UNIVERSIDADE CATÓLICA DE PELOTAS
CENTRO POLITÉCNICO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

SHARED-GM: Arquitetura de Memória Distribuída para o Ambiente D-GM.

por
Gustavo Mata Zechlinski

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

Orientadora: Prof. Dr. Renata Hax Sander Reiser

DM-2010/2-002

Pelotas, setembro de 2010

AGRADECIMENTOS

A Deus, por permitir que eu esteja aqui, me concedendo a vida e me dando saúde e força interior para que eu pudesse superar as dificuldades e alcançar esta grande conquista.

A minha esposa Flávia, que soube entender a minha necessidade de crescimento pessoal, sempre me incentivando e dando o suporte necessário para esta longa caminhada. Também compreendeu os momentos de angústia e sempre me apoiou no que foi necessário.

Aos meus filhos, que muitas vezes tiveram as suas diversões podadas, tendo que permanecer em casa devido as provas ou trabalhos necessários para minha conclusão do mestrado.

Aos meus pais, por me trazerem ao mundo e me criarem com educação, permitindo-me o desenvolvimento da inteligência necessária para a vida. Além de me incentivarem, também forneceram o suporte técnico para que eu pudesse cursar este mestrado.

A minha família em geral, pois sem vocês não teria chegado até aqui.

Ao professor Adenauer e especialmente à professora Renata, que me deram esta oportunidade e acreditaram no meu potencial, sem eles também não teria chegado até aqui.

Ao colega Anderson, por não medir esforços em me ajudar nas etapas de implementação e testes do protótipo.

Aos colegas e funcionários do mestrado que direta ou indiretamente colaboraram para a conclusão desta dissertação.

Obrigado a todos que estiveram comigo nesta jornada!

Não se deve ir atrás de objetivos fáceis.
É preciso buscar o que só pode ser alcançado
por meio dos maiores esforços.
— ALBERT EINSTEIN

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE LISTAGENS

LISTA DE ABREVIATURAS E SIGLAS

LAN	Local Area Network
WAN	Wide Area Network
GM	Geometric Machine
EXHEDA	Execution Environment for High Distributed Applications
D-GM	Distributed-Geometric Machine
VPE	Visual Programming Environment
VirD	Virtual Distributed
TCP/IP	Transmission Control Protocol/Internet Protocol
IP	Internet Protocol
DSM	Distributed Shared Memory
RAM	Random Access Memory
JVM	Java Virtual Machine
API	Application Program Interface
IPC	Inter Process Communication
RMI	Remote Method Invocation
RPC	Remote Procedure Call
DSO	Distributed Shared Objects
SSI	Single System Image
BSD	Berkeley Software Distribution
API	Application Program Interface
JMM	JAVA Memory Model
JPC	JavaParty Compiler
P2P	Peer-to-Peer
DSO	Distributed Shared Objects
HPC	High Performance Cluster

HA	High Availability
RAM	Random Access Memory
POJO	Plain Old JAVA Objects
JDK	JAVA Development Kit
JRE	JAVA Runtime Environment
JEE	JAVA Enterprise Edition
XML	Extensible Markup Language

RESUMO

O recente avanço das tecnologias de computadores impulsionaram o uso de *clusters* de computadores para execução de aplicações que exijam um grande esforço computacional, tornando esta prática uma forte tendência atual. Acompanhando esta tendência, o Ambiente D-GM (*Distributed-Geometric Machine*) constitui-se em uma ferramenta compreendendo dois módulos de *software*, VPE-GM (*Visual Programming Environment for Geometric Machine*) e VirD-GM (*Virtual Distributed Geometric Machine*), os quais objetivam o desenvolvimento de aplicações da computação científica aplicando a programação visual e a execução paralela e/ou distribuída, respectivamente.

O núcleo do Ambiente D-GM está fundamentado na Máquina Geométrica (*Geometric Machine*-GM), um modelo de máquina abstrato para computações paralelas e/ou concorrentes cujas definições abrangem os paralelismos existentes para execução de processos. A principal contribuição deste trabalho é a formalização e desenvolvimento de uma memória distribuída para o Ambiente D-GM através da concepção, modelagem e construção da integração entre o Ambiente D-GM e um sistema DSM (*Distributes Shared Memory*). Portanto, visando melhoria na dinâmica de execução com maior funcionalidade e, possivelmente, com melhor desempenho no ambiente D-GM. A esta integração, cujo objetivo é fornecer um modelo de memória compartilhada distribuída para o Ambiente D-GM, dá-se o nome de ShareD-GM. Com base no estudo de implementações em *software* de DSM e nas características que atendem aos requisitos de implementação da memória distribuída do Ambiente D-GM, este trabalho considera o uso do sistema *Terracotta*. Salientam-se duas facilidades apresentadas pelo *Terracota*: a portabilidade e a adaptabilidade para execução distribuída em *clusters* de computadores com pouca ou até nenhuma modificação no código (*codeless clustering*), as quais retornam grandes benefícios quando da integração com aplicações JAVA. Além disso, verifica-se o fato de que o *Terracotta* não utiliza RMI (*Remote Method Invocation*) para comunicação entre os objetos em um Ambiente JAVA. Neste perspectiva, procura-se minimizar o *overhead* dos dados produzidos pelas serializações (*marshalling*) nas transmissões via rede. Pôde-se também comprovar durante o desenvolvimento de testes de avaliação da implementação da arquitetura proporcionada pela integração ShareD-GM, que a execução de aplicações modeladas no Ambiente D-GM, como o algoritmo de *Smith-Waterman* e o método de Jacobi, apresentaram menor tempo de execução quando comparados com a implementação anterior, no módulo VirD-GM de execução do Ambiente D-GM.

Palavras-chave: Memória Distribuída, Ambiente D-GM, Implementações de DSM, Modelagem.

TITLE: "SHARED-GM: DISTRIBUTED MEMORY ARCHITECTURE FOR D-GM ENVIRONMENT"

ABSTRACT

The recent advances in computer technology have increased the use of computer clusters for running applications which require a large computational effort, making this practice a strong tendency. Following this tendency, the D-GM (Geometric Distributed-Machine) environment is a tool, composed by two software modules, VPE-GM (*Visual Programming Environment for Geometric Machine*) and VirD-GM (*Virtual Distributed Geometric Machine*), whose goals are the development of applications of the scientific computation applying visual programming and parallel and/or distributed execution, respectively.

The core of the D-GM environment is based on the Geometric Machine (GM Model), which is an abstract machine model for parallel and/or concurrent computations, whose definitions cover the existing parallels to process executions.

The main contribution of this work is the formalization and development of a distributed memory for the D-GM environment, designing, modeling and constructing the integration between such environment and a distributed shared memory (DSM) system. Therefore, it aims at obtaining a better execution dynamic with major functionality and possibly, an increase in performance in the D-GM execution applications.

This integration, whose objective is to supply a shared distributed memory module to the D-GM environment, is called ShareD-GM environment. Based on the study of DSM softwares implementations, mainly on their characteristics which meet all the requirements to implement the distributed memory of the D-GM environment, this work considers the use of *Terracotta* system.

This study highlights two facilities both present in *Terracota*: the portability and adaptability for distributed execution in a cluster of computers with no code modifications (codeless clustering).

Besides these characteristics, one can observe that *Terracotta* does not make use of RMI (Remote Method Invocation) for communication among objects in a JAVA environment. From this point of view, one may also minimize the overhead of data serializations (marshalling) in network transmissions. In addition, the development of applications to evaluate the implementation of the architecture model provided by the ShareD-GM integration, as the algorithm Smith-Waterman and the Jacobi method, showed a shorter running time when compared to the previous VirD-GM execution module.

Keywords: Distributed Shared Memory, D-GM Environment, DSM Implementations, Modeling.

1 INTRODUÇÃO

Nos dias de hoje, a computação em *cluster* tem tido a sua importância aumentada tanto no cenário das aplicações que necessitam maior poder de computação quanto no cenário das aplicações paralelas e distribuídas. Neste contexto, a pesquisa sobre ferramentas que se utilizam da computação em *cluster* para a execução de aplicações, como o Ambiente D-GM, tem aumentado significativamente. Por outro lado, ferramentas que viabilizem uma programação, comunicação e execução mais dinâmica de ambientes de execução de aplicações em um *cluster*, como os sistemas DSM (*Distributed Shared Memory*), também se tornam temas de pesquisa significativos.

A discussão de novas perspectivas para a área de pesquisas sobre sistemas DSM tem acontecido de forma gradativa, principalmente pela necessidade de execução de aplicações em um *cluster*, viabilizando um possível aumento de desempenho sem a necessidade de modificações no código das aplicações. Concomitantemente ao desenvolvimento de sistemas DSM, tem-se evidenciado o seu aperfeiçoamento no sentindo de minimizar problemas como o *overhead* de comunicação em rede provocado por tais sistemas. O *software Terracotta* foi desenvolvido com base nestas premissas, indicando soluções oportunas para os problemas básicos dos sistemas DSM e consistindo em uma alternativa viável para concepção, modelagem e desenvolvimento da memória distribuída do Ambiente D-GM.

1.1 Tema e Justificativa

Esta dissertação tem como tema principal a concepção, modelagem e desenvolvimento de uma arquitetura de memória distribuída para o Ambiente D-GM, visando uma melhor dinâmica e maior funcionalidade na execução das aplicações desenvolvidas sob os componentes de desenvolvimento e de execução do referido ambiente. Almeja-se também a obtenção de um desempenho mais favorável na execução de aplicações em relação a atual implementação.

Na sua atual estruturação, o Ambiente D-GM não dispõe da especificação de sua memória distribuída, necessitando usar outros subsistemas, principalmente do *middleware* EXEHDA(*Execution Environment for High Distributed Applications*) (YAMIN, 2004), para realizar esta função. Além disso, recursos como o envio do objeto que estrutura a memória disponível para aplicações do Ambiente D-GM para modificação nos nodos do *cluster* são também utilizados.

Considera-se portanto relevante na concepção, modelagem e desenvolvimento da arquitetura de memória distribuída o uso de implementações em *software* de sistemas de

memória compartilhada distribuída (DSM), avaliando as suas características e funcionalidades de acordo com as necessidades do Ambiente D-GM.

Atualmente, os sistemas DSM (ESKICIOGLU, 2004) disponibilizam a possibilidade de uma maior abstração na programação de aplicações destinadas à executar de forma distribuída em um *cluster*. Assim, algumas das vantagens propostas por sistemas DSM são: (*i*) otimização da comunicação inter-processos, gerenciada de forma automática pelo sistema DSM; (*ii*) maior abstração na programação, não necessitando explicitar no código a comunicação dos processos; (*iii*) possibilidade de aumento de desempenho, devido à troca de mensagens otimizada entre os processos; e (*iv*) capacidade de adquirir funções de *middleware*, gerenciando a execução das tarefas da aplicação nos nodos e realizando o balanceamento automático de carga.

Frequentemente, a literatura indica que os sistemas DSM podem produzir *overhead* de comunicação entre os nodos de um *cluster*, prejudicando a funcionalidade e o desempenho das aplicações que executam sob tais sistemas. O surgimento de novas propostas, como o *software Terracotta* (TERRACOTTA, 2008), cuja estrutura busca minimizar esse problema, justificando a sua possibilidade de utilização na concepção, modelagem e desenvolvimento da arquitetura de memória distribuída proposta para o Ambiente D-GM.

1.2 Contextualização

As tecnologias de computadores tiveram um crescimento sem igual em outras áreas desde a metade do século passado. Esta evolução fez com que o custo e o tamanho das máquinas diminuísse, incentivando desta forma a utilização de um maior número de máquinas para a realização de tarefas que exijam um maior poder computacional.

Conjuntamente com este crescimento tivemos o advento e a evolução das redes de computadores de alta velocidade denominadas:

- LANs (*Local Area Networks*), as quais permitem que milhares de máquinas, dentro de um prédio, estejam conectadas e possam transferir entre elas pequenas quantidades de dados em poucos microsegundos, enquanto que grandes quantidades de dados podem ser transferidos a taxas variando de 10 a 1000 milhões de bits por segundo ou;

- WANs (*Wide Area Networks*) permitindo que milhões de máquinas espalhadas pela terra estejam conectadas a velocidades variando de 64kbps a gigabits por segundo (TANENBAUM; STEEN, 2001).

Como resultado da evolução destas tecnologias, tem-se uma maior facilidade para se utilizar sistemas computacionais compostos de um grande número de computadores conectados por uma rede de alta velocidade, que chamamos de redes de computadores, sistemas distribuídos ou *clusters*. Conseqüentemente este fato impulsiona as pesquisas nas áreas de sistemas distribuídos, programação paralela e computação em *cluster*, visando aumento de desempenho no processamento de certas aplicações. Dessa forma, são classificados os sistemas de computação de acordo com o nível de acoplamento dos seus componentes em multiprocessados e multicomputadores.

Nos sistemas multiprocessados, tem-se um forte acoplamento entre os processadores e a memória utilizada pelos processos é uniforme, ou seja, os processos compartilham a mesma área de memória permitindo que os mesmos se comuniquem com

uma maior agilidade. Neste caso, a programação se torna mais simples, pois o programador não precisa se preocupar com a comunicação entre processos que é feita de forma genérica. Aspectos como sincronização, contenção e escalabilidade devem ser considerados nesse modelo.

Em um sistema de multicomputadores o acoplamento dos processadores é fraco pois temos vários computadores interconectados através de dispositivos de ligação, o que nos apresenta um cenário de memória distribuída com diferentes espaços de endereçamento, fisicamente distantes um do outro, onde a comunicação entre os processos ocorre através da troca de mensagens. E além disso, utilizam protocolos bem elaborados para garantir a qualidade dos dados transferidos.

Apesar de apresentar problemas de desempenho devido à comunicação entre os processos estar baseada na troca de mensagens, este sistema tem um grande poder computacional possibilitado pela sua escalabilidade.

Neste contexto, um sistema de memória compartilhada distribuída (DSM - *Distributed Shared Memory*) tenta aliar as vantagens de programação de um sistema multiprocessado a um sistema de multicomputadores, ocultando os problemas da comunicação em rede do programador ao mesmo tempo que permite a escalabilidade inerente dos sistemas multicomputadores. Além disso, uma interface de memória compartilhada procura ser mais desejável do ponto de vista do programador da aplicação, permitindo que ele se preocupe mais com o desenvolvimento da aplicação em si do que com o gerenciamento da comunicação entre processos (PARK, 2000).

Observa-se então, que, um sistema DSM pode apresentar vantagens para programação paralela e distribuída, justificando a sua possibilidade de uso em aplicações que utilizam a computação paralela e distribuída.

O Ambiente D-GM, possui aplicações que fazem uso de sistemas multicomputadores (*cluster*) para a execução de processos que necessitem de processamento paralelo e/ou distribuído, normalmente aplicações da computação científica.

A concepção, modelagem e desenvolvimento de uma arquitetura de memória distribuída para o Ambiente D-GM empregando como base um sistema DSM, além de preencher uma lacuna quanto a falta de especificação, também considera as possíveis vantagens já mencionadas anteriormente. O surgimento de novas propostas na área de sistemas DSM também oportuniza uma possível melhoria no desempenho de aplicações que executam sob estes sistemas, sinalizando que as aplicações modeladas e executadas no Ambiente D-GM podem vir a ter um melhor desempenho.

1.3 Motivações

Como motivações para o desenvolvimento deste trabalho, destacam-se:

(i) a importância de incentivar o estudo das implementações em *software* de DSM e as possíveis demandas visando a integração destas com ambientes de programação e execução de aplicações científicas;

(ii) a aproximação da área de pesquisa em DSM com a área de processamento paralelo e distribuído, objetivando melhorias na funcionalidade e desempenho no que diz respeito a execução de aplicações da computação científica;

(iii) a necessidade prover uma arquitetura de memória distribuída para o Ambiente D-GM, através da consolidação da integração entre uma implementação em *software* de DSM (*Terracotta*) e o módulo de execução de aplicações (VirD-GM) do Ambiente D-GM.

1.4 Objetivos

O objetivo geral deste trabalho é realizar a concepção, modelagem e desenvolvimento de uma arquitetura de memória distribuída para o Ambiente D-GM através da integração de uma implementação em *software* de DSM ao referido Ambiente, para tal deve-se:

- Considerar os requisitos pertinentes as demandas do Ambiente D-GM e referentes a definição da memória distribuída deste ambiente;
- Avaliar as implementações em *software* de DSM, considerando alguns pontos específicos com o intuito de propiciar a seleção de uma destas implementações que melhor se adapte ao *software* já desenvolvido no Ambiente D-GM.
- Modelar, desenvolver e implementar a integração entre uma implementação em *software* de DSM e o módulo de execução do Ambiente D-GM proporcionando uma arquitetura de memória distribuída para este ambiente.

Como objetivos específicos, podem-se destacar:

- Definição do contexto ao qual pertence o Ambiente D-GM, incluindo a computação em *cluster* e a especificação da linguagem JAVA;
- Revisão dos fundamentos de sistemas de memória compartilhada distribuída (DSM);
- Sistematização das características dos principais projetos que exploram DSM;
- Identificação dos requisitos do Ambiente D-GM, referentes à funcionalidade e ao desempenho de um sistema DSM;
- Definição de uma implementação em *software* de DSM para integração ao Ambiente D-GM, servindo como base para a modelagem da memória compartilhada distribuída deste Ambiente;
- Modelagem e implementação da arquitetura ShareD-GM, com suporte a heterogeneidade, baseada nos estudos anteriores; e
- Realização de testes com aplicações científicas programadas no Ambiente D-GM com o intuito avaliar a implementação da arquitetura ShareD-GM realizada, bem como comparar o desempenho de execução de tais aplicações no Ambiente D-GM com a arquitetura de memória distribuída e sem esta arquitetura.

