArquivoCkpt()
  {
    Collator c = Collator.getInstance();
    String retorno;
    if (c.compare(diretorio,".")==0)
      retorno=arquivos[tl].getName();
    else
      retorno=diretorio+arquivos[tl].getName();
    return retorno;
  }//obtemUltimoNomeArquivoCkpt
```

```
//---------------------------------------------------------------------------
  public String obtemDeterminadoNomeArquivoCkpt(int pos)
  {
    Collator c = Collator.getInstance();
    String retorno="";
    if (pos<=tl)
    {
      if (c.compare(diretorio,".")==0)
        retorno=arquivos[pos].getName();
      else
        retorno=diretorio+arquivos[pos].getName();
    }
    return retorno;
  }//obtemDeterminadoNomeArquivoCkpt

//---------------------------------------------------------------------------
  public boolean verificaExistenciaArquivosCkpt()
  {
    if (tl>=0)
      return true;
    else
      return false;
  }//verificaExistenciaArquivosCkpt

//---------------------------------------------------------------------------
  public void excluirArquivosAnteriores()
  {
    int i;
    leNomesArquivos();
    for (i=0 ; i<tl ; i++)
    {
      arquivos[i].delete();
    }
    leNomesArquivos();
    quantArquivos=tl+1;
  }//excluirArquivosAnteriores

}//ArquivoCkpt class
```

Anexo 1.2 Checkpointing.java

```
import java.rmi.MarshalledObject;
import java.util.*;
import java.io.*;
import java.text.*;

public class Checkpointing implements Definicoes
{
  private transient String nome_arquivo;
  private transient FileOutputStream arquivo_out;
  private transient ObjectOutputStream obj_out;
  private transient FileInputStream arquivo_in;
  private transient ObjectInputStream obj_in;
  private transient Tempo tempo;
  private transient ParametrosCkpt param;
  private transient ArquivoCkpt nome_arquivo_ckpt;
  private transient EstatisticaCkpt estatistica;

  private transient boolean flagCkpt;
  private transient boolean flagRecuperacao;
  private transient boolean flagEntra;
  private transient boolean flagEstatistica;
  private transient boolean flagTempo;
```

```
  private transient boolean flagPrimeiraVez;
  private transient boolean flagEstabelece;

  private transient long tempos_extras;

  //Conteiner para guardar os objetos que estarao presentes no checkpoint
  private Map conteiner_ckpt = new HashMap();
  private Map conteiner_ckpt_estados = new HashMap();
  private transient int pos_conteiner_ckpt;

  //Conteiner para cabecalho do checkpoint
  private Map cabecalho = new HashMap();

//-------------------------------------------------------------------------
  public Checkpointing()
  {
    flagCkpt=false; //checkpoint nao foi estabelecido nenhuma vez
    flagRecuperacao=false;
    flagEntra=false;
    flagTempo=false;
    param = new ParametrosCkpt();
    nome_arquivo_ckpt = new ArquivoCkpt(param.getDiretorio());
    //Estatistica
    estatistica = new EstatisticaCkpt(param.getIntervalo(),
                                      param.getTipoIntervalo(),
                                      param.getNumMaxArquivos(),
                                      param.getIncremental(),
                                      param.getDiretorio());
    pos_conteiner_ckpt=INICIO_CONTEINER;
    flagPrimeiraVez=true;
    flagEstabelece=false;
  }//Checkpointing constructor

//-------------------------------------------------------------------------
  //Estatistica
  public void tempoInicialEstatistica(String nomeArquivoEstatistica)
  {
    flagEstatistica=true;
    estatistica.marcaTempoInicial(nomeArquivoEstatistica);
  }//tempoInicialEstatistica

//-------------------------------------------------------------------------
  //Estatistica
  public void tempoFinalEstatistica()
  {
    estatistica.marcaTempoFinal();
  }//tempoFinalEstatistica

//-------------------------------------------------------------------------
  public void inicializaSerializacao()
  {
    nome_arquivo=nome_arquivo_ckpt.constroiNomeArquivoCkpt();
    try
    {
      //saida
      arquivo_out = new FileOutputStream(nome_arquivo);
      obj_out = new ObjectOutputStream(arquivo_out);
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//inicializaSerializacao

//-------------------------------------------------------------------------
  public void inicializaDeSerializacao()
  {
    if (!flagRecuperacao)
    {
      nome_arquivo=nome_arquivo_ckpt.obtemUltimoNomeArquivoCkpt();
    }
    try
```

```
    {
      arquivo_in = new FileInputStream(nome_arquivo);
      obj_in = new ObjectInputStream(arquivo_in);
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//inicializaDeSerializacao

//-------------------------------------------------------------------------
  public void iniDeSerializacaoArquivo(int posArq)
  {
    nome_arquivo=nome_arquivo_ckpt.obtemDeterminadoNomeArquivoCkpt(posArq);
    try
    {
      arquivo_in = new FileInputStream(nome_arquivo);
      obj_in = new ObjectInputStream(arquivo_in);
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//iniDeSerializacaoArquivo

//-------------------------------------------------------------------------
  public void inicializaTempo()
  {
    tempo = new Tempo();
    //obtem valores do arquivo de parametros "paramckpt.dat" e
    //estabelece parametros do controle de tempo
    tempo.setParametros(param.getIntervalo(),param.getTipoIntervalo());
  }//inicializaTempo

//-------------------------------------------------------------------------
  public boolean verificaObjetosModificados(Object arg1,
                                            MarshalledObject arg2)
  {
    boolean retorno=false;
    try
    {
      MarshalledObject m_obj = new MarshalledObject(arg1);
      if (m_obj.equals(arg2))
        retorno=true;
    }catch(IOException e) { System.out.println("Error: "+e); }
    return retorno;
  }//verificaObjetosModificados

//-------------------------------------------------------------------------
  public void save(Object arg)
  {
    long tini,tfim;
    tini=System.currentTimeMillis();

    if (!flagCkpt)
    {
      flagTempo=true;
      flagCkpt=true;
      inicializaTempo();
      inicializaSerializacao();
    }

    if (pos_conteiner_ckpt==1)
    {
      if ((tempo.verificaTempoAcimaIntervalo()) ||
                          (param.getTipoIntervalo()=='V') || (flagTempo))
        flagEstabelece=true;
    }
    if (flagEstabelece)
    {
      if (flagPrimeiraVez)
      {
        tempos_extras=0;
        pos_conteiner_ckpt=INICIO_CONTEINER;
        flagPrimeiraVez=false;
      }
```

```
      //remove o objeto anterior do conteiner_ckpt da key pos_conteiner_ckpt
      conteiner_ckpt.remove(new Integer(pos_conteiner_ckpt));
      cabecalho.remove(new Integer(pos_conteiner_ckpt));

      if (param.getIncremental()=='S')
      {
        if (!verificaObjetosModificados(arg,(MarshalledObject)
         conteiner_ckpt_estados.get(new Integer(pos_conteiner_ckpt))))
        {
          conteiner_ckpt.put(new Integer(pos_conteiner_ckpt),arg);
          cabecalho.put(new Integer(pos_conteiner_ckpt),
                        new Integer(MODIFICADO));
          //insere no conteiner conteiner_ckpt_estados
          try
          {
            MarshalledObject m_obj = new MarshalledObject(arg);
           conteiner_ckpt_estados.put(new Integer(pos_conteiner_ckpt),m_obj);
          }catch(IOException e) { System.out.println("Error: "+e); }
        }
        else
        {
          cabecalho.put(new Integer(pos_conteiner_ckpt),
                        new Integer(NAO_MODIFICADO));
        }
      }
      else
      {
        conteiner_ckpt.put(new Integer(pos_conteiner_ckpt),arg);
        cabecalho.put(new Integer(pos_conteiner_ckpt),
                      new Integer(MODIFICADO));
      }
    }
    pos_conteiner_ckpt++;
    tfim=System.currentTimeMillis();
    tempos_extras+=(tfim-tini);
  }//save

//---------------------------------------------------------------------------
  public void checkpoint_here()
  {
    if (flagEstabelece)
    {
      //marca tempo inicial da gravacao do checkpoint
      if (flagEstatistica)
        estatistica.marcaTempoInicialCkpt();

      int i;
      boolean flagMerge=false;

      try
      {
        if (flagTempo)
          cabecalho.put(NUM_ARQUIVO,
                    new Integer(nome_arquivo_ckpt.getNumArquivo()-1));
        else
          cabecalho.put(NUM_ARQUIVO,
                      new Integer(nome_arquivo_ckpt.getNumArquivo()));
        conteiner_ckpt.put(HEAD,cabecalho);
        if (param.getIncremental()=='S')
        {
          if ((nome_arquivo_ckpt.getQuantArquivos()>=
                        param.getNumMaxArquivos()) &&
                        (param.getNumMaxArquivos()>1))
          {
            realizaMergeCheckpoint(CHECKPOINTING);
            //exclui os arquivos anteriores
            nome_arquivo_ckpt.excluirArquivosAnteriores();
```

```
            flagMerge=true;
          }
        }
        if (!flagMerge)
        {
          if ((nome_arquivo_ckpt.getQuantArquivos()>
                       param.getNumMaxArquivos()) &&
                       (param.getNumMaxArquivos()>1))
          {
            //exclui os arquivos anteriores
            nome_arquivo_ckpt.excluirArquivosAnteriores();
          }
          if (flagEntra)
          {
            inicializaSerializacao();
          }
          //salva estado dos objetos no checkpoint
          obj_out.writeObject(conteiner_ckpt);
          obj_out.flush();
          obj_out.close();
        }
      }catch(IOException e) { System.out.println("Error: "+e); }
      tempo.zeraTempo();
      flagEntra=true;
      //marca tempo final do checkpointing
      flagTempo=false;
      flagEstabelece=false;
      if (flagEstatistica)
        estatistica.marcaTempoFinalCkpt(nome_arquivo,tempos_extras);
      tempos_extras=0;
    }
    pos_conteiner_ckpt=INICIO_CONTEINER;
  }//checkpoint_here

//-------------------------------------------------------------------------
  public boolean verificaTodosObjsPresentes(Map arg)
  {
    int i;
    boolean retorno=true;
    Integer est = new Integer(0);

    for (i=1 ; i<arg.size() ; i++)
    {
      est=(Integer)arg.get(new Integer(i));
      if (est.intValue()==NAO_MODIFICADO)
      {
        retorno=false;
      }
    }
    return retorno;
  }//verificaTodosObjsPresentes

//-------------------------------------------------------------------------
  public Object recover()
  {
    Object retorno = new Object();
    if (pos_conteiner_ckpt<=conteiner_ckpt.size())
    {
      retorno=conteiner_ckpt.get(new Integer(pos_conteiner_ckpt));
      pos_conteiner_ckpt++;
    }
    return retorno;
  }//recover

//---------------------------------------------------------------------
  public void recover_oockpt()
  {
    int i;
```

```
    pos_conteiner_ckpt=INICIO_CONTEINER;

    nome_arquivo_ckpt.leNomesArquivos();
    if (nome_arquivo_ckpt.verificaExistenciaArquivosCkpt())
    {
      inicializaDeSerializacao();
      try
      {
        //recuperacao Checkpoint
        conteiner_ckpt = (HashMap)obj_in.readObject();
        cabecalho=(HashMap)conteiner_ckpt.get(HEAD);

        nome_arquivo_ckpt.setNumArquivo(
                                ((Integer)cabecalho.get(HEAD)).intValue());
        //recuperacao Checkpoint Incremental
        if (param.getIncremental()=='S')
        {
          if (!verificaTodosObjsPresentes(cabecalho))
            realizaMergeCheckpoint(RECUPERACAO);
          nome_arquivo_ckpt.leNomesArquivos();
          inicializaDeSerializacao();
          conteiner_ckpt = (HashMap)obj_in.readObject();
          cabecalho=(HashMap)conteiner_ckpt.get(HEAD);
        }
        else
          nome_arquivo_ckpt.setNumArquivo(
                               nome_arquivo_ckpt.getNumArquivo()+1);
      }catch(IOException e) { System.out.println("Error: "+e); }
       catch(ClassNotFoundException e) { System.out.println("Error: "+e); }

      flagEntra=false;
      flagCkpt=false;
    }
  }//recover_oockpt

//-------------------------------------------------------------------------
  public void copia_to_conteiner_ckpt_estados()
  {
    int i=1;
    int cont=1;

    conteiner_ckpt_estados.put(HEAD,cabecalho);
    while (cont<conteiner_ckpt.size())
    {
      if (conteiner_ckpt.containsKey(new Integer(i)))
      {
        if (((Integer)cabecalho.get(new Integer(i))).intValue()==MODIFICADO)
        {
          try
          {
            MarshalledObject m_obj = new
                        MarshalledObject(conteiner_ckpt.get(new Integer(i)));
            conteiner_ckpt_estados.put(new Integer(i),m_obj);
          }catch(IOException e) { System.out.println("Error: "+e); }
        }
        cont++;
      }
      i++;
    }
  }//copia_to_conteiner_ckpt_estados

