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

Francisco Assis da Silva[1,2], Ingrid Jansch-Pôrto[2], Maria Lúcia Lisbôa[2]

chico@apec.unoeste.br, ingrid@inf.ufrgs.br, llisboa@inf.ufrgs.br

[1] Faculdade de Informática de Pres. Prudente
Universidade do Oeste Paulista – UNOESTE

[2] Pós-Graduação em Ciência da Computação
Instituto de Informática – UFRGS

Resumo

A recuperação de processos visa propiciar maior disponibilidade para aplicações computacionais, minimizando o tempo total de execução quando da ocorrência de falhas. Com o advento do paradigma da orientação a objetos, novos problemas foram introduzidos na atividade de recuperação quanto aos mecanismos de salvamento de estados e retomada da execução. Tendo em vista prover tolerância a falhas para aplicações computacionais na presença de falhas do sistema, e contribuir na execução do processo de recuperação de aplicações orientadas a objetos escritas em Java, objetiva-se neste trabalho o desenvolvimento de uma biblioteca que implementa os mecanismos de checkpointing e recuperação. 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.
Neste artigo, é descrita a biblioteca proposta; são comentados alguns aspectos de implementação e resultados obtidos a partir da avaliação funcional e de desempenho temporal dos mecanismos. A apresentação de conceitos básicos restringe-se aos relacionados à área de orientação a objetos empregados no artigo.

1. Introdução

Na área de Tolerância a Falhas, a busca de parâmetros adequados na propriedade de disponibilidade dos serviços computacionais, mesmo em presença de falhas do sistema, traz consigo o interesse na pesquisa das atividades de recuperação em sistemas computacionais. Do ponto de vista prático, isto se traduz na necessidade de minimizar o tempo total de execução de aplicações computacionais de longa duração, ao mesmo tempo em que as prepara para não sofrerem perdas significativas de desempenho, em caso de falhas. 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).

Os estados salvos durante a execução de um processo, para os quais ele possa mais tarde ser restaurado, são conhecidos como pontos de recuperação ou *checkpoints* [1]. O termo *checkpointing* é usado para a ação de estabelecer o ponto de recuperação. O estado de um programa em execução deve ser periodicamente salvo para um meio estável (protegido contra falhas do meio de armazenamento), do qual pode ser recuperado a partir da detecção do erro [2]. Este enfoque orientado a processos é bastante tradicional; tendo sido explorado em vários trabalhos, incluindo o desenvolvimento de bibliotecas, tais como a *libckpt* [2].

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). Ao buscar os dados para reutilização, a partir do armazenamento estável, eles necessitam ser convertidos da representação externa para a representação interna. Em um programa orientado a objetos, os dados são representados em forma de objetos, e produzir objetos persistentes é uma maneira eficiente de salvar os dados da aplicação no armazenamento estável [3].

Segundo Lawall e Muller [4], programas escritos em uma linguagem orientada a objetos, tal como Java, introduzem novas demandas de *checkpointing*: (i) o estilo de programação orientado a objetos propicia a criação de muitos objetos pequenos. Cada objeto pode ter alguns campos que são somente de leitura, e outros que são freqüentemente modificados; (ii) o programador de Java não tem nenhum controle sobre a alocação de objetos. Assim, é impossível assegurar que objetos modificados freqüentemente estejam 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 desalocados da memória; (iii) 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 o *checkpointing* realizado em nível de linguagem aumenta o tamanho do programa fonte com um código para registrar os estados dos objetos do programa. Para promover segurança de funcionamento, esse código de *checkpointing* deveria ser introduzido sistematicamente, e interferir o mínimo possível no comportamento padrão do programa. *Checkpointing* incremental pode ser implementado associando uma variável de instância (*flag*) para cada objeto, indicando se o objeto foi modificado desde o último *checkpoint* prévio [3,5].

A atividade de *checkpointing* numa abordagem orientada a objetos visa salvar o estado dos objetos do programa em execução. Essa atividade pode ser implementada utilizando o mecanismo de serialização de objetos em Java. A recuperação baseia-se nos objetos serializados e o reinício da execução da aplicação. A partir da seqüência de *bytes* (objetos serializados no *checkpoint* prévio livres de erros), os objetos podem depois ser recriados por meio da de-serialização dos objetos. 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 [4].