1.5 Metodologia

As seguintes metodologias foram utilizadas durante a produção desta dissertação:

- Estudo da computação em cluster e da especificação da linguagem JAVA;
- Revisão bibliográfica das fundamentações do Ambiente D-GM, considerando o material já escrito e desenvolvido para este ambiente;
- Estudo de casos de implementações em *software* de memória compartilhada distribuída (DSM), bem como o estudo das fundamentações destes sistemas através da leitura de artigos e pesquisa na *Web*;
- Revisão das abordagens disponíveis para implementação de DSM, com especial atenção para as que usam a linguagem JAVA e para o *software Terracotta*;
- Modelagem e implementação de uma arquitetura de memória distribuída através da integração entre um ambiente usado para programação e execução de aplicações paralelizáveis e uma implementação em *software* de DSM; e
- Realização de testes funcionais e testes para medição de tempo de execução a partir da programação de aplicações científicas no Ambiente D-GM.

1.6 Organização do Texto

A apresentação deste texto está organizada em seis capítulos, brevemente resumidos a seguir.

A presente introdução, onde são apresentados o contexto onde este trabalho está inserido, a motivação e as tendências atuais de pesquisa, o tema e os objetivos, gerais e específicos, do trabalho, bem como a metodologia empregada e a organização do texto.

A descrição do Ambiente D-GM e dos componentes envolvidos, considerando suas principais características e tecnologias empregadas é apresentada no Capítulo 2.

No Capítulo 3, considera-se uma revisão bibliográfica dos fundamentos de sistemas de memória compartilhada distribuída, abordando a sua estrutura e os métodos para manipulação, organização e distribuição dos dados compartilhados neste tipo de sistema. São revistos conceitos fundamentais de consistência e coerência de cache, abordando os principais modelos existentes e estratégias utilizadas.

As implementações em *software* de DSM, categorizadas de acordo com a organização dos dados compartilhados empregada, são apresentadas no capítulo 4. Inicialmente, considera-se as abordagens para os algoritmos de implementação usados em sistemas DSM. No desenvolvimento, três implementações baseadas em páginas e cinco implementações baseadas em objetos são descritas de acordo com suas arquiteturas e funcionamento, buscando identificar o emprego dos fundamentos de memória compartilhada distribuída. Na seção 4.3.3, alguns conceitos relacionados ao modelo de memória, comunicação remota e mecanismos de sincronização do JAVA, são detalhados. Os principais problemas enfrentados pelas implementações baseadas em páginas e objetos são abordados conjuntamente à uma comparação de suas principais características na seção 4.4. A seção 4.5, faz uma avaliação em separado das implementações DSM para JAVA por serem as mais relevantes para o Ambiente D-GM. A seleção de suas principais

características e funcionalidades contribui para a escolha de uma implementação de DSM com o intuito de realizar a integração com o módulo de execução do Ambiente D-GM.

A grande aceitação desta linguagem para programação concorrente e distribuída está baseada em características como: (i) *multithreading*; (ii) independência de plataforma; e (iii) facilidade de programação para redes. O sistema *multithread* JAVA tem grande capacidade para o paralelismo de aplicações, através da execução de *threads* em processadores físicos diferentes em máquinas de memória compartilhada, tornando-se mais complexa a sua implantação em sistemas de memória distribuída. Atualmente, mesmo com esta complexidade, alguns sistemas de DSM tem sido desenvolvidos na linguagem JAVA, devido ao grande potencial para se obter alto desempenho em um sistema de memória distribuída usando o modelo *multithreaded* do JAVA, como se estivessem em um sistema de memória compartilhada.

O Capítulo 5 define os requisitos mais importantes de acordo com as demandas do Ambiente D-GM, plenamente atendidos pelo *software Terracotta*. A partir da definição do *Terracotta* como o sistema de memória compartilhada distribuída à ser integrado ao Ambiente D-GM, procede-se com um estudo mais aprofundado do mesmo, apresentando o seu funcionamento interno e explicitando as opções de configuração para a realização da referida integração. A realização de testes com o *software Terracotta* permite a aprendizagem de suas características e funcionalidades. Segue-se, na seção 5.2, a modelagem proposta nesta dissertação para posterior implementação da arquitetura de memória distribuída do Ambiente D-GM.

As aplicações científicas utilizadas para avaliação da arquitetura ShareD-GM são descritas no capítulo 6. Considera-se os algoritmos de *Smith-Waterman* usado principalmente para comparações de sequências de strings em bases de dados, mais especificamente utilizado para comparar cadeias de DNA desconhecidas com bases de dados de cadeias de DNA já conhecidas, em busca de similaridades para identificação de DNA e o algoritmo do método de Jacobi, um método iterativo para resolução de grandes e esparsos sistemas lineares de equações algébricas (SELAs). A partir da implementação destes algoritmos de acordo com as especificações do Ambiente D-GM, pôde-se realizar a execução dos mesmos e obter os resultados discutidos nas seções 6.1.2 e 6.2.2, fazendo uma comparação com a implementação anterior do módulo de execução do Ambiente D-GM.

Por fim, no Capítulo 7, destacam-se as considerações referentes ao tema pesquisado e as conclusão alcançadas com o desenvolvimento da dissertação. Apresenta-se também neste capítulo, as contribuições técnicas específicas e gerais realizadas com o término da dissertação e as sugestões de continuidade para futura complementação desta dissertação.

2 AMBIENTE D-GM

O Ambiente D-GM (*Distributed-Geometric Machine*) visa disponibilizar as abstrações do modelo GM (*Geometric Machine*) em uma plataforma com suporte ao desenvolvimento e execução distribuída e/ou concorrente de algoritmos da computação científica. Mais especificamente, o Ambiente D-GM gerencia um *software* de programação paralela e distribuída, possibilitando a criação e execução de aplicações baseadas em algoritmos paralelos da computação científica. Tais aplicações devem ser desenvolvidas segundo as determinações do modelo GM por meio de um módulo de *software* responsável pela programação.

2.1 Modelo GM

O modelo GM (*Geometric Machine*) (REISER, 2002) é um modelo de máquina abstrato utilizado para modelagem de computações paralelas e/ou concorrentes e não-determinísticas. Este modelo é constituído de uma máquina abstrata, com memória conceitualmente infinita e tempo de acesso constante e foi concebido para ter suporte à semântica dos algoritmos da computação científica. Esta máquina abstrata apresenta uma construção ordenada para processos e estados computacionais definida por uma estrutura matemática indutiva, com fundamentação na Teoria dos Domínios.

Neste contexto, os conceitos básicos como memória e processos são construídos a partir de um espaço geométrico onde a indexação da memória e dos processos ocorre pela construção geométrica que considera apenas a distribuição dos pontos no espaço geométrico, sem analisar suas propriedades métricas e topológicas. Dessa forma, o conjunto S de estados e o conjunto P de processos são, respectivamente, definidos pelos conjuntos de valores e de ações computacionais, ambos rotulados por elementos do conjunto I representando posições do espaço geométrico, mostrados na figura 2.1.

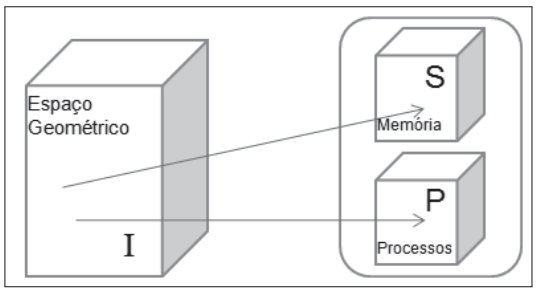

Figura 2.1: Modelo GM (Geometric Machine)

Assim, o modelo GM contempla dois tipos especiais de paralelismo:

- O paralelismo espacial onde os processos ocorrem em múltiplos recursos (processadores) em um mesmo instante de tempo, com memória e processos possivelmente infinitos, definidos por estruturas matriciais; e

- O paralelismo temporal onde se tem processos concorrentes, criados por um conjunto enumerável de modelos GM e utilizando posições distintas de uma memória global em instantes de tempo sincronizados.

Pode-se também afirmar que podem ser representados no modelo GM programas com estrutura lógica mais complexa, compatível com as novas arquiteturas e sistemas de organização de computadores, como o processamento paralelo síncrono e/ou distribuído (REISER; COSTA; DIMURO, 2003)

Para disciplinar a elaboração de programas paralelos e/ou concorrentes, o modelo GM dispõe de uma linguagem de programacão visual denominada VLGM (*Visual Language for the Geometric Machine*), cuja estruturação estimula o aprendizado de técnicas de programação paralela e concorrente baseada numa semântica bidimensional inerente as linguagens visuais, provendo uma representação espaço-temporal da estrutura da memória e dos processos representados no modelo GM (REISER et al., 2003).

A partir da especificação e definição da VLGM foi construído um ambiente de programação visual inicialmente chamado de APV-MG (Ambiente de Programação Visual para Máquina Geométrica) e atualmente denominado VPE-GM (*Visual Programming Environment for the Geometric Machine*) (PRESTES et al., 2005). O VPE-GM é um módulo de *software* integrante do Ambiente D-GM responsável por propiciar um ambiente de desenvolvimento através de programação visual para algoritmos paralelos da computação científica.

Nas próximas seções ocorre o detalhamento de duas estruturas básicas do modelo GM e da VLGM, o processo elementar e os construtores, necessárias à programação dos algoritmos, bem como do VPE-GM, exemplificando alguns tipos de construções visuais de processos e de construção de produtos.

2.1.1 Processo Elementar

De acordo com as abstrações do modelo GM, a estrutura do processo elementar, instituído por (FONSECA, 2008), estabelece a unidade computacional básica do modelo GM, sendo caracterizado por alterar uma única posição de memória após o término de sua execução. A partir do processo elementar, operações computacionais como soma, subtração e multiplicação podem ser realizadas. Operações mais complexas também podem ser definidas desde que estas funções estejam bem definidas na biblioteca de funções VirD-Lib descrita na seção 2.2.1.3, ou seja, deve existir na biblioteca uma entrada correspondente com o mesmo nome da operação definida pela aplicação desenvolvida no VPE-GM.

Na figura 2.2 tem-se a representação visual de um processo elementar. Neste processo elementar, sum^0, tem-se o identificador *sum* indicando que este processo é uma soma e o número 0 indicando a posição de memória onde o resultado será armazenado. Na modelagem, devem ser especificados os parâmetros da operação, os quais podem ser posições de memória ou dados compatíveis com os tipos definidos pela operação *sum*.

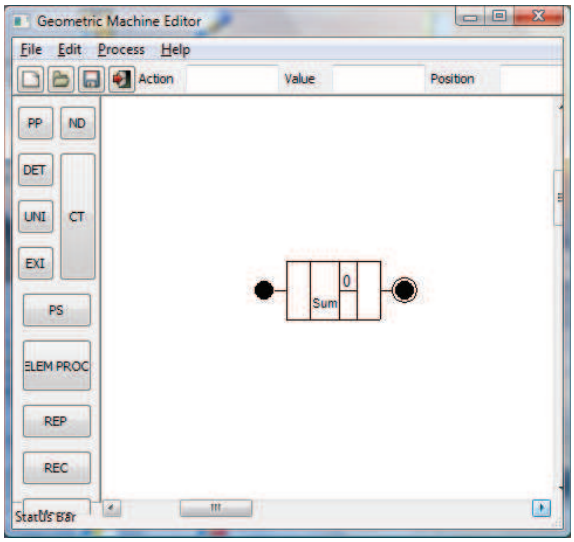

Figura 2.2: Processos Elementares e Exemplos de Construtores

2.1.2 Construtores

Os Construtores determinam como a composição de dois ou mais processos elementares será executada, se de forma produto sequencial ou em paralelo, iterativa sequencial ou paralela ou soma determinística ou não-determinística. Os construtores possíveis para o modelo GM, são:

- Construtor Produto Sequencial: Determina que os processos serão executados de acordo com a ordem em que eles foram construídos no editor. Neste caso, enfatiza-se a dependência de dados, onde um processo depende dos resultados do processo anterior;

- Construtor Produto Paralelo: Neste construtor a ordem de execução dos processos é irrelevante e a execução de dois ou mais processos é feita de forma simultânea desde que não haja dependência de dados entre estes processos;

- Construtor Soma Não-determinística: Este construtor tem como principal característica a aleatoriedade da escolha do processo a ser executado, dentre os que compõem a operação de soma não-determinística. A soma não-determinística somente pode ser aplicada sobre dois ou mais processos mutuamente exclusivos, ou seja, somente um destes processos poderá alterar a posição de memória de cada vez. O índice que representa a posição de memória deve ser o mesmo para os dois ou mais possíveis processos participantes dessa operação;

- Construtor Iterativo Sequencial: Estabelece a inserção de dependências entre sucessivas iterações, ou seja, cada iteração n depende da correspondente execução da iteração $n - 1$. Considera na sua execução a verificação de dependências externas, identificando quais processos precisam ser finalizados para início da corrente iteração, e de dependências internas, referente a ordenação das computações;

- Construtor Iterativo Paralelo: Caracteriza-se pela não dependência de dados entre as iterações, referente a processos concorrentes, ou seja, cada iteração n pode ser

executada em paralelo com qualquer outra iteração do processo, alterando somente a posição de memória onde os dados serão armazenados;

- Construtor Soma Determinística: Responsável pela definição de estruturas de controle, viabiliza a definição de desvios condicionais no fluxo de execução de uma aplicação no modelo GM. Este construtor é definido por três parâmetros: um operador *Teste* e dois operadores *Processos*. Enquanto a semântica de um processo no modelo GM é dada por uma transição de estados de memória, um *Teste* consiste em um operador que recebe um estado de memória e retorna um valor booleano de uma proposição (condicional). Assim, para correta execução deste construtor, faz-se necessário a verificação da posição de memória associada a condicional que define o operador *Teste*;

Os construtores mais frequentemente usados são os produtos sequenciais ou paralelos e iterativos sequenciais ou paralelos.

2.1.3 Módulo de Programação Visual VPE-GM

O VPE-GM é um ambiente visual para modelagem de computações paralelas que foi construído visando melhorias no acesso, manipulação e compreensão da programação paralela e/ou distribuída concebidas segundo as abstrações do modelo GM. Neste ambiente, as tradicionais técnicas das linguagens de programação textual são combinadas com a construção de um ambiente gráfico e visual. Os componentes de *software* que compõem o VPE-GM são: (i) o editor de processos, mostrado na figura 2.3 onde são modelados os processos paralelos e/ou concorrentes; (ii) o editor de memória, apresentado na figura 2.4, onde ocorre a configuração de memória e instanciação dos estados de acordo com a aplicação modelada.

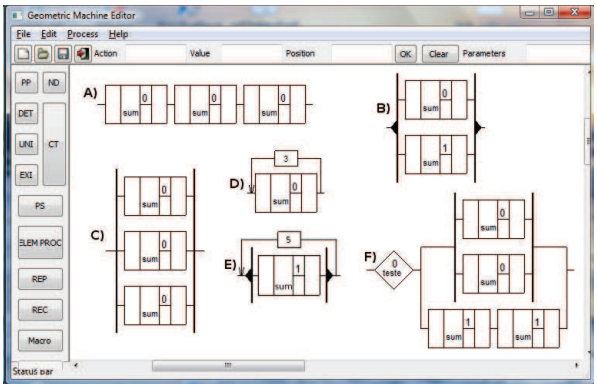

Figura 2.3: Editor de Processos

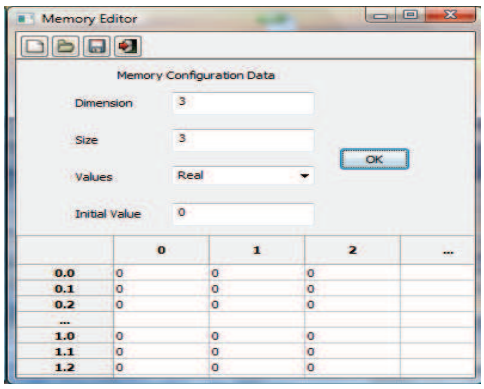

Figura 2.4: Editor de Memória

2.1.3.1 Editor de Processos

O editor de processos do VPE-GM é onde ocorre a representação gráfica e a manipulação dos processos da aplicação. A descrição dos processos graficamente construídos no VPE-GM, bem como a configuração destes processos são exportadas para um arquivo XML, sendo este o arquivo descritor de processos para execução que deve ser enviado ao módulo VirD-GM. O editor de processos tem como principais objetos gráficos,

o processo elementar e os construtores, a partir dos quais pode-se construir as aplicações. Como principais construtores destacam-se:

- Produto sequencial, figura 2.3 (A); e
- Produto paralelo, figura 2.3 (B).

Além destes construtores principais, o editor de processos ainda representa graficamente outros construtores, tais como:

- Soma não-determinística, para modelagem de processos randômicos, figura 2.3 (C);
- Processos iterativos, nas versões sequencial e paralela, figuras 2.3 (D) e 2.3 (E) respectivamente;
- Soma determinística, para modelagem dos processos condicionais, figura 2.3 (F);
- Construtor de macros, possibilitando o agrupamento de processos e construtores em um único elemento visual e com o uso de envelopes;

E construtores direcionados a computação quântica e recentemente implementados no editor como:

- Construtor de projeções;
- Construtor de medidas.

Estes construtores, além de serem usados para implementação de algoritmos básicos, podem ser usados também para modelagens de aplicações mais complexas como aplicações da computação quântica. Além destes dois tipos de estrutura, construtores e processo elementar, existe ainda o objeto teste elementar, que estrutura dois testes computacionais:

- Teste determinístico, nos casos de computações padrões e determinísticas; e
- Testes universais e existenciais, nos casos de computações não-determinísticas.

2.1.3.2 Editor de Memória

O editor de memória do VPE-GM permite a configuração do estado inicial da memória da aplicação através de uma representação gráfica, proporcionando dessa forma:

- a escolha da estrutura dimensional da memória, unidimensional ou bidimensional;
- a inicialização de valores nas posições de memória; e
- a definição do tipo de dados dos valores informados.

A partir dessa configuração o editor de memória do VPE-GM traduz as opções escolhidas e os valores informados em descrições compatíveis com a linguagem XML, gerando o arquivo XML descritor da memória da aplicação.