//-------------------------------------------------------------------------
  public void copia_from_conteiner_ckpt_estados()
  {
    int i;
    try
    {
      for (i=1 ; i<conteiner_ckpt_estados.size() ; i++)
```

```
      {
        conteiner_ckpt.put(new Integer(i),
                  ((MarshalledObject)conteiner_ckpt_estados.get(
                  new Integer(i))).get());
      }
    }catch(IOException e) { System.out.println("Error: "+e); }
     catch(ClassNotFoundException e) { System.out.println("Error: "+e); }
  }//copia_from_conteiner_ckpt_estados

//-------------------------------------------------------------------------
  public void realizaMergeCheckpoint(char flag)
  {
    long tini,tfim;
    tini=System.currentTimeMillis();

    int aux_tl;
    int i;
    Map cabecalho_aux = new HashMap();
    Integer estado = new Integer(0);
    Integer estado_aux = new Integer(0);

    if (flag==RECUPERACAO)
    {
      //Copia conteiner_ckpt p/ conteiner_ckpt_estados
      copia_to_conteiner_ckpt_estados();

      cabecalho=(HashMap)conteiner_ckpt.get(HEAD);
      nome_arquivo_ckpt.leNomesArquivos();
      aux_tl=nome_arquivo_ckpt.getTL()-1;

      while ((!verificaTodosObjsPresentes(cabecalho)) && (aux_tl>=0))
      {
        iniDeSerializacaoArquivo(aux_tl);
        try
        {
          //recuperacao Checkpoint
          conteiner_ckpt=(HashMap)obj_in.readObject();
          cabecalho_aux=(HashMap)conteiner_ckpt.get(HEAD);
          //recuperacao Checkpoint Incremental
        }catch(IOException e) { System.out.println("Error: "+e); }
         catch(ClassNotFoundException e) { System.out.println("Error: "+e); }

        for (i=1 ; i<cabecalho_aux.size() ; i++)
        {
          estado=(Integer)cabecalho.get(new Integer(i));
          estado_aux=(Integer)cabecalho_aux.get(new Integer(i));
          if (estado_aux.intValue()==MODIFICADO)
          {
            if (estado.intValue()==NAO_MODIFICADO)
            {
              cabecalho.put(new Integer(i),new Integer(MODIFICADO));
              try
              {
                MarshalledObject m_obj = new
                        MarshalledObject(conteiner_ckpt.get(new Integer(i)));
                conteiner_ckpt_estados.put(new Integer(i),m_obj);
              }catch(IOException e) { System.out.println("Error: "+e); }
            }
          }
        }
        aux_tl--;
      }
    }
    else
    {
       for (i=1 ; i<cabecalho.size() ; i++)
       cabecalho.put(new Integer(i),new Integer(MODIFICADO));
    }
```

```
    //Copia conteiner_ckpt_estados p/ conteiner_ckpt
    copia_from_conteiner_ckpt_estados();

    //---- salva a unificacao (merge) em novo checkpoint ----

    //marca tempo inicial do checkpointing
    if (flagEstatistica)
    estatistica.marcaTempoInicialCkpt();
    try
    {
      if (flag==RECUPERACAO)
        nome_arquivo_ckpt.setNumArquivo(nome_arquivo_ckpt.getNumArquivo()+1);
      inicializaSerializacao();
      cabecalho.put(NUM_ARQUIVO,
                          new Integer(nome_arquivo_ckpt.getNumArquivo()-1));

      //salva estado dos objetos no checkpoint
      conteiner_ckpt.put(HEAD,cabecalho);
      obj_out.writeObject(conteiner_ckpt);
      obj_out.flush();
      obj_out.close();
    }catch(IOException e) { System.out.println("Error: "+e); }
    //marca tempo final do checkpointing
    tfim=System.currentTimeMillis();
    tempos_extras+=(tfim-tini);
    if ((flagEstatistica) && (flag==RECUPERACAO))
      estatistica.marcaTempoFinalCkpt(nome_arquivo,tempos_extras);
    //---

  }//realizaMergeCheckpoint

//-------------------------------------------------------------------------
  public void mostraExemplosArgumentos()
  {
    System.out.println(
             "+-------------------------------------------------------+");
    System.out.println(
             "| Argumentos validos na recuperacao                     |");
    System.out.println(
             "|  a) java <<Aplicacao>> recover                        |");
    System.out.println(
             "|  b) java <<Aplicacao>> recover -a checkpoint_????.ckp |");
    System.out.println(
             "+-------------------------------------------------------+");
  }//mostraExemplosArgumentos

//-------------------------------------------------------------------------
  public boolean verificaRecuperacao(String args[])
  {
    Collator c = Collator.getInstance();
    String s = new String();
    int tl_args;
    boolean flag_ok=false;

    nome_arquivo_ckpt.leNomesArquivos();
    tl_args=args.length;

    if (tl_args > 0)
    {
      if (nome_arquivo_ckpt.getTL()>-1)
      {
        s=args[0];
        s=s.toUpperCase();
        if (c.compare(s,"RECOVER")==0)
        {
          if (tl_args==1)
          {
```

```
            flag_ok=true;
          }
          else
          if (tl_args>=3)
          {
            s=args[1];
            s=s.toUpperCase();
            if (c.compare(s,"-A")==0)
            {
              s=args[2];
              s=s.toLowerCase();
              if (nome_arquivo_ckpt.buscaBinaria(s))
              {
                flagRecuperacao=true;
                if (c.compare(param.getDiretorio(),".")==0)
                  nome_arquivo=args[2];
                else
                  nome_arquivo=param.getDiretorio()+args[2];
                flag_ok=true;
              }
              else
               flag_ok=false;
            }
          }
          if (flag_ok)
          {
            System.out.println("iniciando recuperacao...");
            return true;
          }
          else
          {
            System.out.println("Argumentos invalidos na recuperacao...");
            System.out.println("Execucao do programa interrompida...");
            mostraExemplosArgumentos();
            System.exit(0);
            return false;
          }
        }
        else
        {
          System.out.println("Argumentos invalidos na recuperacao...");
          System.out.println("Execucao do programa interrompida...");
          mostraExemplosArgumentos();
          System.exit(0);
          return false;
        }
      }
      else
      {
        System.out.println("Nao existem arquivos de checkpoint...");
        System.out.println("Execucao do programa interrompida...");
        System.exit(0);
        return false;
      }
    }
    else
      return false;
  }//verificaRecuperacao

}//Checkpointing class
```

Anexo 1.3 EstatisticaCkpt.java

```
import java.io.*;

public class EstatisticaCkpt
{
  private FileOutputStream arquivoEst_out;
  private DataOutputStream stringEst_out;
  private File arq;
  private long tempoInicialAplicacao;
  private long tempoFinalAplicacao;
  private long tempoInicialCkpt;
  private long tempoFinalCkpt;
  private String nomeArquivo;
  private long est_intervalo;
  private char est_tipo_intervalo;
  private int est_num_max_arquivos;
  private char est_incremental;
  private String est_diretorio;
  private String nomeArqCkpt;

//---------------------------------------------------------------------------
  public EstatisticaCkpt(long est_intervalo, char est_tipo_intervalo,
                         int num_max_arquivos, char est_incremental,
                         String est_diretorio)
  {
    this.est_intervalo=est_intervalo;
    this.est_tipo_intervalo=est_tipo_intervalo;
    this.est_num_max_arquivos=est_num_max_arquivos;
    this.est_incremental=est_incremental;
    this.est_diretorio=est_diretorio;
  }//EstatisticaCkpt constructor

//---------------------------------------------------------------------------
  public void criaArquivoEstatistica()
  {
    try
    {
      arq = new File(nomeArquivo);
      arq.createNewFile();
      escreveParametros();
      stringEst_out.writeBytes("\r\n[tempo de cada checkpointing]\r\n");
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//criaArquivoEstatistica

//---------------------------------------------------------------------------
  public void escreveParametros()
  {
    try
    {
      //gravacao
      arquivoEst_out = new FileOutputStream(arq);
      stringEst_out  = new DataOutputStream(arquivoEst_out);
      //gravacao dos parametros
      stringEst_out.writeBytes(
                           "[parametros do arquivo \"paramckpt.dat\"]\r\n");
      stringEst_out.writeBytes("intervalo="+est_intervalo+"\r\n");
      stringEst_out.writeBytes("tipo_intervalo="+est_tipo_intervalo+"\r\n");
      stringEst_out.writeBytes(
                           "num_max_arquivos="+est_num_max_arquivos+"\r\n");
      stringEst_out.writeBytes("incremental="+est_incremental+"\r\n");
      stringEst_out.writeBytes("diretorio="+est_diretorio+"\r\n");
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//escreveParametros
```

```
//---------------------------------------------------------------------------
  //chamada da Aplicacao
  public void marcaTempoInicial(String nomeArquivo)
  {
    tempoInicialAplicacao=System.currentTimeMillis();
    this.nomeArquivo=nomeArquivo;
    criaArquivoEstatistica();
  }//marcaTempoInicial

//---------------------------------------------------------------------------
  //chamada da Aplicacao
  public void marcaTempoFinal()
  {
    tempoFinalAplicacao=System.currentTimeMillis();
    try
    {
      stringEst_out.writeBytes("\r\n[execucao da aplicacao]\r\n");
      stringEst_out.writeBytes("tempo execucao total = "+
                               (tempoFinalAplicacao-tempoInicialAplicacao));
      stringEst_out.writeBytes("\r\n\n-> tempos em milissegundos");
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//marcaTempoFinal

//---------------------------------------------------------------------------
  public void marcaTempoInicialCkpt()
  {
    tempoInicialCkpt=System.currentTimeMillis();
  }//marcaTempoInicialCkpt

//---------------------------------------------------------------------------
  public void marcaTempoFinalCkpt(String nomeArqCkpt, long temposExtras)
  {
    this.nomeArqCkpt=nomeArqCkpt;
    try
    {
      tempoFinalCkpt=System.currentTimeMillis();
      stringEst_out.writeBytes(nomeArqCkpt+" -> tempo = "+
                       (tempoFinalCkpt-tempoInicialCkpt+temposExtras)+"\r\n");
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//marcaTempoFinalCkpt

}//EstatisticaCkpt class
```

Anexo 1.4 InterParamCkpt.java

```
import java.io.*;
import java.text.*;

public class InterParamCkpt extends ParametrosCkpt
{

//---------------------------------------------------------------------------
  public InterParamCkpt()
  {
    super();
    leArquivo();
  }//InterParamCkpt

//---------------------------------------------------------------------------
  public String leElementoString()
  {
    String line="";
    DataInputStream meuDataInputStream = new DataInputStream(System.in);
```

```
    try
    {
      line=meuDataInputStream.readLine();
    }
    catch(Exception e){}
    return line;
  }//leElementoString

//---------------------------------------------------------------------------
  public int leElementoInteger()
  {
    String line="";
    DataInputStream meuDataInputStream = new DataInputStream(System.in);
    try
    {
      line=meuDataInputStream.readLine();
      int retorno=Integer.valueOf(line).intValue();
      return retorno;
    }
    catch(Exception e)
    {
      return -1;
    }
  }//leElementoInteger

//---------------------------------------------------------------------------
  public long leElementoLong()
  {
    String line="";
    DataInputStream meuDataInputStream = new DataInputStream(System.in);
    try
    {
      line=meuDataInputStream.readLine();
      long retorno=Long.valueOf(line).longValue();
      return retorno;
    }
    catch(Exception e)
    {
      return -1;
    }
  }//leElementoLong