Com base nas considerações expostas, o objetivo central do artigo é descrever as características e aspectos de projeto de uma biblioteca que oferece métodos de *checkpointing* e recuperação e que foi denominada *Libcjp – Library of checkpoints in Java programs*.

Dos trabalhos investigados, o que despertou maior interesse, e pode ser considerado como referência fundamental à biblioteca desenvolvida neste trabalho, é a *libckpt* de Plank *et al*. [2]. Porém, a presente implementação é bastante diferente, visto que explora o modelo de objetos. Os trabalhos de Lawall e Muller [4] e de Killijian *et al* [5] serviram de fonte de conhecimento sobre *checkpointing* orientado a objetos em Java, embora apresente soluções distintas para a atividade de *checkpointing* e recuperação. Assim, neste artigo são apresentados os conceitos básicos de persistência e serialização, as características da biblioteca, alguns aspectos de projeto e implementação, exemplo de uso e resultados preliminares obtidos nos testes realizados.

2. Persistência e serialização

A persistência, em linguagens de programação orientadas 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* [6]. Em Java, essa habilidade de persistência pode ser conseguida através da serialização de objetos.

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 pela rede para outra máquina [7]. O modelo de memória estável escolhido depende das hipóteses de falhas consideradas e deve ser adequadamente implementado.

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, é possível restaurar o estado de execução de P1, utilizando os dados persistentes de *obj1* e *obj2* [7].

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 [8, 9]. 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 estabelecido pelo estado representado pela seqüência de *bytes*; efetivamente é criada uma cópia do objeto serializado, ao invés de sobrescrever o estado de um objeto existente [10].

3. A biblioteca proposta: características e funcionalidades

A biblioteca *Libcjp – Library of checkpoints in Java programs* foi projetada com o propósito de trabalhar com aplicações computacionais escritas na linguagem de programação Java. *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. A biblioteca proposta facilita o estabelecimento de *checkpoints* e a retomada do processamento pós-falhas (ou recuperação) de aplicações orientadas a objetos. Foi desenvolvida com funcionalidades para capturar o estado de um ou mais objetos de uma aplicação em execução, e representá-los no meio de armazenamento (memória secundária) para posterior recuperação, após a ocorrência de falhas. Esta representação dos estados dos objetos da aplicação no disco (usado como opção de armazenamento, por simplicidade, apesar de formalmente não atender aos requisitos do meio estável) foi alcançada utilizando os mecanismos de persistência e serialização presentes na linguagem Java, através da API de serialização Java [8]. A retomada da execução da aplicação ocorre restaurando os estados dos objetos, a partir dos arquivos de *checkpoint* salvos.

Sabe-se que em um ambiente livre de falhas, o uso de mecanismos de salvamento intermediário de estados da computação acarretará queda de desempenho - se comparado à execução sem esta operação adicional. Entretanto, no momento em que ocorre falha, o reinício integral da aplicação pode ser bastante oneroso. A adoção cuidadosa de procedimentos - tais como o uso de técnicas otimizadas e a escolha adequada da freqüência na

execução de salvamentos, planejada de acordo com a probabilidade de falhas - deve trazer um resultado proveitoso no cômputo do desempenho. O projeto da biblioteca levou em conta estas questões, permitindo ao programador estabelecer adequadamente estes parâmetros.

Portanto, alguns benefícios fornecidos pelo enfoque proposto para a biblioteca *Libcjp* podem ser observados: (i) fazendo o uso da biblioteca, consegue-se prover maior disponibilidade para muitas aplicações computacionais orientadas a objetos escritas em Java, aprimorando a retomada da execução, diante da ocorrência de falhas; (ii) diferentes formas de utilização da biblioteca poderão ser feitas pelo programador. Em alguns casos a biblioteca adiciona pequena sobrecarga de processamento para salvamento dos estados dos objetos sobre o tempo de execução da aplicação; (iii) uma das principais vantagens da linguagem Java é a independência de plataforma: um programa Java pode ser executado em qualquer plataforma que possua uma JVM e deverá apresentar o mesmo comportamento em cada uma destas plataformas. A *Libcjp* foi preparada para trabalhar não apenas na plataforma a qual foi desenvolvida (sistema operacional *Windows*), mas poderá ser utilizada também no sistema operacional *Linux*; (iv) os *checkpoints* são independentes de plataforma, ou seja, uma dada aplicação Java poderá ser reiniciada em uma outra JVM de outra plataforma, sem levar em consideração o ambiente de criação dos *checkpoints*.