2.2 D-GM: Versão Distribuída do Modelo GM

A versão distribuída do modelo GM explora o paralelismo temporal, considerando um conjunto enumerável de máquinas com memória global compartilhada e sincronizadas no tempo. A implementação da versão distribuída do modelo GM está sendo proporcionada pelo Projeto D-GM (*Distributed-Geometric Machine*, e agora, Ambiente D-GM, tendo por principal objetivo promover uma execução efetivamente distribuída e paralela para as abstrações do modelo GM. (FONSECA, 2008).

O desenvolvido do Ambiente D-GM (*Distributed-Geometric Machine*) conta atualmente com dois módulos de *software*, o VPE-GM (*Visual Programming Environment for the Geometric Machine*) detalhado na seção 2.1.3 anterior e responsável por fornecer um ambiente de programação visual para o desenvolvimento de algoritmos paralelos da computação científica; e o VirD-GM (*Virtual Distributed Geometric Machine*) (FONSECA, 2008), gerente da execução paralela e/ou distribuída das computações paralelas definidas de acordo com as abstrações do modelo GM.

O estágio atual de desenvolvimento do Ambiente D-GM permitiu a validação das abstrações do modelo GM através da criação de aplicações tais como *Picalc* (cálculo do π pelo método de Monte Carlo) e *Passbreak* (quebra de senhas)(FONSECA, 2008).

Na figura 2.5 tem-se a visão funcional do Ambiente D-GM, onde pode-se visualizar como ocorre a ligação entre os módulos do ambiente. Nas próximas seções, o módulo VirD-GM será apresentado para um melhor entendimento do funcionamento desta versão distribuída do modelo GM.

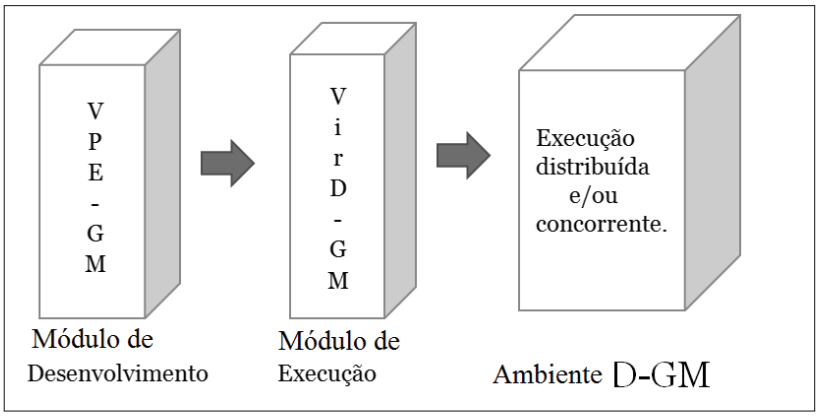

Figura 2.5: Ambiente D-GM: Visão Funcional

2.2.1 Módulo VirD-GM

O módulo VirD-GM foi inicialmente implementado objetivando o uso do ambiente de execução distribuída denominado EXHEDA (YAMIN, 2004), o qual estabelece um *middleware* direcionado a dar suporte à computação das aplicações distribuídas e/ou paralelas, para execução de processos graficamente modelados no editor do módulo de programação visual VPE-GM (FONSECA, 2008).

Dessa forma, o VirD-GM estende a organização do EXHEDA para o Ambiente D-GM, provendo os subsistemas básicos para a execução paralela e/ou concorrente das aplicações modeladas no VPE-GM. Compõem o VirD-GM três abstrações decorrentes da organização do EXEHDA:

- VirD-cel: constitui a área de atuação de uma VirD-base e dos seus VirD-nodos. É no âmbito de uma VirD-Cel que acontecem as computações distribuídas e/ou paralelas concebidas seguido o modelo D-GM;

- VirD-base: equipamento responsável pela gerência da arquitetura da D-GM como um todo ou ainda a infra-estrutura para operação dos VirD- nodos;

- VirD-nodo: são os equipamentos de processamento disponíveis, sendo responsáveis pela execução das computacões da D-GM.

Além disso, três componentes de *software* completam a arquitetura do VirD-GM, o *VirD-Loader*, o *VirD-Launcher* e o *VirD-Exec*, mostrados no diagrama da figura 2.6. Estes componentes são responsáveis pelo gerenciamento da execução distribuída e/ou paralela.

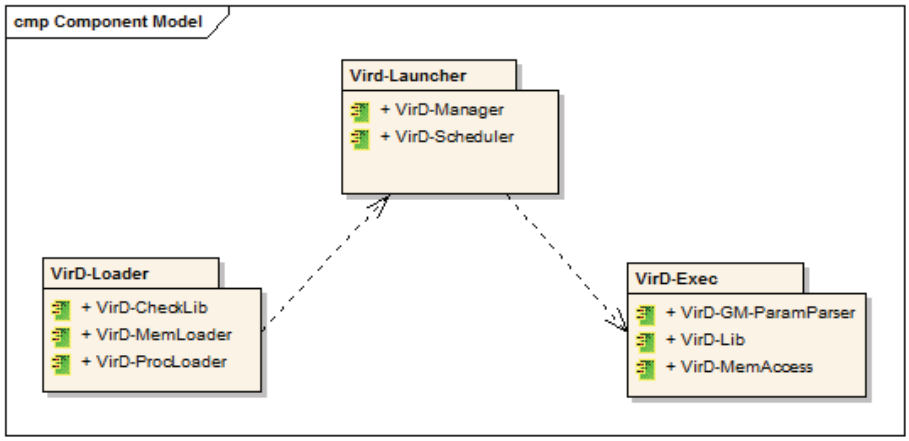

Figura 2.6: Diagrama de Componentes

2.2.1.1 VirD-Loader

Este componente de *software* é responsável por receber os arquivos descritores dos processos e da memória da aplicação exportados pelo VPE-GM, os quais contém também os parâmetros de execução. Integram este componente três funções:

- A *VirD-ProcLoader*, responsável pelo mapeamento do arquivo descritor em XML para estruturas de dados internas do ambiente de execução;

- A *VirD-MemLoader*, cuja incumbência é instanciar e inicializar a memória global compartilhada. A memória é atualizada com os valores obtidos pelo mapeamento do arquivo descritor XML da memória modelada no VPE-GM; e

- A *VirD-CheckLib*, confere no repositórios de funções a disponibilidade de bibliotecas auxiliares requeridas pelas computações descritas no arquivo descritor dos processos.

Na sequência, o *VirD-Loader* efetua as leituras, cria a matriz de adjacências para controle do fluxo de execução e prepara a arquitetura do VirD-GM para o disparo da aplicação através do VirD-Launcher.

2.2.1.2 VirD-Launcher

O componente *VirD-Launcher* tem como principal objetivo iniciar a execução da aplicação, a partir das informações recebidas do componente *VirD-Loader*, e gerenciar o seu processamento através de uma lista de processos em execução, de políticas de escalonamento e considerando a matriz de adjacências, criada pelo *VirD-Loader*, que controla os acessos conflitantes de memória.

Para a realização destas tarefas, duas funções principais deste componente devem entrar em funcionamento, são elas:

- A *VirD-Manager*, que gerencia o controle da execução a partir dos dados obtidos na matriz da adjacências e insere por meio da função VirD-TaskManager os processos da aplicação na lista de execução; e
- A *VirD-Scheduler*, cujo objetivo é realizar o mapeamento físico dos processos inclusos na lista de execução nos nodos disponíveis através de políticas de escalonamento previamente definidas no momento da configuração dos parâmetros operacionais do módulo de execução.

Após as configurações iniciais feitas pelos módulos *VirD-Loader* e *VirD-Launcher*, a aplicação modelada no módulo de programação visual do Ambiente D-GM tem a sua execução propriamente dita iniciada pelo componente *VirD-Exec*.

2.2.1.3 VirD-Exec

O *VirD-Exec* executa os processos concorrentes e/ou paralelos da aplicação nos VirD-Nodos disponíveis. Para tal, faz-se necessário a utilização de serviços do *middleware* EXHEDA e também o uso de operações de acesso a memória compartilhada.

Neste componente, três funções são necessárias para a execução dos processos:

- A *VirD-GM-ParamParser* com a tarefa de analisar os parâmetros de entrada e saída dos processos e considerar a sintaxe associada a cada processo associado as apliações criadas no VPE-GM;
- A *VirD-MemAccess*, que proporciona o acesso a memória global e a realização de operações de leitura e escrita em memória nas posições definidas pelos parâmetros de entra e saída dos processos; e
- A *VirD-Lib*, responsável por proporcionar o acesso ao repositório de funções pertencentes aos processos da aplicação.

2.2.2 Estrutura de Memória do VirD-GM

Faz-se necessário também neste trabalho o estudo da estrutura de memória atualmente utilizada no VirD-GM com o intuito de integrar esta estrutura com o sistema de memória compartilhada distribuída proposto e modelar a memória distribuída do Ambiente D-GM.

A estrutura de memória padrão disponibilizada no VPE-GM (PRESTES et al., 2005) permitia instanciar valores de memória de um mesmo tipo para cada aplicação, isto é, todas posições de memória armazenam valores inteiros, por exemplo. Esta característica impedia modelagens mais complexas que necessitassem manipular valores de memória de vários tipos como, *booleans* e *strings*, na mesma aplicação.

Em (MUNHOZ, 2009), essa estrutura de memória foi modificada de modo a permitir a instanciação de valores de memória de múltiplos tipos e na mesma aplicação.

Assim, o arquivo de memória principal definido como `VirD-MemGlobal` pode conter dados de múltiplos tipos em subclasses (`VirD-Integer`, `VirD-Float`, `VirD-String`, `VirD-Boolean`) e também ponteiros para arquivos de memória secundária definidos como `VirD-MemLocal` que poderá armazenar estruturas do tipo `VirD-Array`.

O `VirD-Array` permite estruturas de dados de múltiplas dimensões como matrizes, mas com dados apenas de um só tipo, como na definição padrão da estrutura de memória do Ambiente D-GM. A figura 2.7 apresenta um exemplo de estrutura de memória (`VirD-MemGlobal`) com múltiplos tipos incluindo memória secundária (`VirD-MemLocal`).

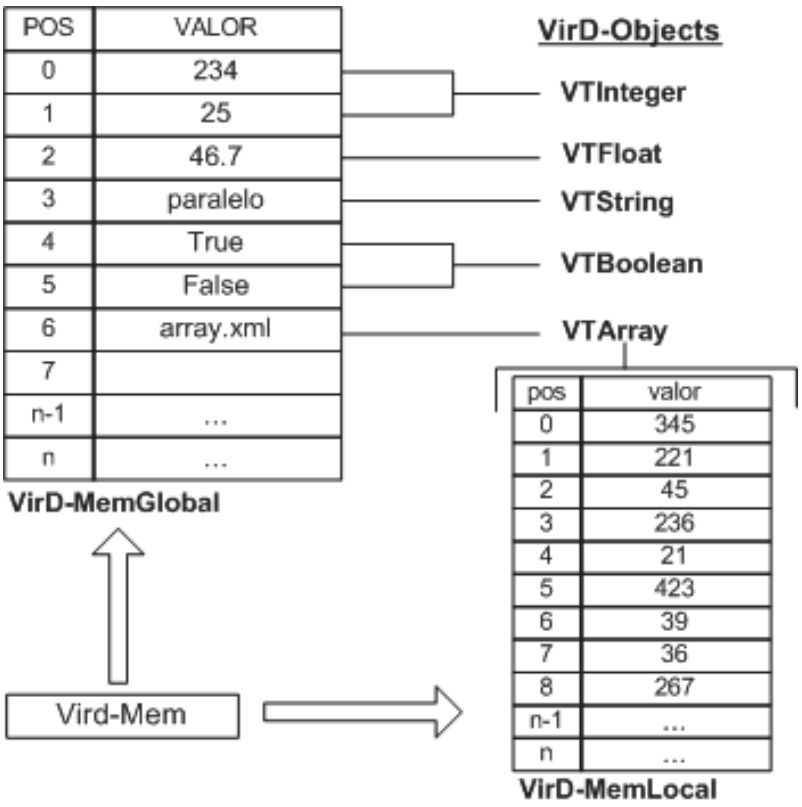

Figura 2.7: Arquitetura de memória do VirD-GM (MUNHOZ, 2009)

Permanecem inalterados os acessos à memória no desenvolvimento de aplicações, onde cada processo elementar pode alterar apenas uma posição de memória.

Entende-se por processo elementar uma operação aritmética como soma, subtração, divisão ou multiplicação. Sendo possível agora a utilização de matrizes e vetores além de múltiplos tipos de dados na mesma aplicação.

Para a definição destes novos tipos de dados, foram criados tipos de dados internos ao ambiente de execução VirD-GM, classificados da seguinte forma:

- Tipos de dados unidimensionais:
  - Tipos de dados elementares: `VirD-Object`, `VirD-Integer`, `VirD-Long`, `VirD-Float`, `VirD-Double`, `VirD-String` e `VirD-Boolean`;
  - Tipos de dados agregados: `VirD-Point` e `VirD-File`;
- Tipos de dados multidimensionais, denominados `VirD-Array` e `VirD-List`, os quais agregam atributos de dados unidimensionais.

A figura 2.8 apresenta o diagrama de classes modelando a estrutura de associação entre os tipos de dados multidimensionais como o `VirD-Array` e os tipos de dados unidimensionais.

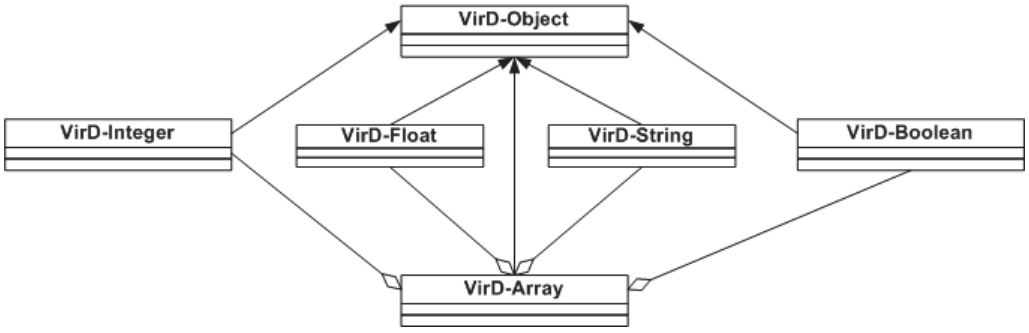

Figura 2.8: Estrutura dos tipos de dados (Diagrama de Classes) (MUNHOZ, 2009)

Neste caso, um objeto do tipo *array* poderá conter vários dados dos tipos *integer*, *float*, *boolean* e *strings*, combinados ou não, permitindo a definição de estruturas mais robustas como matrizes e vetores.

2.2.3 Acesso a Memória no Ambiente D-GM

No Ambiente D-GM, o acesso à memória é realizado conforme a política determinada pelo modelo GM, onde cada processo elementar pode escrever em apenas uma posição de memória e vários processos em paralelo podem alterar várias posições diferentes de memória ao mesmo tempo.

A leitura das posições de memória pode ser feita de diversas posições de memória por apenas um processo elementar. Pode-se considerar, como processo elementar operações básicas de soma, subtração, multiplicação e divisão.

2.2.4 Integração dos Módulos - Ambiente D-GM

A figura 2.9 mostra a integração dos módulos descritos anteriormente, formando o Ambiente D-GM. Conjuntamente com o diagrama de componentes é apresentado o fluxo de uma computação no momento da execução de uma aplicação desenvolvida no VPE-GM.

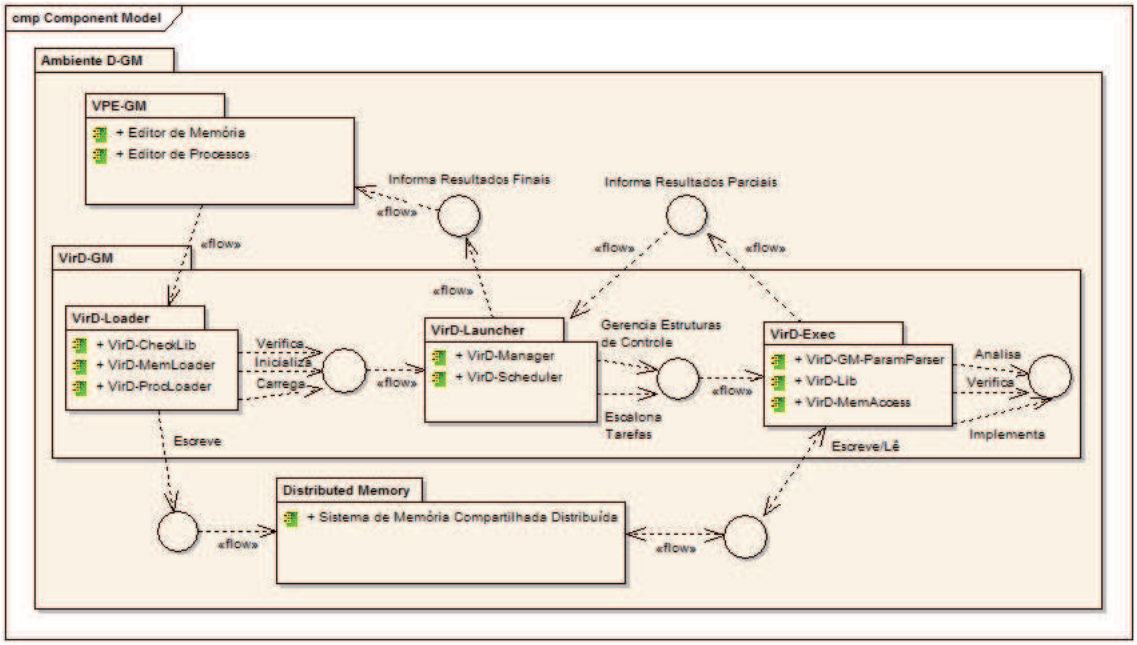

Figura 2.9: Integração dos Módulos

O único módulo ainda não descrito, chamado *distributed memory*, será o foco deste trabalho. O referido módulo será concebido, modelado e desenvolvido no decorrer do trabalho a partir da utilização de um sistema de memória compartilhada distribuída existente.