//---------------------------------------------------------------------------
  public void exibePossiveisValoresParametros()
  {
    System.out.println("_____________________________________________");
    System.out.println(
           "valores possiveis dos parametros do arquvivo \"paramckpt.dat\"");
    System.out.println(
              "1) intervalo       -> 1... (numero correspondente ao tempo)");
    System.out.println("2) tipo_intervalo");
    System.out.println("   -> a) H (hora)        -> b) M (minuto)");
    System.out.println("   -> c) S (segundo)     -> d) L (milissegundo)");
    System.out.println("   -> e) V (sem intervalo)");
    System.out.println("3) num_max_arquivos");
    System.out.println("   -> a) 0,1 (sem efeito)");
    System.out.println(
       "   -> b) 2... (numero maximo de arquivos de checkpoint incremental)");
    System.out.println("4) incremental");
    System.out.println("   -> a) S (sim)         -> b) N (nao)");
    System.out.println("5) diretorio       -> . (diretorio desejado)");
  }//exibePossiveisValoresParametros

//---------------------------------------------------------------------------
  public void exibeValoresParametros()
  {
    System.out.println("_____________________________________________");
    System.out.println("  intervalo="+getIntervalo());
```

```
    System.out.println("  tipo_intervalo="+getTipoIntervalo());
    System.out.println("  num_max_arquivos="+getNumMaxArquivos());
    System.out.println("  incremental="+getIncremental());
    System.out.println("  diretorio="+getDiretorio());
    System.out.println("");
  }//exibeValoresParametros

//---------------------------------------------------------------------------
  public void exibeMenu()
  {
    System.out.println("_____________ Menu Principal _____________");
    System.out.println(" 1 - Exibe possiveis valores dos parametros");
    System.out.println(" 2 - Altera valores dos parametros");
    System.out.println(" 3 - Exibe valores dos parametros");
    System.out.println(" 0 - Sair do Programa");
  }//exibeMenu

//---------------------------------------------------------------------------
  public void alteraValores()
  {
    long aux_intervalo;
    char aux_tipo_intervalo;
    int aux_num_max_arquivos;
    char aux_incremental;
    String aux_diretorio;

    String aux;
    boolean flag=false;
    char resp_char=' ';

    System.out.println("_____________________________________________");
    System.out.println("Entre com os novos valores:");
    System.out.println("");

    //intervalo
    System.out.print("  intervalo=["+getIntervalo()+"]: ");
    aux_intervalo=leElementoLong();
    if (aux_intervalo==-1)
    {
      aux_intervalo=intervalo;
      System.out.println("  intervalo="+aux_intervalo);
    }

    //tipo_intervalo
    System.out.print("  tipo_intervalo=["+getTipoIntervalo()+"]: ");
    aux=leElementoString();
    String st_ti = new String(aux);
    if (st_ti.length()>0)
        aux_tipo_intervalo=st_ti.charAt(0);
      else
        aux_tipo_intervalo=' ';
      if ((aux_tipo_intervalo != 'H') && (aux_tipo_intervalo != 'M') &&
         (aux_tipo_intervalo != 'S') && (aux_tipo_intervalo != 'L') &&
         (aux_tipo_intervalo != 'V'))
      {
         aux_tipo_intervalo=tipo_intervalo;
         System.out.println("  tipo_intervalo="+aux_tipo_intervalo);
      }

      //num_max_arquivos
      System.out.print("  num_max_arquivos=["+getNumMaxArquivos()+"]: ");
      aux_num_max_arquivos=leElementoInteger();
      if (aux_num_max_arquivos==-1)
      {
        aux_num_max_arquivos=num_max_arquivos;
        System.out.println("  num_max_arquivos="+aux_num_max_arquivos);
      }
```

```
       //incremental
      System.out.print("  incremental=["+getIncremental()+"]: ");
      aux=leElementoString();
      String st_i = new String(aux);
      if (st_i.length()>0)
        aux_incremental=st_i.charAt(0);
      else
        aux_incremental=' ';
      if ((aux_incremental != 'S') && (aux_incremental != 'N'))
      {
         aux_incremental=incremental;
         System.out.println("  incremental="+aux_incremental);
      }

      //diretorio
      System.out.print("  diretorio=["+getDiretorio()+"]: ");
      aux=leElementoString();
      String st_d = new String(aux);
      if (st_d.length()>0)
        aux_diretorio=st_d;
      else
      {
        aux_diretorio=diretorio;
        System.out.println("  diretorio="+aux_diretorio);
      }

       System.out.println("");
      do
      {
        System.out.print("Confirma alteracoes? <S/N>: ");
        aux=leElementoString();
        String resp = new String(aux);
        resp=resp.toUpperCase();
        if ((resp.length()>0) && ((resp.charAt(0)=='S') ||
                                  (resp.charAt(0)=='N')))
        {
          flag=true;
          resp_char=resp.charAt(0);
        }
      }while (!flag);

      if (resp_char=='S')
      {
        atualizaArquivo(aux_intervalo, aux_tipo_intervalo,
                        aux_num_max_arquivos, aux_incremental,
                        aux_diretorio);
      }
      System.out.println("");
  }//alteraValores

//---------------------------------------------------------------------------
  public void controleMenu()
  {
    int op;
    try
    {
      exibeMenu();
      do
      {
        op=leElementoInteger();
        switch(op)
        {
          case 1:exibePossiveisValoresParametros();
                 exibeMenu();
                 break;
          case 2:alteraValores();
                 exibeMenu();
                 break;
```

```
          case 3:exibeValoresParametros();
                 exibeMenu();
                 break;
        }
      }while (op!=0);
    }catch(Exception erro) {}
  }//controleMenu

//-------------------------------------------------------------------------
  public static void main(String args[])
  {
    InterParamCkpt ipckpt = new InterParamCkpt();
    ipckpt.controleMenu();
  }//main

}//InterParamCkpt class
```

Anexo 1.5 ParametrosCkpt.java

```
import java.io.*;
import java.text.*;

public class ParametrosCkpt
{
  protected FileOutputStream arquivo_out;
  protected DataOutputStream string_out;
  protected FileInputStream arquivo_in;
  protected DataInputStream string_in;
  protected File arq;
  protected String nomearquivo;
  //parametros
  protected long intervalo;
  protected char tipo_intervalo;
  protected int num_max_arquivos;
  protected char incremental;
  protected String diretorio;

//-------------------------------------------------------------------------
  public ParametrosCkpt()
  {
    nomearquivo="paramckpt.dat";
    try
    {
      arq = new File(nomearquivo);
      if (!arq.exists())
      {
        arq.createNewFile();
        gravaArquivoPadrao();
      }
      else
      {
        leArquivo();
      }
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//ParametrosCkpt constructor

//-------------------------------------------------------------------------
  public long getIntervalo()
  {
    return intervalo;
  }//getIntervalo
```

```
//---------------------------------------------------------------------------
  public char getTipoIntervalo()
  {
    return tipo_intervalo;
  }//getTipoIntervalo

//---------------------------------------------------------------------------
  public int getNumMaxArquivos()
  {
    return num_max_arquivos;
  }//getNumMaxArquivos

//---------------------------------------------------------------------------
  public char getIncremental()
  {
    return incremental;
  }//getIncremental

//---------------------------------------------------------------------------
  public String getDiretorio()
  {
    return diretorio;
  }//getDiretorio

//---------------------------------------------------------------------------
  public void setIntervalo(long intervalo)
  {
    this.intervalo=intervalo;
  }//setIntervalo

//---------------------------------------------------------------------------
  public void setTipoIntevalo(char tipo_intervalo)
  {
    this.tipo_intervalo=tipo_intervalo;
  }//setTipoIntevalo

//---------------------------------------------------------------------------
  public void setNumMaxArquivos(int num_max_arquivos)
  {
    this.num_max_arquivos=num_max_arquivos;
  }//setNumMaxArquivos

//---------------------------------------------------------------------------
  public void setIncremental(char incremental)
  {
    this.incremental=incremental;
  }//setIncremental

//---------------------------------------------------------------------------
  public void setDiretorio(String diretorio)
  {
    this.diretorio=diretorio;
  }//setDiretorio

//---------------------------------------------------------------------------
  public String leValorParametro(String param)
  {
    String st = new String(param);
    int i;
    int ini;
    int cont;
    char[] ch = new char[30];

    i=0;
    while ((i<st.length()) && (st.charAt(i)!= '='))
    {
      i++;
    }
```

```
    i++;  //passa para frente do '='
    while ((i<st.length()) && (st.charAt(i)== ' '))
    //elimina os espacos em branco
    {
      i++;
    }
    ini=i;
    cont=0;
    while ((i<st.length()) && (st.charAt(i)!= ' '))
    {
      i++;
      cont++;
    }
    String retorno;
    retorno=st.substring(ini,ini+cont);
    return retorno;
  }//leValorParametro

//---------------------------------------------------------------------------
  public void leArquivo()
  {
    String linha;
    int cont=0;
    try
    {
      //leitura
      arquivo_in  = new FileInputStream(arq);
      string_in   = new DataInputStream(arquivo_in);

      while ((linha=string_in.readLine()) !=null)
      {
        switch (cont)
        {
          case 0:try
                 {
                   String s_intervalo;
                   s_intervalo=leValorParametro(linha);
                   intervalo=Long.parseLong(s_intervalo);
                 }catch(NumberFormatException e)
                 {
                 System.out.println("Valores errados no arquivo de
                                                          parametros...");
                 }
                 break;

          case 1:String st_ti = new String(leValorParametro(linha));
                 tipo_intervalo=st_ti.charAt(0);
                 if ((tipo_intervalo != 'H') && (tipo_intervalo != 'M') &&
                     (tipo_intervalo != 'S') && (tipo_intervalo != 'L') &&
                     (tipo_intervalo != 'V'))
                 {
                   System.out.println("Valores errados no arquivo de
                                                          parametros...");
                 }
                 break;

          case 2:try
                 {
                   String s_num_max_arquivos;
                   s_num_max_arquivos=leValorParametro(linha);
                   num_max_arquivos=Integer.parseInt(s_num_max_arquivos);
                 }catch(NumberFormatException e)
                 {
                   System.out.println("Valores errados no arquivo de
                                                          parametros...");
                 }
                 break;
```

```
          case 3:String st_i = new String(leValorParametro(linha));
                 incremental=st_i.charAt(0);
                 if ((incremental != 'S') && (incremental != 'N'))
                 {
                   System.out.println("Valores errados no arquivo de
                                                          parametros...");
                 }
                 break;

          case 4:String st_d = new String(leValorParametro(linha));
                 diretorio=st_d;
                 if (verificaSOWindows())
                 {
                   if (existeBarraN(diretorio))
                   {
                     System.out.println("Diretorio invalido para o SO
                                                                 Windows");
                     System.out.println("Altere o conteudo do parametro
                                  \"diretorio\" do arquivo paramckpt.dat");
                     System.out.println("Execucao do programa
                                                         interrompida...");
                     System.exit(0);
                   }
                 }
                 else
                 {
                   if (existeBarraI(diretorio))
                   {
                     System.out.println("Diretorio invalido para o SO
                                                                   Linux");
                     System.out.println("Altere o conteudo do parametro
                                  \"diretorio\" do arquivo paramckpt.dat");
                     System.out.println("Execucao do programa
                                                         interrompida...");
                     System.exit(0);
                   }
                 }
                 break;
        }
        cont++;
      }
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//leArquivo

//---------------------------------------------------------------------------
  public void gravaArquivoPadrao()
  {
    try
    {
      //gravacao
      arquivo_out = new FileOutputStream(arq);
      string_out  = new DataOutputStream(arquivo_out);
      //parametros
      setIntervalo(1000);
      setTipoIntevalo('L');
      setNumMaxArquivos(0);
      setIncremental('N');
      setDiretorio(".");
      //grava parametros
      string_out.writeBytes("intervalo=1000\r\n");
      string_out.writeBytes("tipo_intervalo=L\r\n");
      string_out.writeBytes("num_max_arquivos=0\r\n");
      string_out.writeBytes("incremental=N\r\n");
      string_out.writeBytes("diretorio=.\r\n");
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//gravaArquivoPadrao

//---------------------------------------------------------------------------
```

```
  public void atualizaArquivo()
  {
    try
    {
      //gravacao
      arquivo_out = new FileOutputStream(arq);
      string_out  = new DataOutputStream(arquivo_out);
      //parametros
      string_out.writeBytes("intervalo="+getIntervalo()+"\r\n");
      string_out.writeBytes("tipo_intervalo="+getTipoIntervalo()+"\r\n");
      string_out.writeBytes("num_max_arquivos="+getNumMaxArquivos()+"\r\n");
      string_out.writeBytes("incremental="+getIncremental()+"\r\n");
      string_out.writeBytes("diretorio="+getDiretorio());
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//atualizaArquivo