Para prover os benefícios recém mencionados, a biblioteca agrega alguns custos para sua utilização: (i) *checkpointing* com *Libcjp* não é completamente transparente ao programador da aplicação, é necessário algum esforço de programação para o seu uso. Diz-se que *checkpointing* é transparente quando nenhuma modificação precisa ser feita no código-fonte da aplicação. Tal esforço, pequeno em princípio, dependerá das funcionalidades escolhidas e do perfil de aplicação desenvolvida; (ii) o tamanho do código-fonte da aplicação não deverá aumentar significativamente, sendo acrescidas somente algumas linhas de código para especificar à biblioteca os objetos que deverão estar presentes no arquivo de *checkpoint* e para efetuar a recuperação.

As principais funcionalidades propostas e implementadas são descritas a seguir.

Mecanismo de *checkpointing* tradicional: é o método mais direto para estabelecer 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 meio de armazenamento (arquivo de *checkpoint*) são intercaladas com a execução da aplicação.

Mecanismo de *checkpointing* incremental: embora a idéia seja semelhante ao mecanismo tradicional, através deste mecanismo, somente os estados dos objetos modificados desde o *checkpoint* prévio são salvos no arquivo de *checkpoint*. A base do *checkpointing* incremental é não salvar repetidamente os objetos que não foram modificados.

Em geral, o tamanho de um *checkpoint* **não incremental** não varia significantemente ou cresce 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. 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 estão espalhados em 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 o tamanho de cada arquivo de *checkpoint*.

***Checkpointing* programado**: esta técnica pode ser empregada em conjunto com ambos

mecanismos de *checkpointing*: tradicional e incremental. O programador especifica pontos no código da aplicação onde é vantajoso ou necessário o *checkpointing*. O arquivo de *checkpoint* pode ser estabelecido de duas formas: (i) **executado sempre**: toda vez que a linha de execução da aplicação passar por uma chamada ao método que ativa o mecanismo, o *checkpoint* será estabelecido; (ii) **inibido por controle de tempo**: o *checkpointing* apenas ocorrerá depois de decorrido um período mínimo de tempo, desde o último estabelecimento.

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

Na atividade de recuperação, quando os arquivos de *checkpoint* forem estabelecidos através do mecanismo de *checkpointing* incremental, haverá a necessidade de ser realizada a fusão dos arquivos velhos de *checkpoint*. A atividade de fusão 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* estabelecido previamente 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 cuida para manter armazenada apenas a quantidade de arquivos estipulada.

Registros de informações (tempos): esta funcionalidade tem por objetivo propiciar a obtenção de informações de tempos referentes ao estabelecimento dos arquivos de *checkpoint* e a execução da aplicação.

4. Projeto da biblioteca e exemplo de uso

Inicialmente as classes da biblioteca *Libcjp* foram modeladas usando UML – *Unified Modeling Language*; após, foram implementadas utilizando a linguagem orientada a objetos Java (pacote *jdk1.3.1_01*). A Figura 1 mostra um diagrama contendo um modelo simplificado de classes da biblioteca *Libcjp*.