No próximo capítulo as definições básicas de sistemas de memória compartilhada distribuída serão apresentadas com o objetivo de promover o entendimento sobre as características deste tipo de sistema e propiciar as fundamentações para a concepção da modelagem e desenvolvimento da integração entre um sistema de memória compartilhada distribuída e o Ambiente D-GM.

3 DSM - PRINCIPAIS CONCEITOS

Atualmente, os tipos de memória existentes para sistemas com múltiplos computadores ou processadores permitem a seguinte classificação: (i) memória compartilhada, onde dois ou mais processadores utilizam uma memória em comum; (ii) memória distribuída, onde cada computador tem a sua memória e a comunicação entre eles ocorre por troca de mensagens através da rede; e (iii) memória compartilhada distribuída ou DSM, onde cada computador separado fisicamente visualiza a memória global do sistema como se estivessem em um sistema de memória compartilhada.

Mais especificamente, um sistema DSM (*Distributed Shared Memory*), consiste em uma memória abstrata utilizada por vários processadores em um sistema distribuído, no qual se permite aos processos compartilharem memória mesmo que estejam executando em nodos que não compartilham memória fisicamente (LI; HUDAK, 1989).

Apresentando as vantagens de programação de um sistema multiprocessador aplicadas à um sistema multicomputador, um sistema DSM busca facilitar a programação de aplicações destinadas a executar em mais de um computador, ocultando os problemas de comunicação em rede e permitindo a escalabilidade dos sistemas multicomputadores.

Um sistema DSM também propicia uma maior transparência e um maior grau de granularidade fina de comunicação do que troca de mensagens e RPCs(*Remote Procedure Calls*) (BIRRELL; NELSON, 1984). Entendendo-se por granularidade fina como um grande número de pequenas tarefas que podem ser executadas sem que haja necessidade de sincronização e comunicação. A figura 3.1 mostra a estrutura de um sistema DSM.

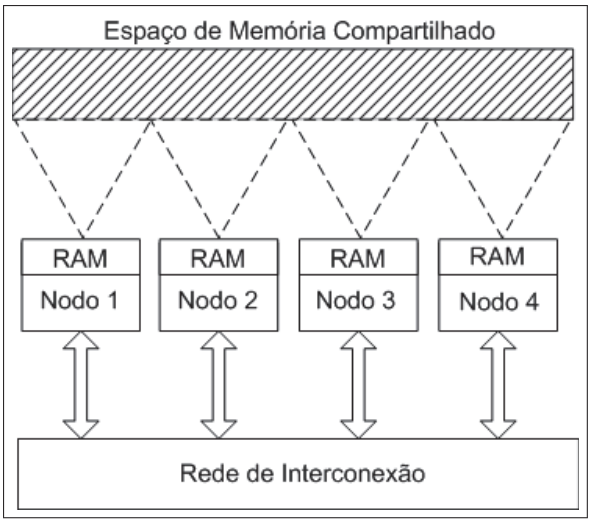

Figura 3.1: Estrutura Geral de um Sistema DSM

Esta estrutura genérica, exemplifica um sistema DSM constituído de quatro distintos e independentes processadores conectados por uma rede, cada um com memória RAM (*Random Access Memory*) própria e usufruindo de um espaço comum de memória compartilhada composto por dados (variáveis) compartilhados.

As implementações de DSM existem tanto em *hardware* quanto em *software*, sendo que as implementações em *software*, tanto as baseadas em páginas: (i) IVY; (ii) Munin; e (iii) TreadMarks, quanto as baseadas em objetos: (i) Linda; e (ii) Orca, bem como as implementações desenvolvidas na linguagem JAVA: (i) *Terracotta*; (ii) *JESSICA*; e (iii) *JavaParty*, são objetos de estudo neste trabalho.

Neste contexto, pode-se também afirmar que os processos executando sob um sistema de memória compartilhada distribuída realizam operações de leitura e escrita em dados compartilhados disponíveis em um depósito de dados (TANENBAUM; STEEN, 2007), assim como em sistemas de arquivos distribuídos e bancos de dados distribuídos. Este depósito de dados distribuído fisicamente por várias máquinas, inclui todas as cópias locais do depósito de dados disponíveis para cada processo de cada máquina integrante do sistema. Cada operação de escrita em qualquer uma das cópias locais deve ser propagada para as outras cópias para que as mesmas estejam sempre atualizadas com o valor da última operação de escrita durante um operação de leitura. As operações de leitura esperam sempre obter os dados mais atualizado, e se o sistema é consistente é isto que deve acontecer.

A seção 3.1 descreverá as possíveis organizações dos dados compartilhados em um sistema DSM, bem como pode ser feita a manutenção da consistência e da coerência dos dados compartilhados neste tipo de sistema.

3.1 Organização dos Dados Compartilhados

Em um sistema DSM, os dados compartilhados representam o espaço de endereçamento compartilhado e podem ser organizados de diferentes maneiras. Serão abordadas aqui as duas maneiras mais comumente usadas: (i) organização em páginas de memória; e (ii) organização em objetos.

A organização em páginas de memória é uma das mais simples organizações de dados existentes. Neste sistema o espaço de endereçamento é dividido em páginas, com cada página presente na memória de cada máquina. Estas páginas armazenam as variáveis compartilhadas pelo sistema.

Na organização em objetos, os dados são encapsulados em estruturas chamadas de objetos ao invés de compartilhar variáveis. Os objetos apresentam além dos dados, métodos que atuam sobre os dados e permitem aos processos a manipulação dos dados através de sua invocação. O acesso direto aos dados não é permitido.

Em sistemas DSM, usando dados organizados em páginas ou dados organizados em objetos, é necessário a utilização de estratégias para que estes dados fiquem disponíveis para as máquinas que compõe o sistema. A estratégia mais comum à organização de dados em páginas é a estratégia de migração e, para a organização de dados em objetos, a estratégia mais usual é a replicação.

3.2 Estratégias de Distribuição dos Dados Compartilhados em Sistemas DSM

Duas estratégias geralmente usadas para distribuir os dados compartilhados em um conjunto de computadores interconectados sob um sistema de memória compartilhada distribuída são replicação e migração. A replicação realiza múltiplas cópias de um mesmo dado em diferentes memórias locais. É utilizado, principalmente, para aumentar a confiabilidade e melhorar o desempenho do sistema, além de permitir acessos simultâneos por diferentes processadores aos mesmos dados compartilhados. A migração implica que somente uma cópia dos dados compartilhados existe em um certo momento, então os dados devem ser movidos quando um processador os requer para acesso exclusivo.

A estratégia de replicação é a mais utilizada por sistemas DSM, pois segundo (STUMM; ZHOU, 1998), esta proposta alcança melhor desempenho para uma grande quantidade de aplicações.

A replicação de dados ajuda a aumentar a confiabilidade do sistema mantendo cópias locais (*backups*) dos dados compartilhados em cada nodo do sistema e melhora o desempenho através do acesso local aos dados compartilhados. Apesar destas grandes vantagens, a replicação introduz também um grande problema: Como manter as cópias replicadas consistentes? Quando uma réplica é atualizada, ela se torna diferente das outras e esta atualização precisa ser propagada de maneira que eventuais inconsistências não sejam percebidas. Com o intuito de solucionar os problemas relacionados a consistência, os modelos de consistência estabelecem um contrato entre os processos e os dados compartilhados onde fica firmado que se os processos concordarem em obedecer as regras estabelecidas, o conjunto de dados compartilhados se compromete a funcionar de forma correta (TANENBAUM; STEEN, 2007). Na seção 3.3 os modelos de consistência mais usados em sistemas DSM são detalhados.

3.3 Consistência dos Dados Compartilhados em Sistemas DSM

Consistência dos dados compartilhados é a política que determina como e quando mudanças feitas por um processador são vistas pelos outros processadores do sistema. A escolha do modelo de consistência define o comportamento pretendido do mecanismo de DSM, com respeito às operações de leitura e escrita.

O modelo mais intuitivo de consistência de memória é a consistência rígida ou estrita, na qual uma operação de leitura retorna o valor de escrita mais recente. Este tipo de consistência é alcançado somente quando existe uma noção global de tempo que possa fornecer uma ordem determinística para todas as leituras e escritas. Entretanto, a expressão “escrita mais recente” é um conceito ambíguo em sistemas distribuídos, onde não existe um relógio global. Por este motivo, e também para melhorar o desempenho, foram desenvolvidos vários modelos para manter a consistência em sistemas DSM (LO, 1994), os quais serão abordados nas seções 3.3.3 e 3.3.4.

3.3.1 Notação

Para o correto entendimento dos exemplos, esta seção descreve a notação utilizada para representar os exemplos de modelos de consistência de memória. Nesta notação,

utiliza-se uma linha para cada processador no sistema e o tempo evolui da esquerda para a direita. Cada operação executada na memória compartilhada aparece na linha do processador e as cinco principais operações são *Write*, *Read*, *Sinchronization*, *Acquire* e *Release* que são representadas como:

- `W(var)valor`, que significa armazenar valor na variável compartilhada `var`;
- `R(var)valor`, que significa realizar a leitura da variável compartilhada `var`, obtendo assim valor;
- `S`, que significa a sincronização dos dados no sistema;
- `Acq(L)` ou `Acq(Lx)` que significa obter um *lock* para acessar uma região de dados compartilhados ou um *lock* associado a uma variável;
- `Rel(L)` ou `Rel(Lx)` que significa liberar um *lock* de uma região de dados compartilhados ou liberar um *lock* associado a uma variável;

Então, por exemplo, a operação "`W(x)1`" significa armazenar o "`1`" na variável "`x`" e a operação "`R(x)2`" significa realizar a leitura da variável "`x`" e obter o valor "`2`", `S` significa que é o momento da sincronização e os dados são atualizados, `Acq(L)` ou `Acq(Lx)` significa travar a variável "`x`" ou a região de memória para acesso por outras operações e `Rel(L)` ou `Rel(Lx)` significa liberar para acesso para outras operações a variável "`x`" ou a região de memória compartilhada.

3.3.2 Mecanismos de Exclusão Mútua

Para a compreensão das seções seguintes, faz-se necessário comentar aqui, as definições de *locks* e *monitors*.

Lock: é uma estrutura fornecida pela linguagem de programação para prover à exclusão mútua no acesso às variáveis compartilhadas. As duas maneiras de se trabalhar com *locks* são:

- *Acquire* (obtenção): espera até o *lock* estar livre e o adquire, obtendo acesso exclusivo a variável ou área de memória controlada por este *lock*;
- *Release* (liberação): libera a variável ou área de memória que estava com seu *lock* adquirido por um comando *Acquire* (*unlock*).

As regras para se usar *locks* são: sempre realizar um *acquire* (obtenção) antes de acessar uma estrutura de dados compartilhados e ao finalizar o acesso sempre realizar um release (liberação).

Monitors: Podem ser um *lock* ou zero ou mais variáveis de condição para o gerenciamento de acesso concorrente a dados compartilhados.

3.3.3 Modelos de Consistência Não-Sincronizada (Ordenados)

Os modelos de consistência que não possuem um mecanismo de sincronização explícito são chamados de modelos ordenados, ou seja, as operações em memória consideram apenas uma determinada ordem.

Destacam-se os seguintes modelos não sincronizados: (i) modelo de consistência seqüencial; (ii) modelo de consistência estrita; (iii) modelo de consistência causal.

Na sequência , descreve-se as principais caracterizações referentes as estes modelos.

3.3.3.1 Modelo de Consistência Seqüencial

A consistência sequêncial (*Sequencial Consistency*), assim como definido por (LAMPORT, 1979), diz que um sistema é sequencialmente consistente se o resultado de qualquer execução independe da ordem sequêncial em que as operações de todos os processadores são executadas, mas garante que as operações de cada processador individualmente apareçam na ordem especificada pelo seu programa. Ou seja, cada processador ou nodo do sistema enxerga as operações de escrita na memória na mesma ordem, embora esta ordem seja diferente da ordem definida pelo tempo real de execução das operações.

Na prática este é o modelo mais utilizado, embora o modelo de consistência estrita seja mais forte (ADVE; GHARACHORLOO, 1996) (TANENBAUM, 1995). Um modelo é dito mais forte que outro se cada condição exigida pelo modelo mais fraco é também exigida pelo modelo mais forte. Assim, um modelo de consistência mais forte tem um comportamento mais rígido que o modelo mais fraco (STEINKE; NUTT, 2004).

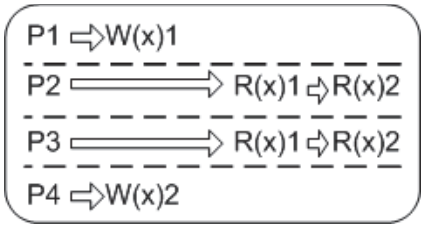

Figura 3.2: Acesso com consistência seqüencial (TANENBAUM; STEEN, 2007)

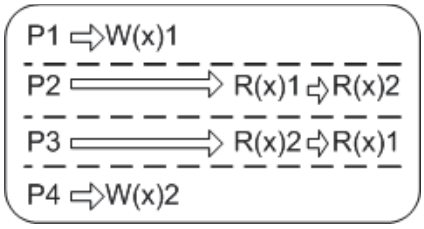

Figura 3.3: Acesso sem consistência seqüencial (TANENBAUM; STEEN, 2007)

Na figura 3.2, percebe-se que há uma seqüência nas operações de leitura e escrita de dados, se o dado "`1`" for o primeiro a ser escrito ele deve ser o primeiro a ser lido, ou seja, deve haver uma seqüência de operações de leitura e escrita. Na figura 3.3 isto não ocorre, pois o processador *P3* faz a leitura primeiramente do dado "`2`" onde deveria fazer a leitura do dado "`1`" quebrando assim a sequência que deveria ser respeitada.

3.3.3.2 Modelo de Consistência Estrita

O modelo de consistência estrita (*Strict Consistency*), também chamado de memória atômica (LAMPORT, 1986) ou *linearizability* (HERLIHY M. P., 1990), é o mais simples e mais rígido de todos, onde uma operação de leitura deve retornar o valor de escrita mais recente, considerando um tempo global absoluto para realizar as operações.

Em um sistema com múltiplas máquinas isto se torna um problema, pois elas poderão estar localizadas em regiões geograficamente distantes, onde o tempo a ser considerado para a propagação dos dados é diferente para cada máquina.

Na figura 3.4, um exemplo de operações de escrita e leitura é apresentando, utilizando-se o modelo de consistência estrita. Na figura 3.5, considera-se um contra-exemplo, ou seja, onde não se utiliza o referido modelo.

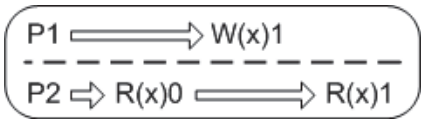

Figura 3.4: Acesso com consistência estrita (STEINKE; NUTT, 2004)

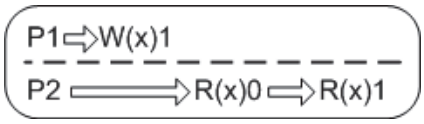

Figura 3.5: Acesso sem consistência estrita (STEINKE; NUTT, 2004)

Pode-se observar, na figura 3.4, que a operação de leitura ocorre um tempo após a operação de escrita na variável "x" estar terminada e ter sido propagada para os outros processadores do sistema. Neste caso o valor obtido da variável "x" pela operação de leitura será o último que foi escrito na variável "x". Já na figura 3.5, não se verifica o modelo de consistência estrita de memória, pois a operação de leitura ocorre antes da operação de escrita na variável "x" ter sido propagada para o processador *P2*, sendo o valor obtido igual a zero e não obtendo o último valor que foi escrito na variável "x".

Conclui-se então que este modelo de consistência é inutilizável em sistemas distribuídos, pois necessita de um tempo global absoluto para as máquinas envolvidas, sabendo-se que isto é uma limitação inerente dos sistemas distribuídos.

3.3.3.3 Modelo de Consistência Causal

Neste modelo de consistência, apenas as operações que potencialmente possuem uma relação causa/efeito devem ser vistos na mesma ordem pelos processadores. As operações concorrentes que não possuem essa relação podem ser vistos em ordens diferentes pelos processadores. Um exemplo de como isso ocorre é mostrado a seguir:

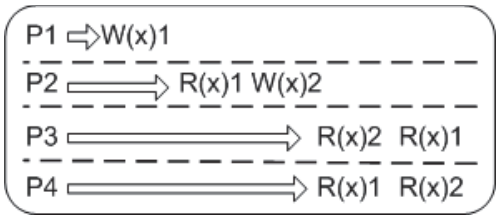

Figura 3.6: Acesso sem consistência causal (TANENBAUM; STEEN, 2007)

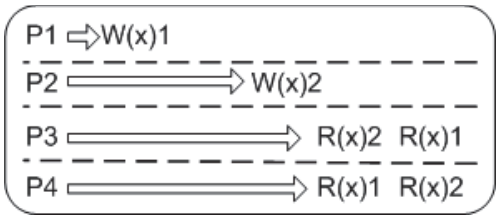

Figura 3.7: Acesso com consistência causal (TANENBAUM; STEEN, 2007)

Na figura 3.6, tem-se o modelo de consistência causal violado, pois assumindo que as operações "W(x)1" e "W(x)2" possuem uma relação causa/efeito, ou seja, dependem uma da outra, esta operações devem ser efetuadas antes que uma operação de leitura como "R(x)1" seja feita. O exemplo da figura 3.7, apresenta o modelo de consistência causal validado.

3.3.3.4 Modelo de Consistência de Processador ou PRAM

O modelo de consistência de processador, também chamado de PRAM (*Pipelined Random Access Memory*) segundo (LIPTON; SANDBERG, 1988) tem a seguinte definição:

"Escritas feitas por um único processador são vistas por todos os outros processos na ordem em que foram realizadas, mas escritas feitas por vários processadores talvez sejam vistas em uma ordem diferente pelos vários processadores".

A idéia básica da consistência PRAM é a que melhor reflete a realidade das redes atuais, nas quais a latência entre os nodos pode ser diferente.

O modelo de consistência de processador ou PRAM deve manter a consistência determinando que as escritas feitas por um único processador e no mesmo local de memória devem ser vistas na mesma ordem seqüencial por todos os processadores. Escritas feitas por vários processadores podem ser vistas em uma ordem diferente. As figuras abaixo mostram um exemplo de consistência de processador ou PRAM e outro que não possui esta consistência.