//-------------------------------------------------------------------------
  public boolean existeBarraN(String p_diretorio)
  {
    int tam,i;
    String aux = new String(p_diretorio);
    tam=aux.length();
    boolean retorno=false;
    i=0;
    while ((i<tam) && (aux.charAt(i)!='/'))
    {
      i++;
    }
    if (i<tam)
      retorno=true;
    return retorno;
  }//existeBarraN

//-------------------------------------------------------------------------
  public boolean existeBarraI(String p_diretorio)
  {
    int tam,i;
    String aux = new String(p_diretorio);
    tam=aux.length();
    boolean retorno=false;
    i=0;
    while ((i<tam) && (aux.charAt(i)!='\\'))
    {
      i++;
    }
    if (i<tam)
      retorno=true;
    return retorno;
  }//existeBarraI

//-------------------------------------------------------------------------
  public String verificaBarrasSOWindows(String p_diretorio)
  {
    int tam,i;
    String aux = new String(p_diretorio);
    String auxfinal = new String();

    tam=aux.length();
    i=0;
    while (i<tam)
    {
      while ((i<tam) && (aux.charAt(i)!='\\'))
      {
        auxfinal=auxfinal+aux.charAt(i);
        i++;
      }
      auxfinal=auxfinal+'\\'+'\\';
      while ((i<tam) && (aux.charAt(i)=='\\'))
```

```
      {
        i++;
      }
    }
    return auxfinal;
  }//verificaBarrasSOWindows

//---------------------------------------------------------------------------
  public String verificaBarrasSOLinux(String p_diretorio)
  {
    int tam,i;
    String aux = new String(p_diretorio);

    tam=aux.length();
    if (aux.charAt(tam-1)!='/')
    aux=aux+'/';
    return aux;
  }//verificaBarrasSOLinux

//---------------------------------------------------------------------------
  public void atualizaArquivo(long p_intervalo, char p_tipo_intervalo,
                              int p_num_max_arquivos, char p_incremental,
                              String p_diretorio)
  {
    Collator c = Collator.getInstance();

    setIntervalo(p_intervalo);
    setTipoIntevalo(p_tipo_intervalo);
    setNumMaxArquivos(p_num_max_arquivos);
    setIncremental(p_incremental);
    if (c.compare(p_diretorio,".")==0)
      setDiretorio(p_diretorio);
    else
    {
      if (verificaSOWindows()) //Sistema Operacional Windows
        setDiretorio(verificaBarrasSOWindows(p_diretorio));
      else //Sistema Operacional Linux
        setDiretorio(verificaBarrasSOLinux(p_diretorio));
    }
    atualizaArquivo();
  }//atualizaArquivo

//---------------------------------------------------------------------------
  public boolean verificaSOWindows()
  {
    Collator c = Collator.getInstance();
    String so = new String();
    so=System.getProperty("os.name");
    if (c.compare(so.substring(0,3),"Win")==0)
      return true;
    else return false;
  }//verificaSOWindows

}//ParametrosCkpt class
```

Anexo 1.6 Tempo.java

```
import java.util.Date;

public class Tempo
{
  private long intervalo;
  private int hora;
  private int minuto;
  private int segundo;
  private long milissegundo;
  private long t_segini;
  private char tipo_intervalo;

//-------------------------------------------------------------------------
  public Tempo()
  {
    intervalo=0;
    t_segini=0;
    tipo_intervalo=' ';
  }//Tempo constructor

//-------------------------------------------------------------------------
  public void setParametros(long intervalo, char tipo_intervalo)
  {
    this.intervalo=intervalo;
    this.tipo_intervalo=tipo_intervalo;
    zeraTempo();
  }//setParametros

//-------------------------------------------------------------------------
  public void zeraTempo()
  {
    Date nowini = new Date();
    hora=nowini.getHours();
    minuto=nowini.getMinutes();
    segundo=nowini.getSeconds();
    milissegundo=nowini.getTime();
    t_segini=(hora*3600)+(minuto*60)+(segundo);
  }//zeraTempo

//-------------------------------------------------------------------------
  public boolean verificaTempoAcimaIntervalo()
  {
    long t_s;
    long t_seg;
    boolean retorno=false;
    int h;
    int m;
    int s;
    int c_hora;
    int c_minuto;
    int c_segundo;
    long c_milissegundo;

    Date now = new Date();
    h=now.getHours();
    m=now.getMinutes();
    s=now.getSeconds();
    c_milissegundo=(now.getTime()-milissegundo);

    t_seg=(h*3600)+(m*60)+(s);
    t_s=t_seg-t_segini;
    switch (tipo_intervalo)
    {
      case 'H':if ((t_s/3600) > intervalo)
                 retorno=true;
```

```
                break;
       case 'M':if ((t_s/60) > intervalo)
                  retorno=true;
                break;
       case 'S':if (t_s > intervalo)
                  retorno=true;
                break;
       case 'L':if (c_milissegundo > intervalo)
                  retorno=true;
                break;
    }
    return retorno;
  }//verificaTempoAcimaIntervalo

}//Tempo class
```

Anexo 1.7 Definicoes.java

```
public interface Definicoes
{
  public final int INICIO_CONTEINER = 1;
  public final char RECUPERACAO = 'R';
  public final char CHECKPOINTING = 'C';
  public final Integer HEAD = new Integer(0);
  public final Integer NUM_ARQUIVO = new Integer(0);
  public final int MODIFICADO = 1;
  public final int NAO_MODIFICADO = -1;
  public final int TF_F = 9999;
}//Definicoes interface
```

Anexo 2 Código-fonte das Aplicações

Anexo 2.1 OperaMatriz.java

```
import java.io.*;
import java.text.*;

public class OperaMatriz
{

  Checkpointing c = new Checkpointing();

  public OperaMatriz()
  {
    c.tempoInicialEstatistica("estatOperaMatriz.dat");
  }//OperaMatriz constructor

//--------------------------------------------------------------------------
  public Integer[][] carregaMatrizA()
  {
    Integer m[][] = new Integer[615][615];
    for (int i=0  ; i<615 ; i++)
      for (int j=0 ; j<615 ; j++)
        m[i][j]=new Integer(i+1);
    return m;
  }//carregaMatrizA

//--------------------------------------------------------------------------
  public Integer[][] carregaMatrizB()
  {
    Integer m[][] = new Integer[615][615];
    for (int i=0  ; i<615 ; i++)
      for (int j=0 ; j<615 ; j++)
        m[i][j]=new Integer(j+1);
    return m;
  }//carregaMatrizB

//--------------------------------------------------------------------------
  public void salvaMatriz(Integer m[][])
  {
    int i,j;
    FileOutputStream arquivo_out;
    DataOutputStream string_out;
    File arq;
    try
    {
      arq = new File("mat_prod.txt");
      arq.createNewFile();
      arquivo_out = new FileOutputStream(arq);
      string_out  = new DataOutputStream(arquivo_out);

      for (i=0 ; i<615 ; i++)
      {
        for (j=0 ; j<615 ; j++)
          string_out.writeBytes(m[i][j]+"  ");
        string_out.writeBytes("\n");
      }
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//salvaMatriz
```

```
//--------------------------------------------------------------------------
  public Integer[][] multMatriz(Integer m1[][], Integer m2[][], String args[])
  {
    Integer m3[][] = new Integer[615][615];
    int per=0,ini=0;

    if (c.verificaRecuperacao(args))
    {
      //restaurando os estados
      c.recover_oockpt();
      m3=(Integer[][])c.recover();
      ini=(((Integer)c.recover()).intValue());
    }

    for (int i=ini  ; i<615 ; i++)
    {
      for (int j=0 ; j<615 ; j++)
      {
        int s=0;
        for (int k=0 ; k<615 ; k++)
          s+=(m1[i][k].intValue()*m2[k][j].intValue());
        m3[i][j] = new Integer(s);
      }
      //salvando os estados
      c.save(m3);
      c.save(new Integer(i+1));
      c.checkpoint_here();
      if (((int)(i*100/615))!=per)
      {
        per=(int)(i*100/615);
        System.out.println(per+"%");
      }
    }
    System.out.println(100+"%");
    return m3;
  }//multMatriz

//--------------------------------------------------------------------------
  public void fim()
  {
    c.tempoFinalEstatistica();
  }//fim

//--------------------------------------------------------------------------
  public static void main(String args[])
  {
    OperaMatriz op = new OperaMatriz();
    Integer m1[][] = new Integer[615][615];
    Integer m2[][] = new Integer[615][615];
    Integer m3[][] = new Integer[615][615];
    m1=op.carregaMatrizA();
    m2=op.carregaMatrizB();
    m3=op.multMatriz(m1,m2,args);

    //Grava matriz resultante no arquivo de saida
    op.salvaMatriz(m3);
    op.fim();
  }//main

}//OperaMatriz class
```

Anexo 2.2 Transformada.java

```
import java.io.*;

public class Transformada
{
  Checkpointing chk = new Checkpointing();

  FileOutputStream arquivo_out;
  DataOutputStream string_out;
  File arq;

  //Calculo da Transformada Discreta do Cosseno
  public final int TAM_BLOCO = 8;
  public final int LARG = 512;
  public final int ALT = 512;

  Double MatImgOriginal[][] = new Double[LARG][ALT];
  Double MatTDC[][] = new Double[LARG][ALT];
  Double MatBlocoInt[][] = new Double[TAM_BLOCO][TAM_BLOCO];

  double alfa1;
  double alfa2;

  public Transformada()
  {
    chk.tempoInicialEstatistica("Transformada.dat");
    try
    {
      arq = new File("Transformada_result.txt");
      arq.createNewFile();
      arquivo_out = new FileOutputStream(arq);
      string_out  = new DataOutputStream(arquivo_out);
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//Transformada constructor

//-------------------------------------------------------------------------
  public double calcula_TDC(int i,int j)
  {
    int x,y,xy;
    double pi=3.141592654,r_TDC;
    double Mcos_i[] = new double[TAM_BLOCO];
    double Mcos_j[] = new double[TAM_BLOCO];

    r_TDC=0;

    // Calculo dos cossenos
    for (xy=0 ; xy<TAM_BLOCO ; xy++)
    {
      Mcos_i[xy]=Math.cos(((2*xy+1)*i*pi)/(2*TAM_BLOCO)); //x
      Mcos_j[xy]=Math.cos(((2*xy+1)*j*pi)/(2*TAM_BLOCO)); //y
    }
    for (x=0 ; x<TAM_BLOCO ; x++)
    {
      for (y=0 ; y<TAM_BLOCO ; y++)
      {
        //  Calculo segundo formula
        //  r_TDC=r_TDC + (MatBlocoInt[x][y] *
        //        cos(((2*x+1)*i*pi)/(2*TAM_BLOCO)) *
        //        cos(((2*y+1)*j*pi)/(2*TAM_BLOCO)));

        // Calculo otimizado (calcula-se primeiro os cossenos
        //                    (eles se repetirao))
        r_TDC=r_TDC + (MatBlocoInt[x][y].doubleValue() *
                                                  Mcos_i[x] * Mcos_j[y]);
      }
```

```
    }

    if (i==0) //alfa(i)
      r_TDC=r_TDC*alfa1;
    else
      r_TDC=r_TDC*alfa2;

    if (j==0) //alfa(j)
      r_TDC=r_TDC*alfa1;
    else
      r_TDC=r_TDC*alfa2;

    return(r_TDC);
  }//calcula_TDC

//---------------------------------------------------------------------------
  public void calcula_Transformada(String args[])
  {
    int i,j,ix,jx,iaux,jaux,nc,c,nl,l;
    int ini_nl=1,ini_nc=1;
    double valTDC;
    c=(int)LARG/TAM_BLOCO;
    l=(int)ALT/TAM_BLOCO;

    alfa1=1/Math.sqrt(TAM_BLOCO);
    alfa2=(Math.sqrt(2)/Math.sqrt(TAM_BLOCO));
    iaux=0;
    jaux=0;

    if (chk.verificaRecuperacao(args))
    {
      chk.recover_oockpt();
      iaux=(((Integer)chk.recover()).intValue());
      jaux=(((Integer)chk.recover()).intValue());
      ini_nl=(((Integer)chk.recover()).intValue());
      ini_nc=(((Integer)chk.recover()).intValue());
      MatTDC=(Double[][])chk.recover();
      MatImgOriginal=(Double[][])chk.recover();
    }

    for (nl=ini_nl ; nl<=l ; nl++)
    {
      for (nc=ini_nc ; nc<=c ; nc++)
      {
        ix=0;
        jx=0;
        //Pega cada Bloco da Imagem e joga em "MatBloco" para
        //calcular em seguida a transformada
        for (i=iaux ; i<TAM_BLOCO+iaux ; i++)
        {
          for (j=jaux ; j<TAM_BLOCO+jaux ; j++)
          {
           MatBlocoInt[ix][jx]=new Double(MatImgOriginal[i][j].doubleValue());
            jx++;
          }
          ix++;
          jx=0;
        }
        //Calcula a Transformada de um Bloco da Imagem
        ix=0;
        jx=0;
        for (i=iaux ; i<TAM_BLOCO+iaux ; i++)
        {
          for (j=jaux ; j<TAM_BLOCO+jaux ; j++)
          {
            valTDC=calcula_TDC(ix,jx);
            MatTDC[i][j] = new Double(valTDC);
            jx++;
```