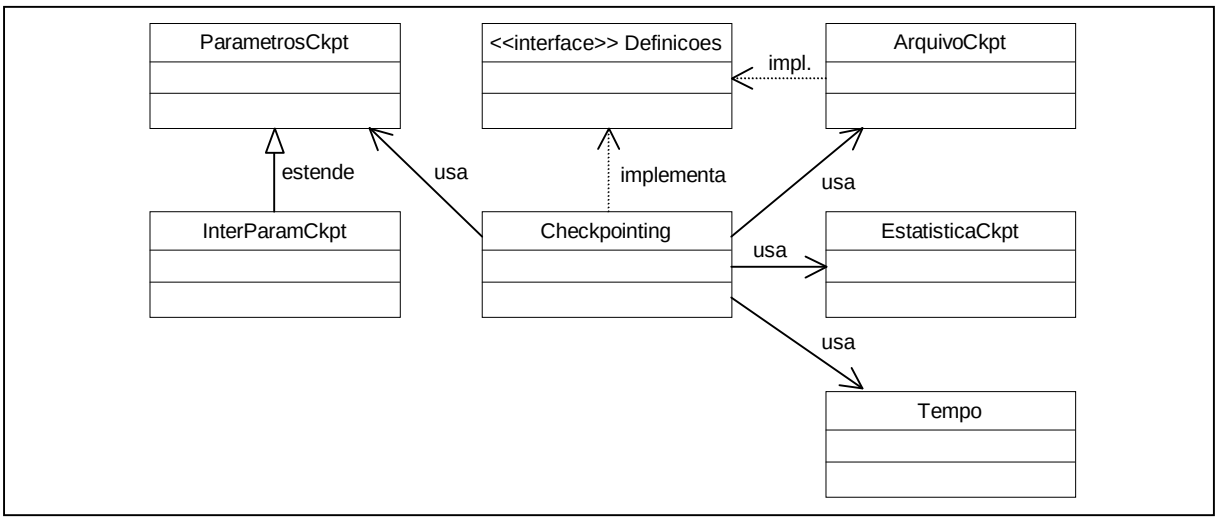

Figura 1 – Modelo simplificado de classes da biblioteca *Libcjp*.

Nesse diagrama, a classe **`Checkpointing`** é a principal, pois nela estão implementados os mecanismos para prover estabelecimento de *checkpoints* e recuperação, e também todo o controle operacional da biblioteca. As classes **`ArquivoCkpt`** e **`Tempo`** dão apoio funcional a

classe **Checkpointing**, no que se refere à manipulação das características dos arquivos de *checkpoint* e controle de tempo para o estabelecimento dos *checkpoints*. A classe **EstatisticaCkpt** faz o registro dos tempos gastos para o estabelecimento de cada *checkpoint* e o tempo total de execução da aplicação. A interface **Definicoes** contém as constantes utilizadas pelas classes **Checkpointing** e **ArquivoCkpt**. As classes restantes **ParametrosCkpt** e **InterParamCkpts** são utilizadas como apoio à biblioteca, provendo um arquivo de parâmetros denominado de **paramckpt.dat**. Estes parâmetros têm por finalidade configurar a biblioteca para alguns tipos diferentes de utilização. Os possíveis parâmetros do arquivo **paramckpt.dat**, seus possíveis valores e seus valores padrão são listados na Figura 2, e explicados a seguir.

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)>
incremental=N	<**S**/**N**>
diretorio=.	<. (diretório desejado)>

Figura 2 – Valores possíveis dos parâmetros do arquivo "**paramckpt.dat**".

O parâmetro "**intervalo**=<*n*>" estabelece um valor correspondente a um período mínimo de tempo que deve decorrer entre um *checkpointing* e outro. A definição de unidade empregada é fornecida através do parâmetro "**tipo_intervalo**=<*H/M/S/L/V*>". O número máximo de arquivos de *checkpoint* para *n* é dado por "**num_max_arquivos**=<*n*>"; este parâmetro pode ser usado tanto para o *checkpointing* incremental ou *checkpointing* tradicional. Valores de *n*>1 permitem ao usuário impor um equilíbrio entre o número de arquivos de *checkpoint* e a quantidade de espaço utilizado no meio de armazenamento. O parâmetro <**incremental**=*S/N*> ativa ou desativa o mecanismo de *checkpointing* incremental. O parâmetro "**diretorio**=<**.**>" especifica o diretório no qual os arquivos de *checkpoint* serão gravados.