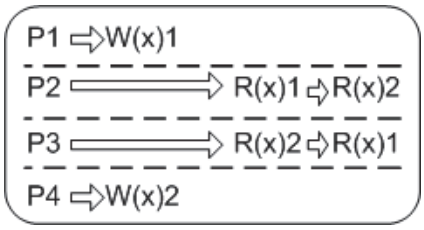

Figura 3.8: Acesso com consistência de processador/PRAM (STEINKE; NUTT, 2004)

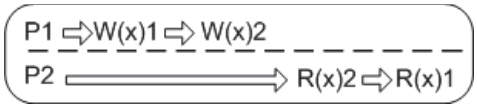

Figura 3.9: Acesso sem consistência de processador/PRAM (STEINKE; NUTT, 2004)

A figura 3.8 que apresenta um cenário inválido para consistência sequencial, tem validade para a consistência de processador ou PRAM, pois se considerarmos que os processadores estão interligados por um barramento linear, com certeza, o valor de "x" escrito por *P4* que é "2" será propagado primeiro para o processador *P3* que o valor escrito pelo processador *P1* em "x" que é "1". Na figura 3.9 o modelo de consistência de processador ou PRAM não se verifica, pois as operações realizadas por um único processador devem ser vistas na ordem que elas aconteceram, o processador *P2* ao realizar a leitura do dado "2" em "x" não segue a ordem de escritas que foram realizadas pelo processador *P1*.

3.3.4 Modelos de Consistência Sincronizada

Os modelos de consistência sincronizada utilizam mecanismos de sincronização explícitos, tal como variáveis de sincronização, introduzindo dessa maneira restrições adicionais às operações triviais de ordenação de memória. Como modelos de consistência sincronizada, têm-se: (i) o modelo de consistência fraca; (ii) o modelo de consistência de liberação; (iii) o modelo de consistência de liberação preguiçosa; e (iv) o modelo de consistência de entrada (STEINKE; NUTT, 2004). As principais características destes modelos são descritas na sequência.

3.3.4.1 Modelo de Consistência Fraca

O modelo de consistência fraca (*Weak Consistency*) introduz a utilização de variáveis de sincronização junto com operações de leitura e escrita de memória, considerando as seguintes propriedades:

- os acessos às variáveis de sincronização devem ser sequencialmente consistentes;
- nenhum acesso a uma variável de sincronização é liberado para execução até que todas as operações de escritas anteriores a esta variável de sincronização tenham sido completadas;
- nenhum acesso a dados, ou seja, operações de leitura e escrita na memória são liberadas para execução até que todos os acessos anteriores às variáveis de sincronização tenham sido completados.

Estas propriedades viabilizam a construção de condições em um ambiente de programação para a implementação de barreiras de sincronização e seções críticas (*locks* e *unlocks*), os quais constituem as primitivas mais comuns fornecidas por sistemas DSM para sincronização. As barreiras, por exemplo, determinam que todas as operações planejadas para serem executadas antes da barreira, devem, obrigatoriamente, obedecer esta restrição. Isso muitas vezes, pode parecer mais forte do que o necessário, pois as operações de sincronização podem ser usadas somente para importar ou exportar informações ou para obtenção (*Acquire*) ou liberação (*Realease*) de um *Lock* (STEINKE; NUTT, 2004). As figuras abaixo descrevem um exemplo de acesso a memória fracamente consistente, utilizando operações de sincronização:

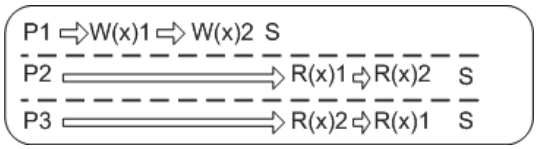

Figura 3.10: Acesso com consistência fraca (STEINKE; NUTT, 2004)

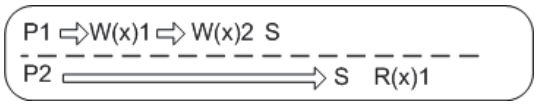

Figura 3.11: Acesso sem consistência fraca (STEINKE; NUTT, 2004)

Na figura 3.10, verifica-se o modelo de consistência fraca, pois todas as operações de leitura e escrita precedem a operação de sincronização. Já na figura 3.11, esta situação não acontece, pois a operação de leitura no processador *P2* faz a leitura do dado "`1`" em "`x`" após a operação de sincronização, quando a variável "`x`" assume o valor "`2`".

3.3.4.2 Modelo de Consistência de Liberação

Proposto por (GHARACHORLOO K., 1990), o modelo de consistência de liberação (*Release Consistency*) é uma otimização para ampliação do modelo de consistência fraca. Para alcançar uma consistência de liberação o sistema deve satisfazer as seguintes premissas:

- antes de qualquer acesso à memória por operações de leitura ou escrita, todas as operações de obtenção (*Acquire*) devem ter sido executadas;
- antes de um acesso de liberação (*Release*) ser executado, todas as operações de leitura ou escrita na memória devem ser executadas;
- acessos especiais como obtenção (*Acquire*) ou liberação (*Release*) devem ser sequencialmente consistentes.

A próxima figura apresenta um acesso à memória com consistência de liberação:

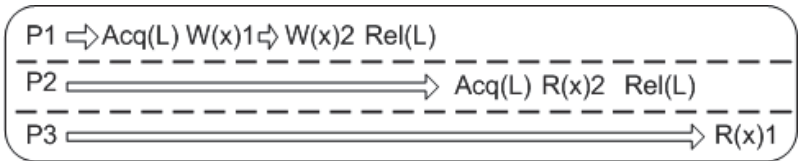

Figura 3.12: Consistência de liberação (STEINKE; NUTT, 2004)

Pode-se observar, na figura 3.12, que as três premissas citadas anteriormente se verificam, validando o modelo de consistência de liberação. Neste modelo, só é possível adquirir os dados mais atualizados de uma variável compartilhada através da realização de uma leitura desta variável dentro de uma seção crítica de código (*Acquire* e *Release*).

3.3.4.3 Modelo de Consistência de Liberação Preguiçosa

O modelo de consistência de liberação preguiçosa (*Lazy Release Consistency*), definido por (KELEHER P.; ZWAENEPOEL, 1992), está baseado no modelo de consistência de liberação, deixando menos rígidas as suas condições. Ou seja, não se exige que antes de um acesso de liberação, todas as operações de leitura e escrita na memória tenham sido executadas.

Este modelo procura retardar o máximo possível a atualização das variáveis compartilhadas. A atualização ocorre somente quando o processo ou processador requisitar aquele valor atualizado.

3.3.4.4 Modelo de Consistência de Entrada

O modelo de consistência de entrada (*Entry Consistency*) definido por (BERSHAD; ZEKAUSKAS, 1991), estabelece que cada variável de sincronização seja associada com uma ou mais variáveis compartilhadas.

Obtenções (*Acquires*) ou liberações (*Releases*) só mantém atualizadas variáveis compartilhadas associadas com determinadas variáveis de sincronização.

O modelo de memória do JAVA é equivalente ao modelo de consistência de entrada (STEINKE; NUTT, 2004), pois os *locks* são associados com os objetos, e valores armazenados em caches locais somente são requisitados para leitura em uma memória global quando *locks* são obtidos (*Acquired*) ou liberados (*Released*). A figura 3.13 apresenta um acesso de leitura e escrita na memória utilizando a consistência de entrada:

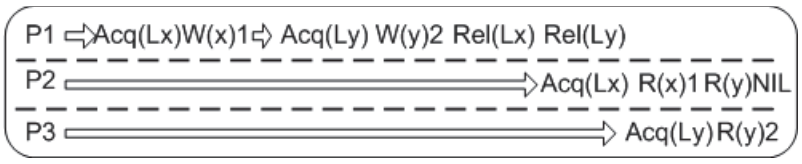

Figura 3.13: Consistência de entrada (TANENBAUM; STEEN, 2007)

Verifica-se também na figura 3.13, que cada variável compartilhada possui um *lock*, ou seja, uma variável de sincronização, validando o modelo de consistência de entrada. Quando o processador *P2* tentou efetuar a leitura de uma variável compartilhada sem, antecipadamente, realizar uma obtenção (*Acquire*) do seu *lock*, o valor retornado foi nulo.

3.3.5 Considerações sobre Modelos de Consistência de Memória

O estudo desenvolvido na seção 3.3 mostra que os modelos de consistência sincronizados requerem um esforço maior do sistema, pois exigem uma memória global sincronizada. Isso se torna aceitável diante da solidez destes modelos. Outro fator relevante é o maior custo gerado por cada acesso em memória de um sistema sequencialmente consistente em relação ao custo da sincronização. Dessa forma, pode-se afirmar que, os modelos sincronizados apresentam vantagens sobre os modelos ordenados, sendo os mais utilizados atualmente. Os modelos sincronizados ainda apresentam uma solução viável para a degradação de desempenho provocada pela manutenção da consistência. Esta solução se dá através do relaxamento da consistência, ou seja, as atualizações podem ser propagadas somente quando necessárias para um determinado dado compartilhado.

3.4 Propagação de Escritas

Um sistema de memória compartilhada distribuída deve possuir algum mecanismos capaz de enviar as escritas ou o conteúdo atualizado para as cópias locais dos dados compartilhados com o intuito de manter o sistema consistente, obedecendo algum dos modelos de consistência mencionados nas seções 3.3.3 e 3.3.4. Os mecanismos mais utilizados em sistemas DSM são os protocolos de envio de atualizações e os protocolos de invalidação, brevemente descritos logo a seguir.

3.4.1 Protocolos de Envio de Atualizações

Os protocolos de envio de atualizações de escrita garantem que todos os processadores que mantém a cópia dos dados irão ver a atualização do novo valor, simultaneamente, incluindo o processador onde este dado foi modificado.

A cada operação de escrita, as cópias replicadas devem receber o novo valor ao invés de uma mensagem de notificação.

Se um dado em um processador é atualizado várias vezes, todas essas modificações serão repassadas para as cópias replicadas nos outros processadores, mesmo que não haja operações de leitura a partir desses processadores, produzindo um maior tráfego na rede, mas tornando o acesso aos dados atualizados mais rápido.

3.4.2 Protocolos de Invalidação

Os protocolos de invalidação garantem a coerência dos dados a partir da invalidação de todas as cópias de um determinado dado modificado em um processador.

Esta invalidação ocorre por meio de uma notificação que os dados foram atualizados, e assim, os nodos do sistema que mantém uma cópia local destes dados são informados que os mesmos não são mais válidos, determinando que uma nova cópia deve ser feita, quando necessário, a partir da cópia onde ocorreu a modificação.

Uma das vantagens deste protocolo é utilizar um menor número de mensagens de notificação, minimizando o tráfego na rede, mas em contrapartida tornando o acesso aos dados atualizados mais demorado pois é necessário buscá-los no nodo servidor ou no nodo onde foi feita a modificação destes dados.

3.5 Coerência de Cache

Em essência, uma cache é um recurso de armazenamento local usado por um ou mais nodos do sistema para armazenar uma cópia dos dados que ele acabou de requisitar (TANENBAUM; STEEN, 2007). As caches locais são usadas, principalmente, para melhorar o desempenho no tempo de acesso aos dados compartilhados.

Particularmente, a cache é um tipo importante de réplica de dados compartilhados, controlada pelo nodo que requisitou esta réplica. Esta situação introduz a necessidade de manter a consistência dos dados nas caches presentes nos nodos do sistema, sendo esta manutenção de consistência nos nodos do sistema denominada coerência de cache.

Geralmente o termo coerência se refere a um item de dados em particular e não a todo o conjunto de dados do sistema como ocorre com a consistência. Os principais instrumentos para realizar a manutenção da consistência das caches são os protocolos de coerência de cache.

Os protocolos de coerência de cache desempenham um importante papel em sistemas paralelos ou distribuídos que trabalham com memória compartilhada. Tais protocolos podem ser implementados completamente em *software*, como no caso de sistemas distribuídos baseados em *middleware*, os quais são construídos baseados em sistemas operacionais de uso geral. A performance dos sistemas de memória compartilhada dependem muito dos protocolos de coerência de cache responsáveis por manter uma visão coerente dos dados replicados (SHEN; ARVIND; SHEN, 1997)

Como no caso das propagações de escrita da seção 3.4, existem duas abordagens para se definir protocolos que realizem a imposição de coerência de cache. A primeira é permitir que um servidor envie uma invalidação a todas as caches sempre que um item de dados for modificado. A segunda é, simplesmente, propagar a atualização. A maioria dos sistemas de cache usa um desses dois esquemas (TANENBAUM; STEEN, 2007).

4 DSM - IMPLEMENTAÇÕES EM *SOFTWARE*

Atualmente as implementações em *software* de DSM são as mais comuns e devido a isso são objetos de estudo neste trabalho. As implementações em *hardware* ainda apresentam custo elevado, limitando suas implementações. Destacam-se, neste capítulo: (i) os algoritmos mais frequentemente utilizados nestas implementações, seção 4.1; (ii) a descrição das principais implementações em *software* de DSM, categorizadas de acordo com a organização dos dados compartilhados, em páginas ou objetos e apresentadas nas seções 4.2 e 4.3; (iii) a avaliação das implementações em *software* de DSM, organizadas tanto em páginas quanto em objetos, considerando as caracterizações mais relevantes apontadas na literatura; e (iv) a descrição das principais implementações em *software* de DSM para JAVA, uma especialização das implementações baseadas em objetos, na seção 4.3.3, realizando uma análise comparativa entre elas na seção 4.5.

4.1 Algoritmos de Implementação de DSM

Os algoritmos de implementação de sistemas DSM desempenham importantes funções capazes de lidar com os aspectos mencionados nas seções 3.2, 3.3, 3.4 e 3.5 que são:

- Distribuição dos dados compartilhados pelo sistema, adotando algum tipo de estratégia e com o intuito de minimizar a latência do sistema;
- Propiciar que o sistema se mantenha consistente como um todo, evitando que os nodos obtenham dados desatualizados; e
- Manter uma visão coerente dos dados compartilhados presentes como cache nos nodos do sistema, minimizando a sobrecarga gerada pelo gerenciamento da coerência.

Segundo (STUMM; ZHOU, 1998) pode-se classificar os algoritmos existentes para implementação de sistemas DSM em quatro categorias:

1. Algoritmo de Servidor Central: Estabelece um servidor central que mantém e administra todos os dados compartilhados e a comnicação entre os clientes e servidor ocorre a partir das operações de leitura e escrita. Esta abordagem impede a replicação de dados compartilhados para os clientes e não permite a migração dos dados compartilhados, ou seja, há uma centralização das informações.

2. Algoritmo de Migração (SRSW - *Single Reader Single Writer*): Determina que os dados compartilhados devem ser enviados para o nodo onde eles foram requeridos, permitindo que as operações de leitura e escrita sobre estes dados sejam executadas localmente, diminuindo o tempo de latência. A restrição nesse algoritmo é que somente um processo ou *thread* pode acessar um determinado dado de cada vez, para executar operações de leitura e escrita.

3. Algoritmo de Replicação para Leitura (MRSW - *Multiple Readers Single Writer*): Trabalha com cópias replicadas dos dados compartilhados e permite a execução local de operações de leituras em cada nodo do sistema. Somente um processo ou *thread* possui permissão para atualizar uma cópia replicada em um dado momento. Estes algoritmos são geralmente baseados em invalidação, ou seja, após um processador escrever sobre um determinado item de dados compartilhado ele deve invalidar todas as outras cópias desse item por meio de uma notificação.

4. Algoritmo de Replicação Total (MRMW - *Multiple Readers Multiple Writers*): Semelhante ao MRSW, mas permite que vários processos possam escrever ao mesmo tempo em um determinado item de dados compartilhados. Para que isto ocorra e seja possível manter os dados consistentes, um novo elemento chamado sequenciador é introduzido. O sequenciador atribui um número a cada alteração feita em um item de dados replicado, e logo após, envia as alterações para serem executadas nas demais réplicas de acordo com a ordem estabelecida pelos números atribuídos.

As abordagens apresentadas para a construção de algoritmos responsáveis por sistemas DSM são empregadas em algumas das implementações em *software* de DSM, cujo detalhamento é apresentado nas seções 4.2 e 4.3.

4.2 Implementações Baseadas em Página

As implementações em *software* de sistemas DSM baseadas em páginas trabalham com dados compartilhados organizados como páginas de memória. Nesta abordagem, as implementações podem utilizar ou não o hardware de memória virtual das máquinas interligadas. A avaliação considerou três sistemas DSM: IVY, Munin e Treadmarks, apresentados na sequência.

4.2.1 IVY

O IVY, acrônimo de *Integrated Virtual Shared-Memory*, foi o primeiro protótipo de sistema DSM desenvolvido na década de 80 como uma provável solução para problemas como: (i) estruturas de passagem de dados complexas; e (ii) migração de processos entre os processadores, gerados pelos sistemas de troca de mensagens (LI, 1988).

A sua implementação é uma extensão do mecanismo de memória virtual (TANENBAUM, 1992) presente nos sistemas computacionais.