```
          }
          ix++;
          jx=0;
        }
        //salvando estados
        chk.save(new Integer(iaux));
        chk.save(new Integer(jaux));
        chk.save(new Integer(nl));
        chk.save(new Integer(nc));
        chk.save(MatTDC);
        chk.save(MatImgOriginal);
        chk.checkpoint_here();
        System.out.print(".");
        iaux=iaux+TAM_BLOCO;
      }
      ini_nc=1;
      jaux=jaux+TAM_BLOCO;
      iaux=0;
    }
  }//calcula_Transformada

//-----------------------------------------------------------------------------
  double calcula_ITDC(int x, int y)
  {
    int i,j,ij;
    double pi=3.141592654,r_ITDC,calc;
    double Mcos_x[] = new double[TAM_BLOCO];
    double Mcos_y[] = new double[TAM_BLOCO];

    r_ITDC=0;

    // Calculo dos cossenos
    for (ij=0 ; ij<TAM_BLOCO ; ij++)
    {
      Mcos_x[ij]=Math.cos(((2*x+1)*ij*pi)/(2*TAM_BLOCO)); //i
      Mcos_y[ij]=Math.cos(((2*y+1)*ij*pi)/(2*TAM_BLOCO)); //j
    }
    for (i=0 ; i<TAM_BLOCO ; i++)
    {
      for (j=0 ; j<TAM_BLOCO ; j++)
      {
        //  Calculo segundo formula
        // if (i==0)
        //   calc=alfa1 * MatBlocoInt[i][j] *
        //        (cos(((2*x+1)*i*pi)/(2*TAM_BLOCO)) *
        //        cos(((2*y+1)*j*pi)/(2*TAM_BLOCO)));
        // else
        //   calc=alfa2 * MatBlocoInt[i][j] *
        //        (cos(((2*x+1)*i*pi)/(2*TAM_BLOCO)) *
        //        cos(((2*y+1)*j*pi)/(2*TAM_BLOCO)));
        // Calculo otimizado (calcula-se primeiro os cossenos
        //                    (eles se repetirao))
        if (i==0)
          calc=alfa1 * MatBlocoInt[i][j].doubleValue() *
                                                  (Mcos_x[i] * Mcos_y[j]);
        else
          calc=alfa2 * MatBlocoInt[i][j].doubleValue() *
                                                  (Mcos_x[i] * Mcos_y[j]);
        if (j==0)
          calc=calc*alfa1;
        else
          calc=calc*alfa2;
        r_ITDC=r_ITDC+calc;
      }
    }
    return(r_ITDC);
  }//calcula_ITDC
```

```
//-----------------------------------------------------------------------------
  public void calcula_TransformadaInversa()
  {
    int i,j,ix,jx,iaux,jaux,nc,c,nl,l;
    c=(int)LARG/TAM_BLOCO;
    l=(int)ALT/TAM_BLOCO;

    alfa1=1/Math.sqrt(TAM_BLOCO);
    alfa2=(Math.sqrt(2)/Math.sqrt(TAM_BLOCO));

    iaux=0;
    jaux=0;

    for (nl=1 ; nl<=l ; nl++)
    {
      for (nc=1 ; nc<=c ; nc++)
      {
        ix=0;
        jx=0;
        //Pega cada Bloco da Imagem na MatTDC e joga em "MatBloco" para
        //calcular a transformada em seguida
        for (i=iaux ; i<TAM_BLOCO+iaux ; i++)
        {
          for (j=jaux ; j<TAM_BLOCO+jaux ; j++)
          {
            MatBlocoInt[ix][jx]=MatTDC[i][j];
            jx++;
          }
          ix++;
          jx=0;
        }
        //Calcula a Transformada Inversa num Bloco da Imagem
        ix=0;
        jx=0;
        for (i=iaux ; i<TAM_BLOCO+iaux ; i++)
        {
          for (j=jaux ; j<TAM_BLOCO+jaux ; j++)
          {
           MatImgOriginal[i][j]=new Double(Math.round(calcula_ITDC(ix,jx)));
            jx++;
          }
          ix++;
          jx=0;
        }
        iaux=iaux+TAM_BLOCO;
      }
      jaux=jaux+TAM_BLOCO;
      iaux=0;
    }
  }//calcula_TransformadaInversa

//-----------------------------------------------------------------------------
  public void carregaMatrizImg()
  {
    int cor=0;
    for (int i=0 ; i<LARG ; i++)
    {
      for (int j=0 ; j<ALT ; j++)
        MatImgOriginal[i][j]=new Double(cor);
      cor++;
      if (cor==255)
        cor=0;
    }
  }//carregaMatrizImg
```

```
//---------------------------------------------------------------------------
  public void escreveMatrizImg()
  {
    System.out.println("\nsalvando a matriz apos a Transformada Inversa");
    try
    {
      string_out.writeBytes("Resultado da Matriz apos a Transformada
                                                                    Inversa\n");
      for (int i=0 ; i<LARG ; i++)
      {
        for (int j=0 ; j<ALT ; j++)
          string_out.writeBytes(MatImgOriginal[i][j].doubleValue()+"  ");
        string_out.writeBytes("\n");
      }
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//escreveMatrizImg

//---------------------------------------------------------------------------
  public void escreveMatrizTDC()
  {
    System.out.println("\nsalvando a matriz apos a Transformada");
    try
    {
      string_out.writeBytes("Resultado da Matriz apos a Transformada\n");
      for (int i=0 ; i<LARG ; i++)
      {
        for (int j=0 ; j<ALT ; j++)
          string_out.writeBytes(MatTDC[i][j].doubleValue()+"  ");
        string_out.writeBytes("\n");
      }
    }catch(IOException e) { System.out.println("Error: "+e); }

  }//escreveMatrizTDC

//---------------------------------------------------------------------------
  public void fim()
  {
    chk.tempoFinalEstatistica();
  }//fim

//---------------------------------------------------------------------------
  public static void main(String args[])
  {
    Transformada t = new Transformada();
    t.carregaMatrizImg();
    System.out.println("Calculando a Transformada");
    t.calcula_Transformada(args);
    t.escreveMatrizTDC();
    System.out.println("\r\nCalculando a Transformada Inversa");
    t.calcula_TransformadaInversa();
    t.escreveMatrizImg();
    t.fim();
  }//main

}//Transformada class
```

Anexo 2.3 ShellSort.java

```
public class ShellSort
{

  Checkpointing c = new Checkpointing();

  public ShellSort()
  {
    c.tempoInicialEstatistica("ShellSort.dat");
  }//ShellSort constructor

//-----------------------------------------------------------------------------
  public void trocaPosicao(Integer vet[], int i, int j)
  {
    Integer aux = new Integer(0);
    aux=vet[i];
    vet[i]=vet[j];
    vet[j]=aux;
  }//trocaPosicao

//-----------------------------------------------------------------------------
  public Integer[] shellSort(Integer vet[], String args[])
  {
    int dist,i,j,k,tl,per=0,ini=1;
    tl=vet.length;
    dist=8;

    if (c.verificaRecuperacao(args))
    {
      //restaurando os estados
      c.recover_oockpt();
      ini=(((Integer)c.recover()).intValue());
      dist=(((Integer)c.recover()).intValue());
      vet=(Integer[])c.recover();
    }

    while (dist>=1)
    {
      for (i=ini ; i<=dist ; i++)
      {
        j=i-1;
        while ((j+dist)<tl)
        {
          if (vet[j+dist].intValue()<vet[j].intValue())
          {
            k=j;
            trocaPosicao(vet,j,j+dist);
            if (k-dist>=0)
            {
              while ((k-dist>=0) &&
                                  (vet[k].intValue()<vet[k-dist].intValue()))
              {
                if (k-dist>=0)
                  trocaPosicao(vet,k,k-dist);
                k-=dist;
              }
            }
          }
          j+=dist;
          //salvando os estados
          c.save(new Integer(i));
          c.save(new Integer(dist));
          c.save(vet);
          c.checkpoint_here();
          System.out.print(".");
```

```
        }
      }
      dist=(int)dist/2;
      ini=1; //inicializa o ini para a proxima repeticao
    }
    return vet;
  }//shellSort

//---------------------------------------------------------------------------
  public void exibe(Integer vet[])
  {
    for (int i=0 ; i<vet.length ; i++)
    {
      System.out.println(vet[i].intValue());
    }
  }//exibe

//---------------------------------------------------------------------------
  public void fim()
  {
    c.tempoFinalEstatistica();
  }//fim

//---------------------------------------------------------------------------
  public static void main(String args[])
  {
    ShellSort s = new ShellSort();
    Integer v[] = new Integer[100000];
    for (int i=0 ; i<v.length ; i++)
      v[i] =  new Integer(100000-i);
    v=s.shellSort(v,args);
    System.out.println("Vetor Ordenado");
    s.exibe(v);
    s.fim();
  }//main

}//ShellSort class
```

Anexo 2.4 HeapSort.java

```
public class HeapSort
{

  Checkpointing c = new Checkpointing();

  public HeapSort()
  {
    c.tempoInicialEstatistica("HeapSort.dat");
  }//HeapSort constructor

//---------------------------------------------------------------------------
  public void trocaPosicao(Integer vet[], int i, int j)
  {
    Integer aux = new Integer(0);
    aux=vet[i];
    vet[i]=vet[j];
    vet[j]=aux;
  }//trocaPosicao

//---------------------------------------------------------------------------
  public Integer[] heapSort(Integer vet[], String args[])
  {
    int meio,f1,f2,fmaior,i,pai,tl,per=0,ini;
    tl=vet.length;
```

```
    meio=(int)(tl-1)/2;
    meio--;
    ini=tl-1;

    if (c.verificaRecuperacao(args))
    {
      //restaurando os estados
      c.recover_oockpt();
      ini=(((Integer)c.recover()).intValue());
      vet=(Integer[])c.recover();
    }

    for (i=ini ; i>0 ; i--)
    {
      for (pai=meio ; pai>=0 ; pai--)
      {
        f1=pai+pai+1;
        f2=pai+pai+2;
        fmaior=f1;
        if ((f2 <= i) && (vet[f2].intValue() > vet[fmaior].intValue()))
          fmaior=f2;
        if (fmaior <= i)
        {
          if (vet[pai].intValue() < vet[fmaior].intValue())
            trocaPosicao(vet,pai,fmaior);
        }
      }
      trocaPosicao(vet,i,0);
      //salvando os estados
      c.save(new Integer(i-1));
      c.save(vet);
      c.checkpoint_here();
      if (((int)((tl-i)*100/tl))!=per)
      {
        per=(int)((tl-i)*100/tl);
        System.out.println(per+"%");
      }
    }
    System.out.println(100+"%");
    return vet;
  }//heapSort

//-------------------------------------------------------------------------
  public void exibe(Integer vet[])
  {
    for (int i=0 ; i<vet.length ; i++)
    {
      System.out.println(vet[i].intValue());
    }
  }//exibe

//-------------------------------------------------------------------------
  public void fim()
  {
    c.tempoFinalEstatistica();
  }//fim

//-------------------------------------------------------------------------
  public static void main(String args[])
  {
    HeapSort h = new HeapSort();
    Integer v[] = new Integer[100000];
    for (int i=0 ; i<v.length ; i++)
      v[i] =  new Integer(100000-i);
    v=h.heapSort(v,args);
    System.out.println("Vetor Ordenado");
    h.exibe(v);
    h.fim();
```