5. Avaliação e testes preliminares

Foram desenvolvidas algumas aplicações para testar e avaliar o desempenho funcional da *Libcjp*. As empregadas nos testes preliminares são programas científicos típicos escritos em Java: (i) MAT efetua a multiplicação de duas matrizes 615 x 615 de elementos inteiros; (ii) TDC calcula a transformada discreta do cosseno. Esta transformada é utilizada no processo de compressão do JPEG. Foi utilizada uma matriz 512 x 512, representando as informações de cada *pixel* de uma imagem, em tons de cinza; (iii) SHELL é o método de ordenação – *ShellSort*. Para ordenação, foi utilizado um vetor de 100.000 elementos inteiros, em ordem inversa; (iv) HEAP corresponde ao método de ordenação *HeapSort*. Também foi utilizado como dado inicial um vetor de 100.000 elementos inteiros em ordem inversa; (v) GAUSPIV efetua o método da eliminação gaussiana com pivoteamento. Foi submetido à aplicação para resolução um sistema linear 30 x 30.

Como ambiente de teste para avaliação da *Libcjp*, foi utilizado 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. Cada aplicação foi executada 100 vezes.

A Figura 3 demonstra a utilização da biblioteca *Libcjp* em uma aplicação que efetua a multiplicação de duas matrizes. As modificações e inserções de novas linhas de código para

ativar os mecanismos de *checkpointing* e recuperação (em negrito), são observadas na segunda coluna da figura. As linhas 8, 9 e 10 são responsáveis por ativar o mecanismo de recuperação, as linhas 22, 23 e 24 são responsáveis por ativar o mecanismo de *checkpointing*.

```
public Integer[][] multMatriz(
          Integer m1[][], Integer m2[][])
{
  Integer m3[][] = new Integer[615][615];

  for (int i=0  ; 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);
    }
  }
  return m3;
}
```

(a) programa sem usar a biblioteca

```
 1 public Integer[][] multMatriz(
 2 Integer m1[][], Integer m2[][], String args[])
 3 {
 4   Integer m3[][] = new Integer[615][615];
 5   int ini=0;
 6   if (c.verificaRecuperacao(args))
 7   {
 8     c.recover_oockpt();
 9     m3=(Integer[][])c.recover();
10     ini=(((Integer)c.recover()).intValue());
11   }
12   for (int i=ini  ; i<615 ; i++)
13   {
14     for (int j=0 ; j<615 ; j++)
15     {
16       int s=0;
17       for (int k=0 ; k<0615 ; k++)
18         s+=(m1[i][k].intValue()*
19         m2[k][j].intValue());
20       m3[i][j] = new Integer(s);
21     }
22     c.save(m3);
23     c.save(new Integer(i+1));
24     c.checkpoint_here();
25   }
26   return m3;
27 }
```

(b) programa fazendo o uso da biblioteca

Figura 3 – Exemplo de utilização da biblioteca *Libcjp*.

A Tabela 1 e a Tabela 2 mostram os resultados parciais obtidos nos testes efetuados com o uso da biblioteca, através de experimentos de *checkpointing*. Estes resultados são preliminares; novas análises serão realizadas sobre os dados para se obter resultados mais completos e um estudo estatístico mais elaborado. Na Tabela 1, estão expressos: os tempos médios de execução de cada aplicação sem e com *checkpointing* (em segundos); o desvio padrão (σx); o intervalo de confiança de 95% (IC); a porcentagem de *overhead* (sobrecarga) de processamento sobre a aplicação causada pelo *checkpointing*; o intervalo de tempo entre cada dois *checkpoints* (Δ); e o número de arquivos de *checkpoint* (Nº arq).

Tabela 1 – Resultado dos experimentos de *checkpointing* para cada aplicação.

Aplicação	Exec_s/*ckpt*	Exec_c/*ckpt*	σx	IC 95%	% *overhead*	Δ (seg)	Nº arq.
MAT	106,5350	126,083	12,7712	5,0063	18,3489	10	10
TDC	33,7787	57,6452	1,2347	0,4840	70,6552	0,5	5
SHELL	368,9478	390,6071	10,1003	3,9593	5,8706	30	12
HEAP	598,5755	636,1563	1,7499	0,6860	6,2784	30	20
GAUSPIV	21,0798	21,811	0,3100	0,1215	3,4687	2	7

A Tabela 2 mostra em cada uma das linhas: o tempo médio do estabelecimento de um *checkpoint* para cada aplicação, utilizando o mecanismo de *checkpointing* tradicional; o desvio padrão (σx) para estes dados; o intervalo de confiança de 95% (IC); e o tamanho médio dos arquivos de *checkpoint* expressos em *Kbytes*. Os valores dos tempos apresentam variações, de uma aplicação para outra, em função do volume de informações salvas em cada *checkpoint*.