A arquitetura do IVY é constituída de cinco módulos: Gerenciador de Processos (*Process Management*), Alocação de Memória (*Memory Allocation*), Inicialização (*Initialization*), Operação Remota (*Remote Operation*) e Mapeamento de Memória (*Memory Mapping*), cuja organização segue a hierarquia apresentada na figura 4.1:

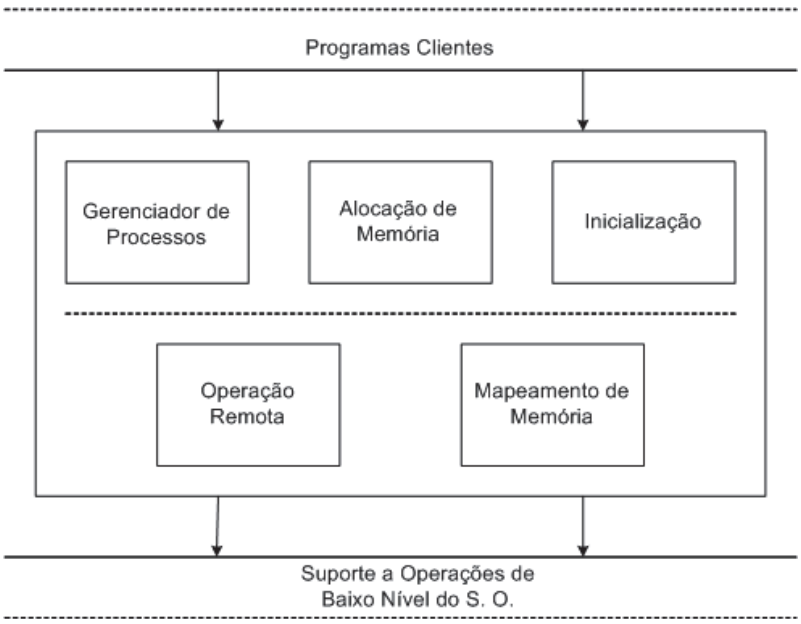

Figura 4.1: Hierarquia do IVY DSM adaptado de (LI, 1988)

Os três módulos superiores da hierarquia do IVY contém um conjunto de primitivas que podem ser usadas pelas aplicações que irão compartilhar a memória virtual. O módulo "Mapeamento de Memória" faz o mapeamento das memórias locais para o espaço de endereçamento da memória virtual compartilhada com a responsabilidade de manter o espaço de endereçamento coerente. O módulo "Operação Remota" realiza a comunicação entre os processos das aplicações que compartilham a memória virtual.

O IVY funciona como um sistema de memória virtual, ou seja, quando um processo tenta acessar uma posição de memória em uma página que não se localiza na sua memória física local acontece uma *page fault* que fica a espera da recuperação da referida página de memória do disco rígido. A única diferença é que no IVY, esta página deve ser trazida não do disco rígido, mas sim de outra máquina que compõe o sistema como mostra a figura 4.2.

Quando uma requisição por uma página é atendida, uma cópia desta página é enviada para o nodo que a está requisitando, isto gera múltiplas cópias da mesma página em diferentes nodos do sistema, as quais são controladas pelo IVY através de um algoritimo MRSW (*Multiple Readers Single Writer*) e um protocolo de invalidação dos dados desatualizados.

Quando um nodo escreve em uma página, todas as outras duplicações dessa página presente nos outros nodos, devem ser invalidadas para manter a coerência das cópias locais. O modelo de consistência de memória empregado pelo IVY é o modelo de consistência sequencial, cuja política é a manutenção de uma ordem total nos acessos a memória pelos processos, ou seja, todos os processos irão enxergar as escritas na memória na ordem determinada pelos processos que estão à executar, e não na ordem que elas realmente foram realizadas.

Este modelo é extremamente custoso em termos de comunicação entre os processadores, frequentemente prejudicando a performance do sistema (FENWICK, 2001). O problema de performance está relacionado diretamente ao *overhead* de comunicação provocado pelo fenômeno do compartilhamento falso (FENWICK, 2001). Este fenômeno

acontece quando o sistema mantém consistente a página de memória onde existem tanto variáveis compartilhadas quanto variáveis que não são compartilhadas e não precisariam da manutenção de consistência.

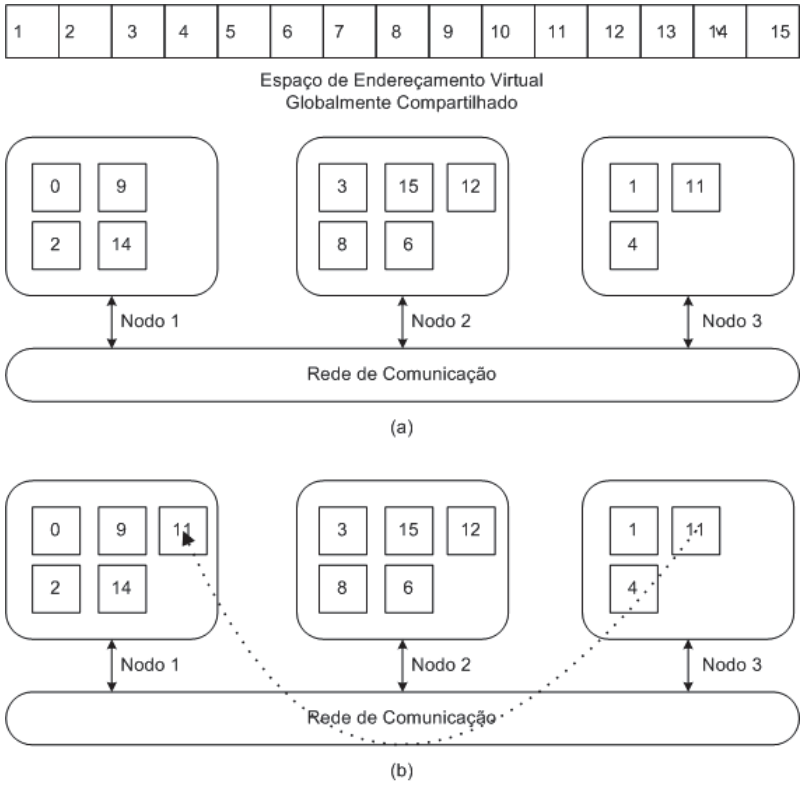

Figura 4.2: Funcionamento do IVY DSM adaptado de (FENWICK, 2001)

4.2.2 Munin

O *Munin* incorpora várias técnicas, na tentativa de tornar os sistemas DSM soluções viáveis para o processamento distribuído através da redução da quantidade de comunicação necessária para manter a consistência da memória distribuída (CARTER, 1995).

Uma das principais inovações propostas pelo *Munin* foi a utilização de múltiplos protocolos de coerência para manter a consistência do sistema, ou seja, cada variável compartilhada irá utilizar um determinado protocolo de coerência, e frequentemente, a melhor opção é escolhida. O monitoramento dos padrões de acesso à variável poderá ser usado para determinar o melhor protocolo de coerência.

A arquitetura do *Munin* mostrada na figura 4.3 apresenta como núcleo a biblioteca de tempo de execução (*runtime library*), cujas funcionalidades são: gerenciamento de *page faults*; suporte a *threads*; e sincronização.

Outras duas estruturas importantes da arquitetura do Munin são:

- Diretório de Objetos, cuja função é manter o estado dos dados compartilhados sendo usados pelas *threads* do usuário local;
- Fila de Atualização Atrasada (DUQ - *Delayed Update Queue*), a qual está encarregada de gerenciar a implementação do modelo de consistência de liberação no

software do Munin.

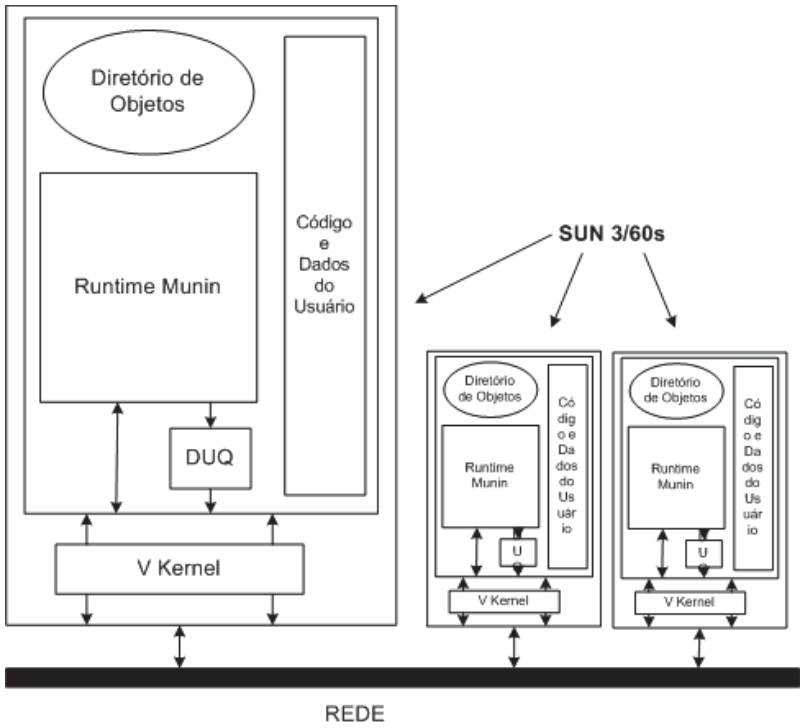

Figura 4.3: Organização do Munin adaptado de (CARTER, 1995)

Um nodo rodando sob o sistema DSM *Munin*, consiste em um conjunto de *threads* de tempo de execução que gerenciam a coerência, as operações de sincronização e os usuários que executam as computações paralelas.

O *Munin* é instalado como o gerenciador padrão de *page faults* do sistema operacional, portanto todas as exceções de memória do sistema operacional serão transferidas ao *Munin* para o devido tratamento. Através da manipulação do sistema de memória virtual do sistema operacional o *Munin* mantém uma visão coerente dos dados compartilhados. Inicialmente implementado sobre o Sistema Operacional V da *SUN*, o *Munin* pode também executar sobre sistemas Unix.

O funcionamento do *Munin* ocorre quando uma de suas *threads* de tempo de execução não acha uma entrada no diretório de objetos na tabela *hash* local. No *Munin* todas as variáveis em uma página são consideradas um único objeto, então, há uma requisição de cópia para o nodo onde a computação iniciou. Este nodo é chamado de nodo raiz e contém todos os dados compartilhados no início da execução do programa.

A chamada de dados compartilhados resulta em uma cópia destes dados para o nodo que os requisitou, ou seja, ocorre uma réplica dos dados requisitados do nodo de origem para o nodo requisitante. Como dito anteriormente, o *Munin* busca utilizar múltiplos protocolos de coerência, portanto, dependendo dos dados compartilhados os protocolos de atualização e invalidação, ou até a sincronização, podem ser utilizados para manter a coerência.

Os algoritmos utilizados podem também variar em função dos dados compartilhados, no *Munin* o MRSW (*Multiple Reader Single Writer*) ou o MRMW (*Multiple Reader Multiple Writer*) estão disponíveis.

4.2.3 Treadmarks

O *Treadmarks* é um sistema que fornece suporte a computação paralela em uma rede de estações de trabalho e sua principal característica consiste em prover um espaço global compartilhado de endereçamento de memória para os computadores do *cluster*. Mais especificamente, o *TreadMarks* é um *software* que permite a programação concorrente com memória compartilhada em uma rede de estações ou em um computador multiprocessador de memória distribuída (PARALLEL-TOOLS, 1994).

A estrutura do *Treadmarks* é constituída de uma matriz linear de *bytes* que fornece a memória compartilhada para as aplicações e o seu modelo de consistência de memória é o de consistência de liberação preguiçosa. Esta implementação utiliza o hardware de memória virtual para detectar os acessos à memória e um protocolo chamado *multiple writer* para amenizar os problemas resultantes dos erros entre o tamanho de página de memória e a granularidade do compartilhamento da aplicação (C. AMZA A.L. COX; ZWAENEPOEL, 1996). Também considera o protocolo de invalidação para atualização das escritas nas cópias replicadas.

Programas escritos em C, C++ e Fortran podem se ligar perfeitamente a biblioteca do *Treadmarks* e o sistema operacional utilizado é o Unix.

O funcionamento de *Treadmarks* ocorre quando a memória compartilhada é acessada por um nodo. Assim, o *TreadMarks* determina se os dados estão presentes naquele nodo e verifica a necessidade de transmitir os dados para aquele nodo, sem a intervenção do programador. Se a memória compartilhada é modificada em um determinado nodo, o *TreadMarks* assegura que os outros nodos que mantém uma cópia daquela área de memória modificada serão avisados da mudança, garantindo desta forma que os nodos não utilizarão dados obsoletos.

Esta notificação não é imediata e muito menos global, pois a notificação imediata é executada muitas vezes, gerando *overhead* e a notificação global alimenta os nodos com muitas informações não relevantes, gerando uma maior tráfego de rede. As notificações utilizadas pelo *TreadMarks* ocorrem no momento da sincronização de dois nodos, as quais são atrasadas e acumuladas em uma única mensagem a ser transmitida. Tal fato, tenta reduzir o *overhead* da comunicação inter-nodos(PARALLEL-TOOLS, 1994).

4.3 Implementações Baseadas em Objetos

Nas implementações baseadas em objetos, a memória é compartilhada entre os nodos de um maneira estruturada, na forma de um espaço de objetos compartilhados (FENWICK, 2001). Estes objetos compartilhados podem ser estruturas de dados simples, como as *structs* do C ou *records* do Pascal ou podem ainda conter métodos. Neste último caso o objeto além de conter os dados compartilhados, possui também uma interface para manipulá-los. Esta característica está presente nas linguagens JAVA e C++. Os sistemas DSM baseados em objetos, *Linda* e *ORCA* foram avaliados.

4.3.1 Linda

Linda foi um dos primeiros sistemas baseados em objetos desenvolvido, criado em 1986 introduziu o conceito de “espaço de tuplas”. Pode-se considerar uma *tupla* como uma unidade de memória elementar, consistindo essencialmente em *records* cujos campos são tipados. As *tuplas* contém os dados compartilhados entre os nodos do sistema, que

podem ser acessados através de algumas poucas operações baseadas em um mecanismo de reconhecimento de padrões de acesso, cuja semântica busca a solução dos problemas de sincronização de acesso a memória compartilhada.

A arquitetura do *Linda* consiste em: (i) um *kernel* de tempo de execução (*run-time kernel*), que implementa a comunicação e a gerência de processos; e (ii) um pré-processador ou compilador (AHUJA; CARRIERO; GELERNTER, 1986). Este *kernel* do *Linda* é independente de linguagem, mas somente testes com conjuntos de bibliotecas C e FORTRAN foram realizados. O *Linda* disponibiliza também operadores que devem ser utilizados nas implementações para realizar a programação paralela de uma aplicação.

Fazendo uma analogia do espaço de memória do *Linda* com um espaço de memória convencional, pode-se dizer que:

- enquanto que na memória convencional a unidade de armazenamento é o *byte* físico, no *Linda* a unidade de armazenamento é a *tupla* lógica ou um conjunto ordenado de valores;
- na memória convencional, os elementos são acessados por endereços enquanto que no *Linda* não há endereços, os elementos são acessados pelo nome da *tupla*, representando este nome qualquer tipo de seleção de dados; e
- enquanto que o acesso a memória convencional é feito por duas operações *read* e *write*, no *Linda* há três operações *read* (comando *read*), *add* (comando *out*) e *remove* (comando *in*).

O *Linda* utiliza estratégias de replicação de *tuplas* para realizar o paralelismo entre os nodos. Neste contexto, não é possível realizar a atualização das *tuplas*, ou seja, a *tupla* deve ser, primeiramente removida, e depois deve ser novamente inserida com os dados atualizados, o que nos dá um tipo de invalidação para manter a coerência dos dados.

Salienta-se que o modelo de consistência de memória mais fortemente associado ao *Linda* é o modelo PRAM, onde a ordem das operações só é observada no momento do processamento de uma determinada operação.

O funcionamento do *Linda* ocorre a partir da comunicação entre os processos, que nunca acontece diretamente entre eles, mas através do espaço de *tuplas*. Para tal, três operações são definidas:

- `out(t)` - Adiciona a *tupla* `t` ao espaço de *tuplas*;
- `in(s)` - Recupera uma *tupla* `t` que é identificada pelo *template* fornecido por `s`. Nesta operação, o valor da *tupla* `t` é atribuído a `s` e a *tupla* `t` é retirada do espaço de *tuplas*. Caso uma *tupla* `t` não seja encontrada através do *template* fornecido por `s`, a execução do processo é paralisada até que uma *tupla* `t` do mesmo tipo do *template* fornecido por `s` seja encontrada;
- `read(s)` - Faz o mesmo procedimento que `in(s)` com a diferença que a *tupla* `t` permanece no espaço de *tuplas*.

A figura 4.4 mostra como o funcionamento do espaço de *tuplas* ocorre a partir do comando `Out("X",6,TRUE)` em um dos nodos: (i) a *tupla* `("X",6,TRUE)` é inserida no espaço de *tuplas* e fica disponível para os outros nodos; (ii) o nodo 3 executa um comando `Read(" X",int i, bool b)` para recuperar a informação de uma *tupla* através deste *template*, mantendo a referida *tupla* no espaço de *tuplas*; e (iii) Os valores `6` e `TRUE` serão atribuídos às variáveis `i` e `b` para o programa que está rodando no nodo 3.

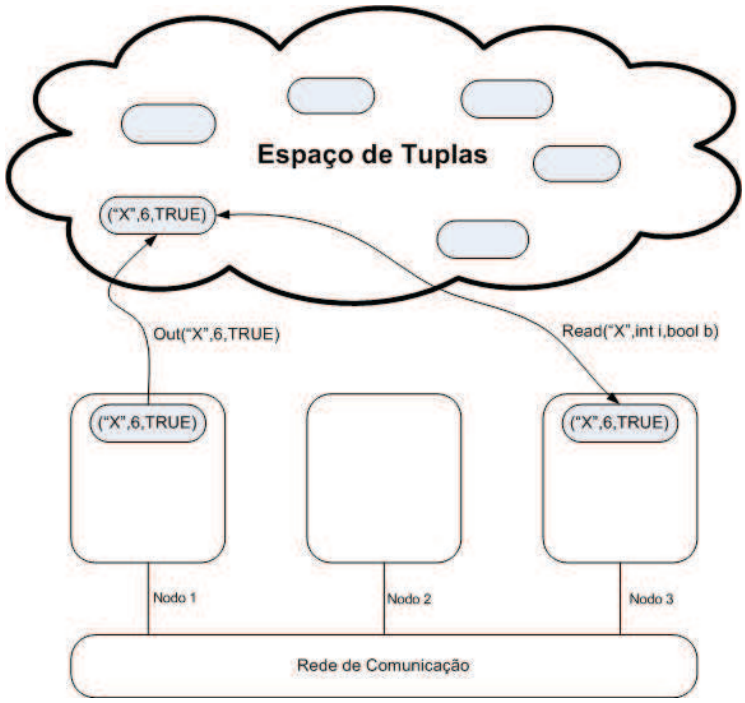

Figura 4.4: Funcionamento do Linda adaptado de (FENWICK, 2001)

4.3.2 ORCA

ORCA é um sistema DSM também baseado em objetos (BAL et al., 1998). O modelo de programação do *ORCA* foi projetado e implementado a priori para outros sistemas DSM (BAL H.E.; A.S., 1992).