```
  }//main

}//HeapSort class
```

Anexo 2.5 GausPiv.java

```
import java.io.*;

public class GausPiv
{

  Checkpointing c = new Checkpointing();

  public final int TAM = 30;

  FileOutputStream arquivo_out;
  DataOutputStream string_out;
  File arq;

  Double matriz[][] = new Double[TAM][TAM];
  Double vetor[] = new Double[TAM];
  Double vetaux[] = new Double[TAM];

  public GausPiv()
  {
    c.tempoInicialEstatistica("GausPiv.dat");
    try
    {
      arq = new File("GausPiv_result.txt");
      arq.createNewFile();
      arquivo_out = new FileOutputStream(arq);
      string_out  = new DataOutputStream(arquivo_out);
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//GausPiv constructor

//-------------------------------------------------------------------------
//... metodo para carregar a MATRIZ dos coeficientes (A) e a matriz dos ......
//... termos independentes (VETOR B).....................................
  public void leMatrizes()
  {
    FileInputStream arquivo_in;
    DataInputStream string_in;
    File arq;
    String nomeArquivo="GausPiv.txt";
                               //Arquivo origem da MATRIZ dos coeficientes e
                                //do VETOR dos termos independentes

    String linha, snum;
    int k,tam,i,j;
    double num;

    try
    {
      arq = new File(nomeArquivo);
      arquivo_in = new FileInputStream(arq);
      string_in  = new DataInputStream(arquivo_in);

      while ((linha=string_in.readLine())!=null)
      {
        i=0;
        j=0;
        //... MATRIZ dos coeficientes (A) ...
        if (linha.charAt(0)=='A')
        {
```

```
          while (((linha=string_in.readLine())!=null) &&
                                                    (linha.charAt(0)!='X'))
          {
            k=0;
            tam=linha.length();
            while (k<tam)
            {
              while ((k<tam) && (linha.charAt(k)==','))
                k++;
              snum="";
              while ((k<tam) && (linha.charAt(k)!=','))
              {
                snum+=linha.charAt(k);
                k++;
              }
              matriz[i][j] = new Double(Double.parseDouble(snum));
              j++;
            }
            j=0;
            i++;
          }
        }
        i=0;

        if (linha.charAt(0)=='X')
        {
          while (((linha=string_in.readLine())!=null) &&
                                                    (linha.charAt(0)!='B'))
          {
          }
        }

        //... vetor dos termos independentes VETOR (B) ...

        if (linha.charAt(0)=='B')
        {
          while ((linha=string_in.readLine())!=null)
          {
            vetor[i] = new Double(Double.parseDouble(linha));
              i++;
          }
        }
      }
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//leMatrizes

//-------------------------------------------------------------------------
  public void mostraMatriz()
  {
    int i,j;

    try
    {
      for (i=0 ; i<TAM ; i++)
      {
        for (j=0 ; j<TAM ; j++)
          string_out.writeBytes(matriz[i][j]+"  ");
        string_out.writeBytes("\n");
      }
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//mostraMatriz

//-------------------------------------------------------------------------
//....... metodo para verificar se algum elemento da diagonal e 0, pois .....
//....... se tivermos algum 0 o sistema nao admite solucao ou o sistema e ....
//....... imcompativel ..................................................
  public double determinante()
  {
```

```
    int i;
    double vdet=1;
    for (i=0 ; i<TAM ; i++)
    {
      vdet=vdet*matriz[i][i].doubleValue();
    }
      return vdet;
  }//determinante

//-------------------------------------------------------------------------
//.... metodo que faz o calculo da matriz triangular superior usando .....
//.... o processo de retrosubstituicao ...................................
  public void sistemaLinearSuperior()
  {
    int i,j;
    double aux,aux2;

    for (i=0 ; i<TAM; i++)
      vetaux[i]=new Double(1);
    for (i=TAM-1 ; i>=0 ; i--)
    {
      aux=0;
      aux2=0;
      for (j=0 ; j<TAM ; j++)
      {
        if (j==i)
          aux=matriz[i][j].doubleValue();
        if (j>i)
          aux2=aux2+(matriz[i][j].doubleValue()*vetaux[j].doubleValue());
      }
      vetaux[i]=new Double((vetor[i].doubleValue()-aux2)/aux);
    }
  }//sistemaLinearSuperior

//-------------------------------------------------------------------------
//... metodo auxiliar usado no pivoteamento do metodo gaussiano ..........
  public void troca(int w, int posm)
  {
    Double vetaux_pivo[] = new Double[TAM];
    double aux_pivo;
    int c;

    for (c=0 ; c<TAM ; c++)
    {
      vetaux_pivo[c]=matriz[w][c];
      matriz[w][c]=matriz[posm][c];
      matriz[posm][c]=vetaux_pivo[c];
    }
    aux_pivo=vetor[w].doubleValue();
    vetor[w]=vetor[posm];
    vetor[posm]=new Double(aux_pivo);
  }//troca

//-------------------------------------------------------------------------
//.... metodo para resolver o metodo gaussiano com pivoteamento ..........
  public void eliminacaoGaussiana(String args[])
  {
    int i,k,w,p,posm,a,b,per=0,ini=0;
    double aux,maior,aux_pivo;

    try
    {
      if (c.verificaRecuperacao(args))
      {
        //restaurando os estados
        c.recover_oockpt();
        matriz=(Double[][])c.recover();
        vetor=(Double[])c.recover();
```

```
      vetaux=(Double[])c.recover();
      ini=(((Integer)c.recover()).intValue());
     }

     for (w=ini ; w<TAM-1 ; w++) //incrementador dos pivos
     {
    //pivoteamento - procura o maximo elemento e se este for maior que o pivo
      //troca as linhas correspondentes entre o maximo e o pivo
      maior=matriz[w][w].doubleValue();
      posm=w;
      for (p=1 ; p<TAM ; p++)
      {
        if (matriz[p][w].doubleValue() > maior)
        {
          posm=p;
          maior=matriz[p][w].doubleValue();
        }
      }
      troca(w,posm);

      //... metodo gaussiano ....
      for (i=w+1 ; i<TAM ; i++)
      //processo que zera todos os elementos abaixo do pivo
      {
        aux=matriz[i][w].doubleValue()/matriz[w][w].doubleValue();
        //calcula o m21, m31, ...
        string_out.writeBytes("m "+(i+1)+" "+(w+1)+"=>"+
                  matriz[i][w].doubleValue()+"/"+matriz[w][w].doubleValue()+
                                                          "="+aux+"\n\n");
        for (k=0 ; k<TAM ; k++)
        {
          vetaux[k]=new Double(aux*(-matriz[w][k].doubleValue()));
          matriz[i][k]=new Double(matriz[i][k].doubleValue()+
                                                 vetaux[k].doubleValue());
        }
        vetor[i]=new Double(vetor[i].doubleValue()+
                                       (aux*(-vetor[w].doubleValue())));
        //... imprime passo a passo (m21, m31 ...) a matriz A e B ...
        for (a=0 ; a<TAM ; a++)
        {
          for (b=0 ; b<TAM ; b++)
          {
            string_out.writeBytes(matriz[a][b]+"  ");
          }
          string_out.writeBytes(Double.toString(vetor[a].doubleValue())+
                                                                  "\n");

        }
        string_out.writeBytes("\n");
      }
      //salvando estados
      c.save(matriz);
      c.save(vetor);
      c.save(vetaux);
      c.save(new Integer(w));
      c.checkpoint_here();
      if (((int)(w*100/(TAM-2)))!=per)
      {
        per=(int)(w*100/(TAM-2));
        System.out.println(per+"%");
      }
    }
  }catch(IOException e) { System.out.println("Error: "+e); }
}//eliminacaoGaussiana
```

```
//---------------------------------------------------------------------------
  public void cabecalho()
  {
    int a,b;

    try
    {
      string_out.writeBytes("-- Metodo Gaussiano com Pivoteamento --\n");
      string_out.writeBytes("\n-- Valores Iniciais --\n");
      string_out.writeBytes(
       "\nMATRIZ dos coeficientes (A) - matriz quadrada ["+TAM+"x"+TAM+"]\n");
      mostraMatriz();
      string_out.writeBytes("\nvetor dos termos independentes VETOR (B)\n");
      for (a=0 ; a<TAM ; a++)
      {
        string_out.writeBytes(vetor[a]+"\n");
      }
      string_out.writeBytes("-----------------------------------------\n\n");
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//cabecalho

//---------------------------------------------------------------------------
  public void resultadoSistema()
  {
    int i;

    try
    {
      string_out.writeBytes("-----------------------------------------\n");
      string_out.writeBytes("-- Resultado Final --\n");
      string_out.writeBytes(
                        "MATRIZ dos coeficientes (A) - matriz triangular\n");
      mostraMatriz();
      string_out.writeBytes("\n");
      string_out.writeBytes("vetor dos termos indepentes VETOR (B)\n");
      string_out.writeBytes("B = [");
      for (i=0 ; i<TAM ; i++)
      {
        string_out.writeBytes(Double.toString(vetor[i].doubleValue()));
        if (i < TAM-1)
          string_out.writeBytes(",");
      }
      string_out.writeBytes("]\n\n");

      string_out.writeBytes(
                       "matriz resultante do sistema triangular VETOR (C)\n");
      string_out.writeBytes("C = [");
      for (i=0 ; i<TAM ; i++)
      {
        string_out.writeBytes(Double.toString(vetaux[i].doubleValue()));
        if (i < TAM-1)
          string_out.writeBytes(",");
      }
      string_out.writeBytes("]\n\n");
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//resultadoSistema

//---------------------------------------------------------------------------
  public void fim()
  {
    c.tempoFinalEstatistica();
  }//fim

//---------------------------------------------------------------------------
  public static void main(String args[])
  {
    GausPiv g = new GausPiv();
```

```
    double vdet;

    g.leMatrizes();
    g.cabecalho();
    g.eliminacaoGaussiana(args);
    vdet=g.determinante();

    if (vdet!=0)
    {
      g.sistemaLinearSuperior();
      g.resultadoSistema();
    }
    else
    {
      g.cabecalho();
      System.out.println("O sistema tem infinitas solucoes ou o sistema");
      System.out.println("nao admite solucao (sistema incompativel)");
    }
    g.fim();
  }//main

}//GausPiv class
```

Anexo 2.6 ConstroiMatriz.java

```
import java.util.Random;
import java.io.*;

public class ConstroiMatriz
{
  public final int TAM = 30;
  public final int VALMAX = 10;

  FileOutputStream arquivo_out;
  DataOutputStream string_out;
  File arq;
  String nomeArquivo;

  Double matriz[][] = new Double[TAM][TAM];
  Double vetorX[] = new Double[TAM];
  Double vetorB[] = new Double[TAM];

//-------------------------------------------------------------------------
  public ConstroiMatriz(String nomeArquivo)
  {
    this.nomeArquivo=nomeArquivo;
    try
    {
      arq = new File(nomeArquivo);
      arq.createNewFile();
      arquivo_out = new FileOutputStream(arq);
      string_out  = new DataOutputStream(arquivo_out);
    }catch(IOException e) { System.out.println("Error: "+e); }
  }//ConstroiMatriz constructor

//-------------------------------------------------------------------------
  public double trunca(double valor, int casas)
  {
    double retorno;
    String saux, saux2;
    int tam;
    saux=Double.toString(valor);
    saux2= new String(saux);
```

```
    tam=saux2.length();
    int i=0;
    while ((i<tam) && (saux2.charAt(i)!='.'))
      i++;
    if ((i+2)==tam)
      retorno=valor;
    else
      retorno=Double.parseDouble(saux2.substring(0,i+casas+1));
    return retorno;
  }//trunca

//---------------------------------------------------------------------------
  public void leMatrizes()
  {
    int i,j;
    double aux,dec,valor,sinal;
    String saux,saux2;

    //... MATRIZ dos coeficientes (A) ...
    Random r = new Random();
    for (i=0 ; i<TAM ; i++)
    {
      for (j=0 ; j<TAM ; j++)
      {
        aux=Math.random();
        dec=Math.random();
        if (dec<=0.5)
          aux=0;
        valor=r.nextInt(VALMAX)+aux;
        sinal=Math.random();
        if ((sinal<=0.25) && (valor!=0.0))
          valor*=-1;
        valor=trunca(valor,2);
        matriz[i][j]= new Double(valor);
      }
    }