Tabela 2 – Resultado das medidas da atividade de cada *checkpointing*.

Aplicação	Tempo *ckpt*. (seg)	σx	IC 95%	Tam. *ckpt*. (*Kbytes*)
MAT	1,9900	1,2416	0,1539	2034,8994
TDC	4,8398	0,9546	0,1673	5352,1743
SHELL	1,6980	0,1480	0,01675	976,9092
HEAP	1,4548	0,06119	0,005363	976,8701
GAUSPIV	0,03504	0,04068	0,005638	13,8711

6. Conclusões

Neste artigo, foram abordados a motivação e os princípios básicos relacionados ao estabelecimento de *checkpoints* em aplicações orientadas a objetos. Em seguida, foram apresentadas as características e funcionalidades da biblioteca *Libcjp*, proposta para atividades de recuperação de objetos. Esta biblioteca não isenta o programador de conhecer os mecanismos básicos envolvidos na execução das atividades de recuperação - ele precisa configurar alguns parâmetros e introduzir algumas chamadas. A biblioteca simplifica o restante da programação por disponibilizar os mecanismos necessários. Os números mostrados através da avaliação preliminar têm por objetivo dar uma idéia da repercussão da biblioteca em aplicações com diferentes perfis. Os valores, entretanto, podem ser bem diferentes se os parâmetros de uso (freqüência de chamada, volume de dados, etc...) variarem através da parametrização adequada ou de alteração nas características da aplicação.

Serão realizados, em uma próxima fase, novos experimentos de *checkpointing* utilizando a biblioteca, e uma análise detalhada dos resultados de desempenho obtidos, visando observar tendências e obter conclusões adicionais.

Referências

[1] N. Singhal; N. Shivaratri. Advanced Concepts in Operation Systems Distributed, Database and Multiprocessor Operation Systems. New York: McGraw-Hill, 1994.

[2] J. S. Plank et al. Libckpt: Transparent Checkpointing under Unix. In: Usenix Technical Conference, New Orleans, 1995. Proceedings. [s.n], January 1995.

[3] M. Kasbekar; et al. Issues in the Design of a Reflective Library for Checkpointing C++ Objects. In: 18th IEEE Symposium on Reliable Distributed Systems, Lausanne, Oct. 1999.

[4] J. L. Lawall; G. Muller. Efficient Incremental Checkpointing of Java Programs. In: Intl. Conference on Dependable Systems and Networks (DSN 2000), New York, June 2000. pp. 61-70.

[5] M. O. Killijian; J. C. Fabre; J. C. Ruiz-Garcia. Using compile-time reflection for object checkpointing. LAAS, February 1999. (Technical Report-99049).

[6] J. Mathis. Persistence and Java Serialization. In: Java 1.1: Unleashed. Indianapolis: Sams.Net, 3ed., 1997.

[7] S. C. Bertagnolli. Integrando Aspectos de Distribuição e de Tolerância a Falhas no Ambiente Java Reflexivo (Jreflex). Porto Alegre: PPGC da UFRGS, Julho 2000 (TI-913).

[8] Sun Microsystems. Java Object Serialization Specification – JDK 1.2 Beta 3. Disponível em URL: ftp://ftp.javasoft.com/docs/jdk1.2/serial-spec-JDK1.2.ps. Sun Microsystems, February, 1998.

[9] K. Arnold; G. James. The Java Programming Language. [S.l]:Addison-Wesley, 2. ed.,1997.

[10]E. K. Hansen. Checkpointing Java Programs on Standard Java Implementations using Program Transformation. Master's Thesis. Department of Computer Science, University of Copenhagen. Nov. 1999. Disponível por www em http://hjem.get2net.dk/esben_krag_hansen/jcp/index.html.