Este sistema difere dos outros modelos de DSM por ser: (i) baseado em linguagem, ou seja, disponibiliza uma linguagem específica para a sua programação; e (ii) baseado em objetos para o compartilhamento da memória do sistema.

Neste sentido, pode ser considerado um precursor para o sistema *Terracotta* e outros sistemas DSM baseados em objetos. O *ORCA* encapsula os dados compartilhados em objetos e permite ao programador definir operações sobre objetos, usando tipos de dados abstratos ao contrário do que fazem os outros sistemas DSM que consideram páginas ou regiões de memória como unidade de compartilhamento (BAL et al., 1998).

A arquitetura do *ORCA* utiliza uma abordagem em camadas para que se possa oferecer portabilidade ao sistema. Várias camadas foram criadas, sendo que os sistemas de baixo nível dependentes de máquina foram isolados na camada mais inferior, tornando os componentes das camadas mais altas, como o compilador e o sistema de execução, totalmente independentes da máquina.

A máquina virtual utilizada é a Panda que fornece as primitivas de comunicação e de multitarefas. O sistema *ORCA* é implementado totalmente em *software* e necessita de um sistema operacional como Unix para fornecer as primitivas básicas de comunicação. Seu protocolo de coerência é o protocolo de atualização e seu modelo de consistência é o modelo de consistência seqüencial.

A figura 4.5 (ARAúJO, 2001) apresenta a arquitetura do *ORCA*, observa-se uma semelhança com os sistemas feitos para JAVA, pois também utiliza uma máquina virtual.

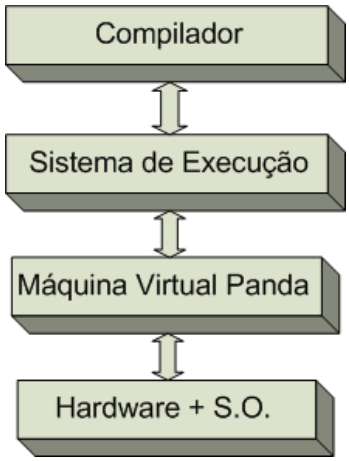

Figura 4.5: Arquitetura do ORCA

O funcionamento do *ORCA* inicia a partir de uma mescla entre migração e replicação dos dados compartilhados, ou seja, dos objetos. Quando há a ocorrência de atualização de algum objeto, o *ORCA* utiliza um protocolo de atualização de escritas para atualizar as cópias replicadas. Para tal, considera-se uma função de transporte (*function shipping*) que carrega a operação e os parâmetros que devem ser atualizados para todas as cópias locais dos objetos (BAL et al., 1998). Estas atualizações devem ser executadas na mesma ordem em todas as máquinas.

Outro fator interessante no funcionamento do *ORCA* é que ele procura replicar somente os objetos que possuem uma alta taxa de leituras e escritas mantendo para isso uma estatística de acessos aos objetos.

4.3.3 Sistemas DSM Desenvolvidos em JAVA

Pode-se definir um sistema DSM em JAVA como uma especialização dos sistemas DSM baseados em objetos, cuja função é permitir que um programa com múltiplas *threads* escrito em código JAVA possa ser executado pelos nodos de um *cluster* (FENWICK, 2001). Nesta seção, o modelo de memória do JAVA e os mecanismos de comunicação remota e sincronização são detalhados para posterior descrição e avaliação de três implementações de DSM para JAVA: *Terracotta*, *JESSICA* e *JavaParty*.

4.3.3.1 Modelo de Memória do JAVA

O JMM (JAVA *Memory Model*) define como as *threads* JAVA interagem com a memória compartilhada e mantém a sua consistência (FENWICK, 2001). Segundo (STEINKE; NUTT, 2004), o JMM tem equivalência ao modelo de consistência de entrada (*entry consistency*) porque os *locks* são associados aos objetos como acontece no modelo de consistência de entrada onde cada variável de sincronização é associada com uma ou mais variáveis compartilhadas. Obtenções (*Acquires*) ou liberações (*Releases*) mantém atualizadas estas variáveis compartilhadas associadas com variáveis de sincronização. A figura 4.6 mostra a estrutura e funcionalidade do JMM.

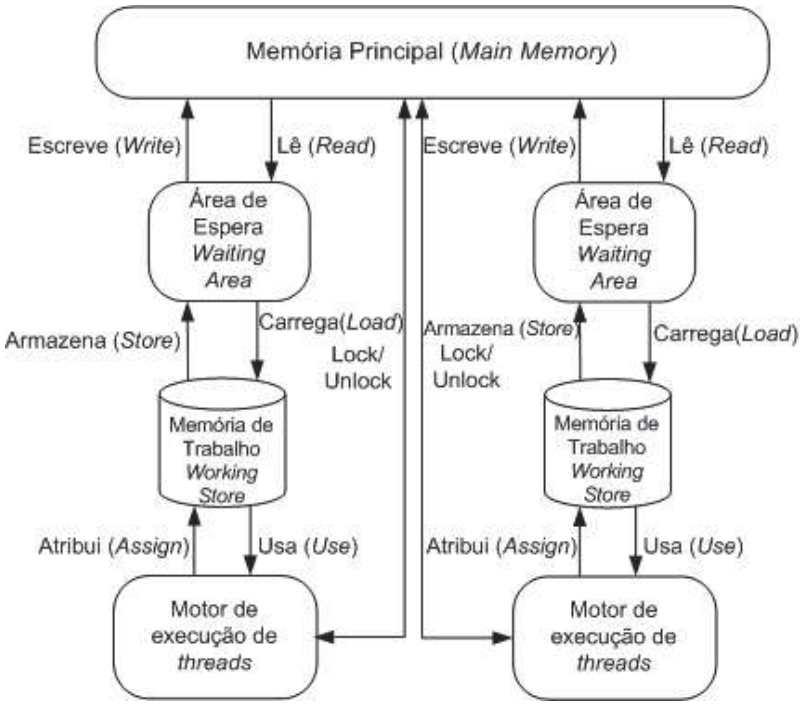

Figura 4.6: Modelo de Memória do JAVA adaptado de (FENWICK, 2001)

Na estrutura do JMM cada *thread* possui uma memória de trabalho que armazena a cópia das variáveis que são trabalhadas pela *thread* através das operações *assign* e *use*. A memória principal e a memória de trabalho interagem, mas são fracamente acopladas.

A transmissão da atualização de uma variável para a memória principal é feita por uma *thread* através de uma operação *assign* transmitida para a memória de trabalho. Esta operação atribui à variável que está sendo atualizada, o conteúdo dela na memória de trabalho. A partir daí, a memória de trabalho chama uma operação de *store*, que só é recebida pela memória principal se a memória principal executar uma operação de escrita (*write*).

Para a leitura de um variável pela *thread* o processo é semelhante, mas as ações *use*, *load* e *read* são usadas. As *threads* podem ser mutuamente excluídas de acessar certos objetos pelos *locks* associados com cada um destes objetos. Isto só pode ser feito por uma *thread* de cada vez e a *thread* que requisitar esse *lock* ficará bloqueada até o *lock* se tornar disponível. Quando uma *thread* obtém um *lock* ele deve limpar sua memória de trabalho e quando a *thread* libera esse *lock* ela deve propagar todos os valores assinalados (*assigned*) durante a manutenção do *lock* de volta para a memória compartilhada.

4.3.3.2 RMI - Modelo para Comunicação Remota em JAVA

Por ser uma linguagem destinada a execução distribuída, o JAVA possui incorporado o modelo de comunicação remota entre objetos RMI (*Remote Method Invocation*). Esse modelo busca fazer uso de um método conhecido como serialização, onde algumas estruturas dos objetos são transformadas em um formato serial de *bytes* para armazenamento persistente ou para transmissão através de uma conexão de rede.

A estrutura do RMI permite que dois objetos, geralmente localizados em computadores ou JVMs diferentes, estabeleçam um canal de comunicação. Ao estabelecer a comunicação, o RMI implementa um mecanismo do tipo cliente/servidor, cujo objeto

cliente é aquele que faz a invocação remota de um método localizado em um objeto presente em outro computador ou JVM e o objeto que recebe a invocação remota se torna o objeto servidor.

O acesso a um objeto remoto acontece da seguinte forma: Quando um objeto invoca um método em um objeto remoto, ele faz essa chamada através da estrutura *stub*, os argumentos dessa chamada podem ser passados por referência ou cópia. Quando cópias dos parâmetros são passadas ao *stub*, ele tem a função de serializar estes parâmetros da chamada de método e enviar isto para a máquina remota. Se a passagem é por referência, esta referência se torna o *stub* do objeto local. O *stub*, além de conhecer o método que deve ser chamado através dos parâmetros que lhe foram passados, também conhece a identidade do objeto remoto. No lado do servidor, o *skeleton* (*object receiver*) deserializa os parâmetros e, assim usa-os para fazer a chamada local ao método desejado da classe apropriada. O *skeleton* faz também o envio do resultado dessa chamada de volta para o *stub* no lado cliente.

Para um melhor entendimento a comunicação do RMI é apresentada graficamente na figura 4.7

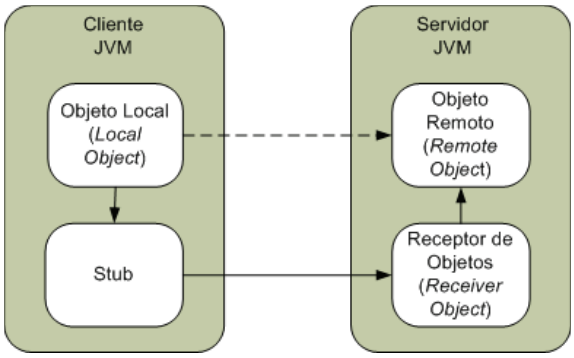

Figura 4.7: Comunicação em RMI adaptado de (FENWICK, 2001)

O uso das estruturas *stub* e *skeleton* (*object receiver*) para implementação da invocação remota de métodos introduz uma latência na rede para acesso ao objeto alvo devido as serializações e deserializações de parâmetros necessárias para execução dos métodos remotos e também para retorno de resultados. Segundo (PHILIPPSEN; HAUMACHER; NESTER, 2000), a serialização de objetos contribui com uma porcentagem de até 65% do custo de execução de uma invocação remota de método, ou seja, a conversão dos objetos em *bytes* é um trabalho que consome tempo. Além disso, a quantidade de dados transferidos pela rede também é maior ao usar RMI, pois além dos parâmetros dos objetos, dados referentes a controle e gerenciamento de mensagens, bem como suporte a exceções e gerenciamento de segurança devem ser transmitidos, causando *overhead* de comunicação na rede.

4.3.3.3 Mecanismos de Sincronização

No momento de acesso aos dados compartilhados, as *threads* devem ser bem coordenadas para que estados inconsistentes não sejam vistos por elas. O mecanismo de *threads* possui dois tipos de sincronização suportados pela estrutura de monitores do JAVA:

- Exclusão Mútua: Presente na linguagem JAVA através do comando *synchronized*

e construída na JVM através dos *locks* de objetos. Todo objeto tem um *lock* associado a ele. Os *locks* são usados para assegurar que as seções críticas de código, sejam métodos inteiros ou blocos de código, sejam executados por apenas uma *thread* (FENWICK, 2001).

- Cooperação: A JVM provê também um mecanismo de monitor através das operações *wait* e *notify*, proporcionando a uma *thread* já possuidora de um monitor suspender a ela mesmo através da chamada do método *wait*. Quando isso acontece, a *thread* que detém o monitor libera o controle dele e entra em um área de espera (*wait set*) onde ficará até outra *thread* chamar o método *notify* durante o controle do monitor, permitindo que a *thread* em espera possa readquirir o monitor. Quando uma *thread* chama o método *notify*, ele espera que esta *thread* saia do monitor para que a *thread* em espera possa reentrar nesse monitor. A chamada de *notify* não faz com que a *thread* notificada entre no monitor imediatamente.

  O esquema de monitores é apresentado na figura 4.8.

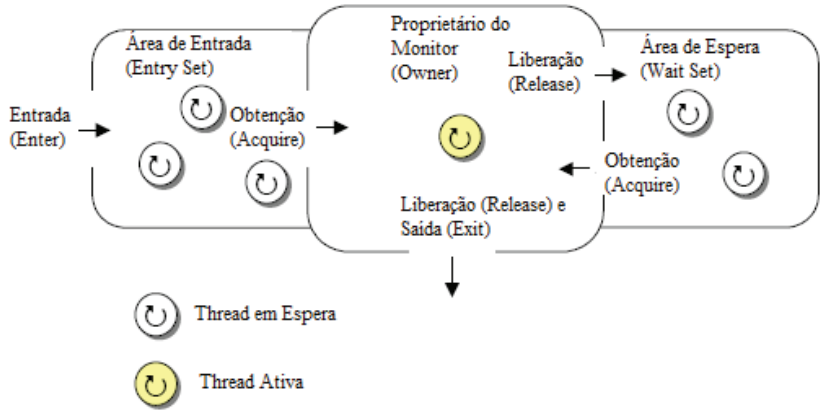

Figura 4.8: Esquema de Monitores em JAVA (FENWICK, 2001)

4.3.4 Terracotta

O *Terracotta* é uma tecnologia de memória compartilhada distribuída de código aberto para a máquina virtual JAVA. Em outras palavras, pode-se dizer que o *Terracotta* é uma tecnologia de *clusterização* em JAVA (TERRACOTTA, 2008). Ele cria um *Heap* JAVA, espaço de memória alocado dinamicamente, virtual e persistente, que é compartilhado através de um *cluster* de JVM's (Máquinas Virtuais Java). Este *Heap* é criado dinamicamente em nível de instrumentação de *bytecode* e, no momento de carga das aplicações, as instruções de leitura e escrita são interceptadas.

As informações sobre as instruções que foram interceptadas são então transmitidas para o servidor *Terracotta* que as envia para outros nodos no *cluster*, quando necessário. Através dessa abordagem muitos programas em JAVA podem ser *clusterizados* ou distribuídos sem modificações no código. Na figura 4.9, pode-se observar a localização do *Terracotta* no sistema.

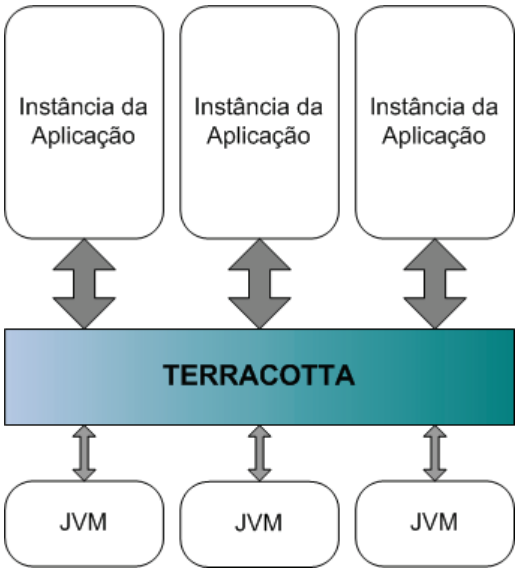

Figura 4.9: Localização do *Terracotta* em um Sistema Genérico.

O *Terracotta* utiliza uma arquitetura cubo-e-raio (*hub-and-spoke*), onde as máquinas virtuais JAVA (JVMs) que executam a aplicação *clusterizada*, conectam-se ao servidor *Terracotta* na inicialização. O servidor *Terracotta* armazena os dados de objeto e coordena tanto a concorrência das *threads* quanto a divisão de um processo em uma ou mais tarefas, entre as JVMs. As bibliotecas DSO (*Distributed Shared Objects*) presentes dentro das aplicações executando nas JVMs manuseiam a instrumentação do *bytecode* em tempo de carregamento de classes e transferem os objetos de dados, incluindo estruturas de código do tipo: (i) requisições de *lock*; (ii) *unlock* das barreiras de sincronização; (iii) requisições *wait()*, que suspende a execução de uma *thread* e a mantém em estado de espera; e (iv) *notify()*, cujo objetivo é viabilizar que as *threads* suspendidas e em estado de espera voltem a execução, entre a aplicação da JVM e o servidor *Terracotta*, em tempo de execução (LETIZI, 2007). A figura 4.10 mostra a arquitetura do *Terracotta*.

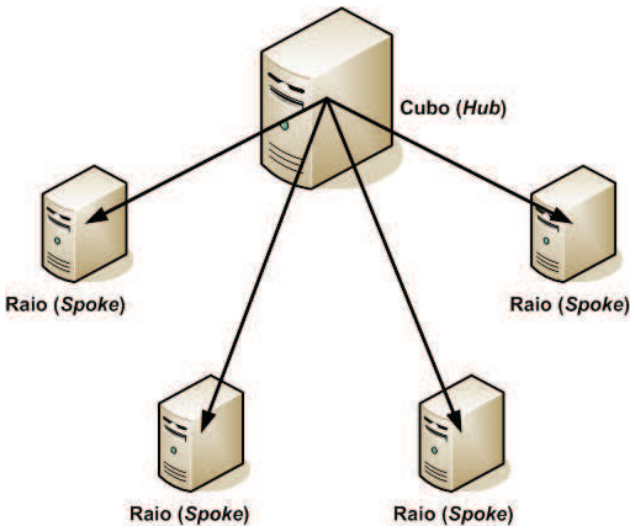

Figura 4.10: Arquitetura do *Terracotta*

Para coerência dos dados, mantém-se uma estratégia de replicação através de um protocolo de atualização, ou seja, quando um dado é atualizado, essa atualização é transmitida para todas as cópias replicadas.