    //... VETOR X ...
    for (i=0 ; i<TAM ; i++)
    {
      aux=Math.random();
      dec=Math.random();
      if (dec<=0.5)
        aux=0;
      valor=r.nextInt(VALMAX)+aux;
      sinal=Math.random();
      if ((sinal<=0.25) && (valor!=0.0))
        valor*=-1;
      valor=trunca(valor,2);
      vetorX[i]= new Double(valor);
    }

    //... multiplica MATRIZ com VETOR X (gera o VETOR B) ...
    for (i=0  ; i<TAM ; i++)
    {
      for (j=0 ; j<TAM ; j++)
      {
        double s=0;
        for (int k=0 ; k<TAM ; k++)
          s+=(matriz[i][k].doubleValue()*vetorX[k].doubleValue());
        vetorB[i] = new Double(trunca(s,2));
      }
    }
  }//leMatrizes
```

```
//---------------------------------------------------------------------------
  public void mostraSalvaMatriz_VetorX()
  {
    int i,j;

    try
    {
      string_out.writeBytes("A MATRIZ dos coeficientes (A)\n");
    }catch(IOException e) { System.out.println("Error: "+e); }
    System.out.println("... MATRIZ dos coeficientes (A) ...");
    for (i=0 ; i<TAM ; i++)
    {
      for (j=0 ; j<TAM ; j++)
      {
        if (j<TAM-1)
        {
          try
          {
            string_out.writeBytes(matriz[i][j]+",");
          }catch(IOException e) { System.out.println("Error: "+e); }
          System.out.print(matriz[i][j]+",");
        }
        else
        {
          try
          {
           string_out.writeBytes(Double.toString(matriz[i][j].doubleValue()));
          }catch(IOException e) { System.out.println("Error: "+e); }
          if (i<TAM-1)
            System.out.print(matriz[i][j]+",");
          else
            System.out.print(matriz[i][j]);
        }
      }
      try
      {
        string_out.writeBytes("\n");
      }catch(IOException e) { System.out.println("Error: "+e); }

      System.out.println(" ");
    }

    System.out.println("\n... VETOR X ...");
    try
    {
      string_out.writeBytes("X vetor das aproximacoes VETOR (X)\n");
    }catch(IOException e) { System.out.println("Error: "+e); }
    for (i=0 ; i<TAM ; i++)
    {
      try
      {
        string_out.writeBytes(vetorX[i]+"\n");
      }catch(IOException e) { System.out.println("Error: "+e); }
      if ((i>=0) && (i<TAM-1))
        System.out.println(vetorX[i]+",");
      else
      {
        System.out.println(vetorX[i]);
      }
    }
  }//mostraSalvaMatriz_VetorX

//---------------------------------------------------------------------------
  public void mostraSalvaVetorB()
  {
    try
    {
```

```
      string_out.writeBytes("B vetor dos termos independentes VETOR (B)\n");
    }catch(IOException e) { System.out.println("Error: "+e); }
    System.out.println("\n... VETOR B ...");
    for (int i=0 ; i<TAM ; i++)
    {
      try
      {
        string_out.writeBytes(vetorB[i]+"\n");
      }catch(IOException e) { System.out.println("Error: "+e); }
      if ((i>=0) && (i<TAM-1))
        System.out.println(vetorB[i]+",");
      else
      {
        System.out.println(vetorB[i]);
      }
    }
  }//mostraSalvaVetorB

//--------------------------------------------------------------------------
  public static void main(String args[])
  {
    ConstroiMatriz cm = new ConstroiMatriz("mv.txt");

    cm.leMatrizes();
    cm.mostraSalvaMatriz_VetorX();
    cm.mostraSalvaVetorB();
  }//main

}//ConstroiMatriz class
```

Anexo 2.7 Aplicação RMI (Loja de Suprimentos Java)

Anexo 2.7.1 LojaClient.java

```
import java.awt.*;
import java.awt.event.*;
import java.rmi.*;

public class LojaClient extends Frame implements ActionListener
{
  Button b_compra = new Button("Comprar");
  Button b_atualiza = new Button("Atualizar");
  Label label_Itens = new Label("Suprimentos");
  Label label_Estoque = new Label("Estoque");
  List l_Itens = new List(8);
  List l_Quantidades = new List(8);
  LojaInterface mi;

//--------------------------------------------------------------------------
  public LojaClient()
  {
    super();
    Insets in = getInsets();
    setSize(430,180);
    setLocation(200,200);
    setTitle("Loja de Suprimentos Java");
    setLayout(null);

    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
```

```
    add(label_Itens);
    label_Itens.setBounds(in.left+150,in.top+30,80,20);

    add(label_Estoque);
    label_Estoque.setBounds(in.left+290,in.top+30,50,20);

    add(l_Itens);
    l_Itens.setBounds(in.left+150,in.top+55,120,100);

    add(l_Quantidades);
    l_Quantidades.setBounds(in.left+290,in.top+55,120,100);

    add(b_atualiza);
    b_atualiza.setBounds(in.left+30,in.top+90,70,30);
    b_atualiza.addActionListener(this);

    add(b_compra);
    b_compra.setBounds(in.left+30,in.top+55,70,30);
    b_compra.addActionListener(this);

    setUp();

    setVisible(true);
  }//LojaClient constructor

//---------------------------------------------------------------------------
  public void setUp()
  {
    try
    {
      System.setSecurityManager(new RMISecurityManager());
      mi = (LojaInterface) Naming.lookup("LojaServer");

      String[] Items = mi.listItems();
      for (int i=0 ; i<Items.length ; i++)
        l_Itens.addItem(Items[i]);

      String[] Quantidades = mi.listQuantidades();
      for (int i=0 ; i<Quantidades.length ; i++)
        l_Quantidades.addItem(Quantidades[i]);
    }catch (Exception e) { e.printStackTrace(); }
  }//setUp

//---------------------------------------------------------------------------
  public void actionPerformed(ActionEvent e)
  {
    if (e.getSource().equals(b_compra))
    {
      try
      {
        String item = l_Itens.getSelectedItem();
        int pos = l_Itens.getSelectedIndex();
        if (item!=null)
        {
          if (mi.existeQuantidade(pos))
          {
            mi.compraItem(item,pos);
            l_Quantidades.removeAll();
            String[] Quantidades = mi.listQuantidades();
            for (int i=0 ; i<Quantidades.length ; i++)
              l_Quantidades.addItem(Quantidades[i]);
          }
        }
      }catch (java.rmi.RemoteException re){ re.printStackTrace(); }
    }
    else
    if (e.getSource().equals(b_atualiza))
```

```
    {
      try
      {
        l_Quantidades.removeAll();
        String[] Quantidades = mi.listQuantidades();
        for (int i=0 ; i<Quantidades.length ; i++)
          l_Quantidades.addItem(Quantidades[i]);
      }catch (java.rmi.RemoteException re){ re.printStackTrace(); }
    }
  }//actionPerformed

//-------------------------------------------------------------------------
  public static void main(String[] args)
  {
    new LojaClient();
  }//main

}//LojaClient class
```

Anexo 2.7.2 LojaInterface.java

```
interface LojaInterface extends java.rmi.Remote
{
  public boolean compraItem(String item, int pos)
                                          throws java.rmi.RemoteException;
  public String[] listItems() throws java.rmi.RemoteException;
  public String[] listQuantidades() throws java.rmi.RemoteException;
  public boolean existeQuantidade(int pos) throws java.rmi.RemoteException;
}//LojaInterface interface
```

Anexo 2.7.3 LojaServer.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.io.*;
import java.util.Vector;

public class LojaServer extends UnicastRemoteObject implements LojaInterface
{
  Checkpointing c = new Checkpointing();

  private Vector Items;
  private Vector Quantidades;

//-------------------------------------------------------------------------
  public LojaServer(String[] args) throws RemoteException
  {
    super();
    Items = new Vector(5,5);
    Quantidades = new Vector(5,5);
    startUp(args);
  }//LojaServer constructor

//-------------------------------------------------------------------------
  public boolean compraItem(String item, int pos) throws RemoteException
  {
    System.out.println("Recebendo pedido de compra: "+item+".\n");
    if (Items.contains(item))
    {
      int valor = Integer.parseInt((String)Quantidades.elementAt(pos));
      valor--;
```

```
      Quantidades.removeElementAt(pos);
      Quantidades.insertElementAt(Integer.toString(valor),pos);
      c.save(Items);
      c.save(Quantidades);
      c.checkpoint_here();
      return true;
    }
    return false;
  }//compraItem

//-------------------------------------------------------------------------
  public boolean existeQuantidade(int pos) throws RemoteException
  {
    if ((Integer.parseInt((String)Quantidades.elementAt(pos)))>0)
      return true;
    else
      return false;
  }//existeQuantidade

//-------------------------------------------------------------------------
  public String[] listItems() throws RemoteException
  {
    System.out.println("Recebido pedido de listagem de itens.\n");
    System.out.println("Enviando resposta.\n");
    String[] Aux = new String[Items.size()];
    for (int i=0 ; i<Aux.length ; i++)
      Aux[i] = (String) Items.elementAt(i);
    return Aux;
  }//listItems

//-------------------------------------------------------------------------
  public String[] listQuantidades() throws RemoteException
  {
    String[] Aux = new String[Quantidades.size()];
    for (int i=0 ; i<Aux.length ; i++)
      Aux[i] = (String)Quantidades.elementAt(i);
    return Aux;
  }//listQuantidades

//-------------------------------------------------------------------------
  public void startUp(String[] args)
  {
    try
    {
      if (c.verificaRecuperacao(args))
      {
        //restaurando os estados
        c.recover_oockpt();
        Items=(Vector)c.recover();
        Quantidades=(Vector)c.recover();
      }
      else
      {
        Items.addElement("Cartucho de Tinta");
        Items.addElement("Fita de Impressora");
        Items.addElement("Disquete");
        Items.addElement("Mouse");
        Items.addElement("CDR");
        Quantidades.addElement("10");
        Quantidades.addElement("10");
        Quantidades.addElement("10");
        Quantidades.addElement("10");
        Quantidades.addElement("10");
      }
      System.out.println("Lista de Produtos registrados:");
      for (int i=0;i<Items.size();i++)
        System.out.println(Items.elementAt(i));
```

```
      System.out.println("Registrando-se no servidor de nomes...\n");
      System.setSecurityManager(new RMISecurityManager());
      Naming.rebind("LojaServer", this);
      System.out.println("Registro concluido com sucesso.
                                    Aguardando chamadas remotas.\n");
    }catch (Exception e) {e.printStackTrace();}
  }//startUp

//--------------------------------------------------------------------------
  public static void main(String[] args)
  {
    try
    {
      new LojaServer(args);
    }
    catch (Exception e) { e.printStackTrace(); }
  }//main

}//LojaServer class
```

Anexo 3 Metodologia e cálculos estatísticos

Segue uma breve explicação nos parágrafos deste anexo, da metodologia e cálculos estatísticos efetuados e apresentados nas tabelas da seção de avaliação de desempenho (seção 5.3).

A média dos tempos apresentada nas tabelas foi calculada segundo a estimativa de média amostral dada pela equação:

$\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$, onde:

$\overline{X}$ é o estimador da média μ_x;

n é o número de observações amostrais.

Foi utilizado o aplicativo Microsoft Excel 2000 para efetuar os cálculos cujos resultados são apresentados na seção de avaliação de desempenho. Para o cálculo de desvio padrão (σ_x), foi utilizada a função **`DESVPAD(núm1;núm2;...)`**, que calcula o desvio padrão a partir de uma amostra dos dados identificados através dos seus argumentos **`núm1;núm2;...`**. O desvio padrão é uma medida do grau de dispersão dos valores em relação à média. O desvio padrão é definido pela raiz quadrada da variância [JAI91]. A variância é definida como: a somatória da distância ao quadrado entre os valores das observações amostrais e a média, dividido pelo número de graus de liberdade (n-1):

$$\mathrm{Var}(x) = \frac{1}{n-1}\left(\sum_{i=1}^{n}(xi - X)^2\right)$$

A largura do intervalo de confiança (IC) foi calculada segundo um desvio padrão conhecido e com o uso da tabela de Distribuição Normal [MEY83] através do coeficiente de confiança. Distribuição Normal é a distribuição de probabilidade mais comumente utilizada em aplicações estatísticas para análise de dados [JAI91].

A equação a seguir representa o cálculo para obter o coeficiente de confiança:

$\Phi(k_{1-\alpha/1}) = 1 - \frac{\alpha}{2}$, onde

$(1-\alpha)$ = 95, então α = 0,05 e $1-\frac{\alpha}{2}$ = 0,975(este é o valor procurado na tabela de Distribuição Normal)

A largura do intervalo de confiança é calculada segundo a equação:

$$L = \frac{2\sigma}{\sqrt{n}} k_{1-\alpha/2}$$

Bibliografia

[AMA99] AMARAL, J. B.; BERTAGNOLLI, S. C.; LISBÔA, M. L. B. Componentes para Apoiar o Desenvolvimento de Aplicações Tolerantes a Falhas. In: SIMPÓSIO DE COMPUTAÇÃO TOLERANTE A FALHAS, 8., 1999, Campinas, SP. **Anais...** Campinas: Unicamp, 1999. 239p. p.142-146.

[AMA2001] AMARAL, J. B. **Definição de classes para comunicação Unicast e Multicast**. 2001. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[AND81] ANDERSON, T.; LEE, P. A. **Fault Tolerance:** Principles and Practice. Englewood Cliffs: Prentice-Hall, 1981. 396p.

[ARN97] ARNOLD, K.; GOSLING, J. A. **Programando em Java**. São Paulo: Makron Books, 1997. 353p.

[AZE96] AZEREDO, P. A. **Métodos de classificação de dados e análise de suas complexidades**. Rio de Janeiro: Campus, 1996. 132p.

[BAN86] BANÂTRE, J. P.; BANÂTRE, M.; LAPALME G.; PLOYETTE, F. The Design and Building of Enchère, a Distributed Electronic Marketing System. **Communications of the ACM**, v.29, n.1, p.19-29, Jan. 1986.

[BAN88] BANÂTRE, J. P.; BANÂTRE, M.; MULLER, G. **Ensuring Data Security and Integrity with a Fast Stable Storage**. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING. 4., 1998. **Proceedings…** Los Angeles, California, USA:IEE Computer Society, 1988. p.285-293.

[BER2000] BERTAGNOLLI, S. C. **Integrando aspectos de distribuição e de Tolerância a Falhas no ambiente Java Reflexivo (Jreflex)**. 2000. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[BOO2000] BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML – Guia do usuário**. 3. ed. Rio de Janeiro:Campus, 2000. 472p.

[BUR97] BURRIS, E. The RMI Package. In: **Java 1.1: Unleashed**. 3rd ed. Indianapolis: Sams.Net, 1997.

[BUZ98] BUZATO, L. E.; RUBIRA, C. M. F. Construção de Sistemas Orientados a Objetos Confiáveis. In: ESCOLA DE COMPUTAÇÃO, 11., 1998, Rio de Janeiro. **Anais...** Rio de Janeiro: DCC\IM; COPPE Sistemas; NCE\UFRJ, 1998.

[CEC2002] CECHIN, S. L. **Protocolo de recuperação por retorno, coordenado, não determinístico**. 2002. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[COK97] COKER, J. **Object Persistence and Distribution**. Feb. 1997 Disponível em: <http://developer.java.sun.com/developer/technicalArticles/RMI/ObjectPersist/index.html>. Acesso em: 10 maio 2001.

[CUN2001] CUNHA, J. C; SILVA, J. G. Software-Implemented Stable Storage in Main Memory. In: THE BRAZILIAN SYMPOSIUM ON FAULT-TOLERANT COMPUTING, SCTF, 9., 2001. **Proceedings...** Florianópolis:DAS\UFSC, 2001. p.43–57.

[ECK98] ECKEL, B. **Thinking in Java**. New Jersey: Prentice-Hall, 1998. 848p.

[ELN92] ELNOZAHY, E. N.; JOHNSON, D. B.; ZWAENPOEL, W. The Performance of Consistent Checkpointing. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 11., 1992. **Proceedings...** Houston, Texas:IEEE Computer Society, 1992. p.39-47.

[ERI98] ERIKSSON, H.; PENKER, M. **UML Toolkit**. New York:John Wiley, 1998. 416p.

[FER2002] FERREIRA, A.; JANSCH-PÔRTO, I. **User Level Checkpointing libraries:** Comparative Study and Tool Analysis. Porto Alegre: PPGC da UFRGS, 2002. 45p.

[FOW2000] FOWLER, M.; SCOTT, K. **UML Essencial – Um breve guia para a linguagem-padrão de modelagem de objetos**. 2. ed. Porto Alegre: Bookman, 2000. 169p.

[FUR98] FURLAN, J. D. **Modelagem de Objetos através da UML:** the unified modeling language. São Paulo:Makron Books, 1998. 329p.

[GAR98] GARBINATO, B. **Protocol Objects and Patterns for Structuring Reliable Distributed Systems**. 1998. Tese (Doutorado em Ciência da Computação) – École Polytechnique Fédérale de Lausanne.

[GRY94] GRYGIER, A. DAL CIN, M. Stable Object Store for Multiprocessors with Distributed Shared Memory. In: INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED REAL-TIME SYSTEMS. 1994. **Proceedings…** Dana Point, California:[s.n.], 1994.

[HAN99] HANSEN, E. K. **Checkpointing Java Programs on Standard Java Implementations using Program Transformation**. Master's Thesis. Department of Computer Science, November - University of Copenhagen. Nov. 1999. Disponível em: <http://hjem.get2net.dk/esben_krag_hansen/jcp/index.html>. Acesso em: 10 maio 2001.

[JAI91] JAIN, R. **The Art of Computer Systems Performance Analysis:** Techniques for Experimental Design, Measurement, Simulation and Modeling. New York: John Wiley, 1991. 685p.

[JAV98] JAVA Object Serialization Specification. Java Software, 1998. Disponível em: <http://java.sun.com/products/jdk/1.2/docs/guide/serialization/>. Acesso em: 10 maio 2001.

[JOR98] JORDAN, M.; ATKINSON, M. Orthogonal persistence for Java – a mid-term report. In: INTERNATIONAL WORKSHOP ON PERSISTENCE AND JAVA. 3., 1998. **Proceedings…** Tiburon, California:[s.n.], 1998. p.1-3.

[KAS99] KASBEKAR, M.; DAS, C. R.; YAJNIK, S.; KLEMM, R.; HUANG, Y. Issues in the Design of a Reflective Library for Checkpointing C++

Objects. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 18., 1999. **Proceedings…** Lausanne: IEEE, 1999. p.224-233.

[KHO90] KHOSHAFIAN S.; ABNOUS, R. **Object Orientation – Concepts, Languages, Databases, User Interfaces**. New York: John Wiley and Sons, 1990.

[KIL99] KILLIJIAN, M. O.; FABRE, J. C.; RUIZ-GARCIA, J. C. **Using compile-time reflection for object checkpointing**. Laboratoire d'Analyse et d'Architecture des Systèmes, 1999. (Technical Report-99049).

[LAW2000] LAWALL, J. L.; MULLER, G. Efficient Incremental Checkpointing of Java Programs. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, DSN, 2000, New York. **Proceedings...** New York: IEEE, 2000. p.61-70.

[LIS95] LISBOA, M. L. B. **MOTF: Meta-Objetos para Tolerância a Falhas**. 1995. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[LIT97] LITZKOW, M.; TANNENBAUM, T; BASNEY, J.; LIVNY, M. **Checkpoint and Migration of UNIX Processes in Condor Distributed Processing System**. Computer Sciences Department, University of Wisconsin-Madison, 1997. (Technical Report-1346).

[MAT97] MATHIS, J. Persistence and Java Serialization. **In: Java 1.1: Unleashed**. Indianapolis:Sams.Net, 3. ed., 1997.

[MEY83] MEYER, P. L. **Probabilidade – Aplicações à Estatística**. 2. ed. Rio de Janeiro:LTC, 1983. 428p.

[MEY88] MEYER, B. **Object-Oriented Software Construction**. New York: Prentice-Hall, 1998. 534p.

[NAU96] NAUGTON, P. **Dominando o Java**. São Paulo: Makron Books, 1996.

[NEL91] NELSON, M. **The Data Compression Book:** Featuring fast, efficient data compression techniques in C. U.S.A.:M&T Books, 1991.

[PIN98] PINHEIRO, E. **Truly-Transparent Checkpointing of Parallel Applications**. COPPE/UFRJ internal report, 1998. Disponível em: <http://athos.rutgers.edu/~edpin/epckpt/paper_html/>. Acesso em: 09 maio 2001.

[PLA95] PLANK, J. S.; BECK, M.; KINGSLEY, G.; LI, K. Libckpt: Transparent Checkpointing under Unix. In: USENIX TECHNICAL CONFERENCE, 1995. **Proceedings…** New Orleans:[s.n.], 1995. p.213-223.

[PLA97] PLANK, J. S. **An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance**. Knoxville: Departament of Computer Science – University of Tennenssee, 1997. (Technical Report UT-CS-97-372)

[PLA98] PLANK, J. S.; PUENING, M. A. Diskless Checkpointing. **IEEE Trans. on Parallel and Distributed Systems**, v.9, n.10, p.972-986, Oct. 1998.

[PLA99] PLANK, J. S.; CHEN, Y.; LI, Kai; BECK, M. et al. Memory exclusion: Optimizing the performance of checkpointing systems. **Software–Practice and Experience**, v.29, n.2, p.125–142, 1999.

[RAM97] RAMKUMAR, B.; STRUMPEN, V. Portable checkpointing for heterogeneous architectures. In: THE ANNUAL INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, FTCS, 1997. **Proceedings...** Seattle:WA, June 1997. p.58-67.

[RAN79] RANDELL, B. **Realible Computing Systems, Operation Systems**: An Advanced Course. New York: Springer-Verlag, 1979. p.282-391.

[REI2000] REINHOLTZ, K. Java will be faster than C++. **ACM SIGPLAN Notices**, New York, v. 35, n.2, p.25-28, Feb. 2000.

[RIC2001] RICARTE, I. L. M. **Programação Orientada a Objetos:** Uma Abordagem com Java. Campinas:Unicamp, 2001. Disponível em: <http://www.dca.fee.unicamp.br/courses/PooJava/poojava.pdf>. Acesso em: 09 maio 2001.

[RMI97] RMI Documentation. 1997. Disponível em: <http://java.sun.com/ products/jdk/1.1/docs/guide/rmi/index.html>. Acesso em: 20 abr. 2001.

[ROC98] ROCHA, H. L. S. Usando Java em Ambientes Distribuidos. In: JAVA OPEN BRASIL, 1., 1998, Brasília, DF. **Anais...** Brasília: [s.n.], 1998.

[RUG88] RUGGIERO, M. G.; LOPES, V. L. R. **Cálculo Numérico:** Aspectos Teóricos e Computacionais. São Paulo: McGraw-Hill, 1988. 295p.

[RUM94] RUMBAUGH, J. **Modelagem e Projetos Baseados em Objetos**. Rio de Janeiro:Campus, 1994. 625p.

[SIL2001] SILVA, F. A.; JANSCH-PÔRTO, I. **Instalação e Análise de uma Biblioteca para Recuperação de Processos com base em Checkpointing**. Janeiro 2001. 51p. Disponível em: <http://www2.unoeste.br/~chico/ti_chico.pdf>. Acesso em: 16 set. 2002.

[SIL2002] SILVA, F. A.; JANSCH-PÔRTO, I.; LISBOA, M. L. Recuperação com Base em Checkpointing: uma Abordagem Orientada a Objetos. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, SBRC, 20., 2002, Búzios, Rio de Janeiro:Núcleo de Computação e Eletrônica – Universidade Federal do Rio de Janeiro, 2002. 108p. p.69-76.

[SIN94] SINGHAL, N.; SHIVARATRI, N. G. **Advanced Concepts in Operation Systems Distributed, Database and Multiprocessor Operation Systems**. New York: McGraw-Hill, 1994. 522p.

[SUN98] SUN MICROSYSTEMS. **Java Object Serialization Specification – JDK 1.2 Beta 3**. Feb. 1998. Disponível em: <ftp://ftp.javasoft.com/docs/jdk1.2/serial-spec-JDK1.2.ps>. Acesso em: 20 abr. 2001.

[THO97] THOMAS, M. D. **Programando em Java para a Internet**. São Paulo:Makron Books, 1997. 665p.

[WIR89] WIRTH, N. **Algoritmos e Estrutura de Dados**. Rio de Janeiro: Prentice-Hall do Brasil, 1989. 255p.