O modelo de consistência de memória utilizado pelo *Terracotta* é equivalente ao modelo de consistência de entrada (*Entry Consistency*), pois, como mencionado anteriormente, este é o modelo de consistência compatível com modelo de memória do JAVA.

A comunicação entre os processos ocorre através da disponibilização dos objetos pela tecnologia *Terracotta* DSO (*Distributed Shared Objects*), os quais são compartilhados e distribuídos através do *cluster*. Estes objetos ficam acessíveis para todos os processos e *threads* da aplicação, bem como as modificações feitas por um processo ou uma *thread* nestes objetos compartilhados.

O *Terracotta* funciona analogamente à um sistema de armazenamento arquivos, onde dispositivos de armazenamento podem ser modificados (cd, DVD, disco rígido), sem que a aplicação precise ser modificada para acessar estes arquivos. Isso quer dizer que um sistema de armazenamento de arquivos é transparente para as aplicações.

Da mesma forma, o *Terracotta* também é transparente, pois armazena os objetos remotamente no servidor *Terracotta* e estes objetos podem então ser replicados e gerenciados como em um servidor de arquivos.

4.3.5 JESSICA

JESSICA é um *middleware* que executa em cima de sistemas operacionais padrão Unix dando suporte a execução paralela de aplicações JAVA *multithreaded* em um *cluster* de computadores. O *JESSICA* faz o *cluster* aparecer como um único computador para as aplicações, ou seja, como uma imagem única do sistema.

Com suporte a migração de *threads*, o *JESSICA* permite a uma *thread* mover-se livremente entre as máquinas durante a sua execução. O compartilhamento global de objetos também está presente no *JESSICA* e é ajudado pelo subsistema de memória compartilhada distribuída (MA; WANG; LAU, 2000).

Na arquitetura do sistema *JESSICA*, mostrada na figura 4.11, o módulo chamado Espaço Global de Threads (*Global Thread Space*), é o ambiente de programação e execução, visto pelo programador da aplicação como um espaço único e global de *threads* formado por múltiplas CPUs (MA; WANG; LAU, 2000). Esta camada ilusória é provida por três subsistemas que gerenciam o redirecionamento das requisições de sistema, o compartilhamento da memória distribuída e a migração de *threads*:

- Subsistema de Execução de *Threads* Delta (*Delta Execution Thread Subsystem*) - fornece suporte a migração de *threads*;
- Subsistema de Redirecionamento de Mensagens Mestre/Escravo (Master/Slave Message Redirection Subsystem) - propicia suporte as operações transparentes de localização;
- Subsistema de DSM (DSM Subsystem) - responsável pela criação do espaço global de objetos e por dar suporte de acesso aos objetos em uma forma distribuída.

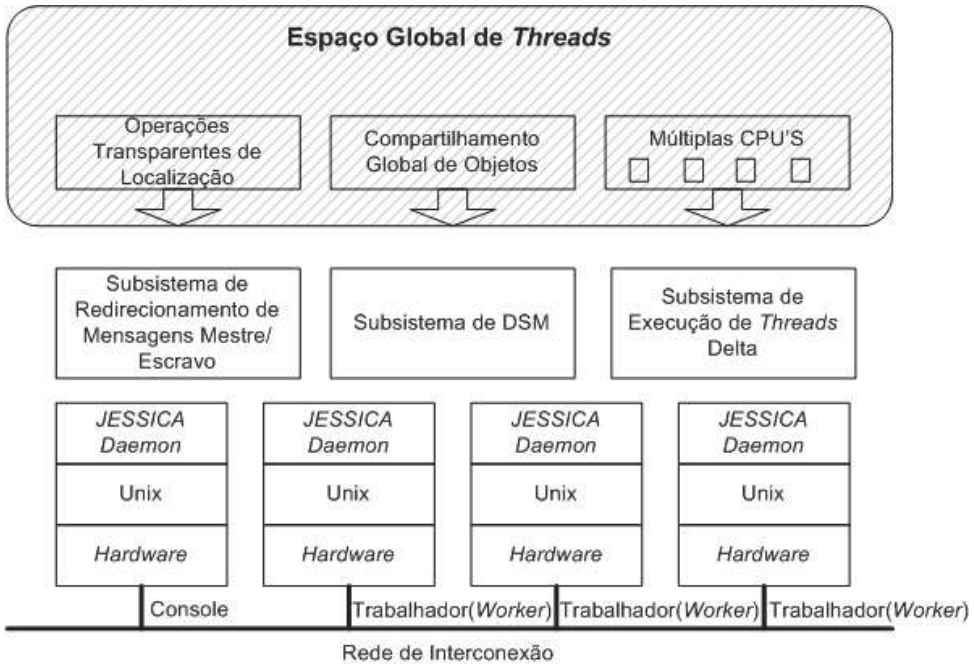

Figura 4.11: Arquitetura do *JESSICA* (MA; WANG; LAU, 2000)

Estes subsistemas são implementados através de processos *daemons* (*JESSICA daemon*) que executam nos nodos do *cluster*. Estes *daemons* são formados por quatro componentes de software responsáveis pela execução do *bytecode*, pelo gerenciamento de memória, pela criação de *threads* e pelo *scheduling* e sincronização de aplicações JAVA, agindo como uma JVM padrão (MA; WANG; LAU, 2000). Estes componentes são:

- BEE - Motor de Execução de Bytecode (*Bytecode Execution Engine*), responsável por endereçar uma *thread* ativa e executar o código de seu método;
- DOM - Gerenciador de Objetos Distribuídos (*Distributed Object Manager*), responsável por gerenciar os recursos de memória no nodo local e pela cooperação com outros DOM's em outros nodos com o objetivo de criar o espaço global de objetos;
- TM - Gerenciador de *Thread* (*Thread Manager*), responsável pela criação das *threads*, *scheduling* e finalização no nodo local. Durante a migração de uma *thread* ele se comunica com TM's de outros nodos para carregar o contexto de execução das *threads* migratórias. A distribuição das *threads* nos diferentes nodos é possível através do uso de sincronização com semáforos no TM console;
- MM - Gerenciador de Migração (*Migration Manager*), responsável por coletar as informações de carga do nodo local e repassar essa informação para os MM's que estão executando em outros nodos, determinando assim uma política de migração.

A especificação do *JESSICA* determina o uso de um paradigma mestre/escravo, ou seja, tem-se o nodo mestre, chamado de nodo console, que é o nodo onde a aplicação iniciou a execução. O nodo console é responsável por gerenciar o sistema de requisição de serviços de uma *thread* migrada que seja dependente de localização. Os nodos escravos, chamados de trabalhadores (*workers*), são os nodos que contém uma ou mais *threads*

migradas da aplicação. Estes nodos ficam subordinados ao nodo console e servindo as requisições direcionadas a partir do nodo console. A aplicação pode ser executada a partir de qualquer nodo do *cluster*, sendo este então o escolhido para ser o nodo console.

A operação do nodo console e trabalhadores (*workers*) no *JESSICA* acontece da seguinte forma: Durante a execução de uma aplicação, as requisições de serviços de sistema feitas pela *thread* migratória serão atendidas pelo nodo trabalhador (*worker*) interessado. O nodo trabalhador (*worker*) determinará se a requisição é independente de localização ou não. Se a requisição é independente de localização, ela é atendida localmente no nodo, se não é independente, a requisição é encaminhada para o nodo console. O nodo console, após receber a requisição, irá executar as operações necessárias e retornar o resultado de volta para a *thread* migratória. Todo esse processo de redirecionamento é feito de forma transparente.

O *JESSICA* usa um pacote do sistema *Treadmarks* para prover o subsistema de DSM necessário, portanto, o modelo de consistência usado é o de liberação preguiçosa bem como um protocolo de invalidação é usado para atualização das réplicas dos dados compartilhados nos nodos. A comunicação entre os processos executando nos diferentes nodos da rede é feita através da API BSD *sockets*, ou simplesmente, *Berkeley sockets* como definido em (PARALLEL-TOOLS, 1994) para o sistema *Treadmarks*. O balanceamento de carga no *JESSICA* é dinâmico, ou seja, as *threads* criadas pelas aplicações são automaticamente distribuídas através da rede com o objetivo de explorar o paralelismo.

4.3.6 JavaParty

O *JavaParty* fornece um espaço de endereçamento compartilhado, ou seja, mesmo que objetos de classes remotas estejam em diferentes máquinas, seus métodos e variáveis podem ser acessados da mesma forma que em JAVA puro. Uma vez que o *JavaParty* esconde o endereçamento e os mecanismos de comunicação do usuário e gerencia as exceções de rede internamente, nenhum protocolo de comunicação explícito é necessário nas implementações e projetos do programador (ZENGER, 1997). A figura 4.12 mostra os componentes do *JavaParty* e sua arquitetura.

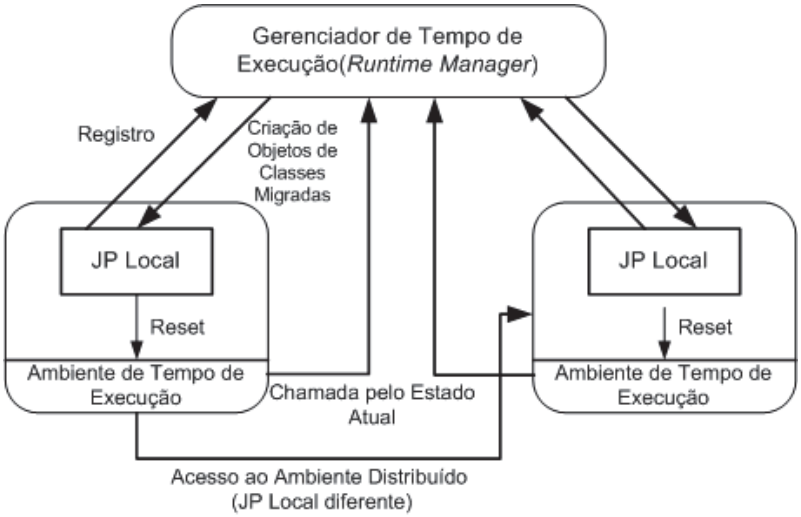

Figura 4.12: Arquitetura do JavaParty adaptado de (ZENGER, 1997)

Esta arquitetura, constitui-se de um sistema de tempo de execução (*runtime sys-*

tem), este sistema consiste em um componente central chamado Gerenciador de Tempo de Execução (*Runtime Manager*). Somado a isto, cada *host* executa um componente chamado *JP Local* que é registrado no Gerenciador (*Manager*). Um *host* e seu *JP Local* podem ser adicionados dinâmicamente ao sistema. O Gerenciador (*manager*) conhece todos os *JPs Locais* e também conhece a localização de todas as classes. Esta informação é replicada para os *JPs Locais* reduzindo a carga no Gerenciador (*manager*). Tanto o Gerenciador (*manager*) quanto os *LocalJPs* não precisam conhecer a localização individual dos objetos remotos. Os *JPs Locais* são necessários para os chamados de construtores das classes e para implementar a migração de objetos (ZENGER, 1997).

Os nodos do *cluster* controlados pelo sistema de tempo de execução (*runtime system*) do *JavaParty* realizam a comunicação através do RMI padrão ou também através de uma versão modificada do RMI, teoricamente mais rápida (FENWICK, 2001).

Devido as mudanças necessárias no código, o compilador a ser usado para compilar o código feito para o *JavaParty* deve ser o JPC (*JavaParty Compiler*). Após a compilação o código *JavaParty* é transformado em código JAVA puro e código RMI.

O modelo de consistência de memória usado é o mesmo do JAVA *Memory Model*, similar ao modelo de consistência de entrada, pois o código após a compilação se transforma em código JAVA puro e, portanto, todas as propriedades da linguagem JAVA e da JVM são utilizadas normalmente.

A distribuição de carga nos nodos é definida em tempo de execução segundo uma estratégia e de forma dinâmica e automática. A coerência deste sistema é mantida através de uma estratégia de migração de objetos para o nodo que requisita o método remoto.

O *JavaParty* entra em operação quando o Gerenciador de Tempo de Execução (*RuntimeManager*) é chamado em um nodo particular do *cluster*. As máquinas virtuais iniciadas em outros nodos se registram automaticamente no Gerenciador (*Manager*), através do envio por *broadcast* das suas localizações e esperam pelo Gerenciador (*manager*) para responder. O acesso aos métodos remotos é feito por uma estratégia de migração de objetos para o nodo que requisitou o método remoto deste objeto.

Quando o objeto é movido para um nodo diferente, ele deixa no nodo anterior um *proxy*. Se uma chamada de método acessa o *proxy*, uma exceção é gerada e junto com a exceção é enviada a nova localização do objeto movido. Essa informação é atualizada no nodo que fez a chamada e uma chamada, de método, para a nova localização é feita. Dessa forma, caracteriza-se a tentativa de reduzir o *overhead* do RMI, pois o acesso a objetos locais tem um custo menor de comunicação.

4.4 Avaliação das Implementações de DSM Baseadas em Páginas e Objetos

Esta seção procura fazer uma avaliação dos sistemas DSM baseados em páginas e objetos estudados, reunindo nas tabelas 4.1 e 4.2 as suas principais características e os principais problemas encontrados nestas implementações, os quais se tornam fatores impeditivos para o uso destes sistemas.

Sistema DSM	Organização dos Dados Compartilhados	Estratégia de Distribuição	Modelo de Consistência	Coerência de Cache
IVY	Páginas	Replicação	Sequencial	Invalidação
Munin	Páginas	Replicação e Migração	Liberação	Invalidação e Atualização
TreadMarks	Páginas	Replicação	Liberação Preguiçosa	Invalidação
Linda	Objetos	Replicação	PRAM	Invalidação
ORCA	Objetos	Replicação	Sequencial	Atualização

Tabela 4.1: Comparação entre Sistemas DSM parte 1

Sistema DSM	Abordagem do Algoritmo	Linguagem de Implementação	Plataforma	Problema Identificado
IVY	Replicação para Leitura(MRSW)	Aegis Modificado	Aegis	Overhead de Comunicação
Munin	Replicação para Leitura ou Total	C	S.O. da Sun ou Unix	Overhead de Comunicação
TreadMarks	Replicação Total	C	Unix	Overhead de Comunicação
Linda	Replicação Total	C e FORTRAN	Unix	Implementação Difícil
ORCA	Replicação Total	Linguagem Prop. *ORCA*	Máquina Virtual Panda	Sistema de Execução

Tabela 4.2: Comparação entre Sistemas DSM parte 2

Dessa forma, os principais problemas identificados nestas implementações foram:

- o *overhead* de comunicação, necessária para manter a coerência entre as cópias replicadas dos dados compartilhados, entre os nodos do *cluster*, cuja ocorrência torna o desempenho de execução de aplicações muito pobre. Este problema é comum aos sistemas baseados em páginas de memória como IVY, *Munin* e *TreadMarks*;

- a descontinuidade dos projetos referentes as estes sistemas, limitando a portabilidade devido a exigência de *softwares* específicos para o funcionamento dos sistemas DSM, vide *ORCA*, e dificultando a implementação nos sistemas operacionais atuais por não existirem versões atualizadas das suas implementações, como acontece com o *Linda*. Além disso, a descontinuidade deste projetos também ocasiona uma incompatibilidade de código, pois muitas vezes são desenvolvidos em linguagens de programação diferentes das mais usadas atualmente; e

- a ocorrência de *false sharing*, problema diretamente ligado a ocorrência de *over-*

head de comunicação e também comum a todos os sistemas DSM baseados em páginas avaliados. O *false sharing*, geralmente acontece quando variáveis usadas por diferentes nodos do *cluster*, de forma independente, estão presentes em uma mesma página do sistema DSM. Desse modo, vários nodos irão acessar essa página para obter os seus dados independentes e esses acessos fazem com que toda a página seja atualizada para manter a coerência dos dados, mesmo que só uma parte destes dados presente na página necessite de atualização, causando uma comunicação extra não necessária.

4.5 Avaliação das Implementações de DSM para JAVA

As Tabelas 4.3 e 4.4 mostram a comparação entre os sistemas DSM para JAVA, uma especialização dos sistemas DSM baseados em objetos, considerados de acordo com alguns critérios importantes no que diz respeito as suas funcionalidades e desempenho.

DSM para JAVA	**Organização dos Dados Compartilhados**	**Estratégia de Distribuição**	**Modelo de Consistência**	**Coerência de Cache**
Terracotta	Objetos	Replicação	Consistência de Entrada	Por Atualização
JESSICA	Objetos	Replicação	Liberação Preguiçosa	Por Invalidação
JavaParty	Objetos	Migração	Consistência de Entrada	Migração

Tabela 4.3: Comparação entre Sistemas DSM para JAVA parte 1

DSM para JAVA	**Modificações no Código**	**Distribuição dos Processos**	**Balanceamento de Carga**	**Comunicação entre Objetos**	**JVM Alterada**
Terracotta	Não	Transparente	Dinâmico e Automático	*Terracotta* DSO	*Não*
JESSICA	Não	Transparente	Dinâmico e Automático	*BSD Sockets*	*Sim*
JavaParty	Sim (Variáveis Compartilhadas)	Transparente	Dinâmico e Automático	RMI	*Não*

Tabela 4.4: Comparação entre Sistemas DSM para JAVA parte 2

Através da avaliação deste apanhado de características dos sistemas DSM para JAVA, concluiu-se que o *Terracotta* possui algumas vantagens que as outras implementações de DSM para JAVA não apresentam.

O sistema *JavaParty* apresenta três peculiaridades não desejáveis para uma arquitetura de memória distribuída, como:

- a necessidade de modificações no código original da aplicação a ser executada fazendo uso deste sistema DSM;

- a utilização do protocolo de comunicação RMI, aumentando o tempo de execução das aplicações devido as serializações e também aumentando o volume de dados enviados na rede devido a transferência dos objetos serializados entre os nodos do *cluster*;

- o uso de migração como estratégia de distribuição dos dados compartilhados pode provocar um baixo desempenho, além de não permitir o acesso simultâneo aos dados compartilhados devido ao modo exclusivo de propriedade destes dados compartilhados por cada nodo do *cluster*.

Outro sistema DSM para JAVA ponderado, o JESSICA, também apresenta algumas características não apreciadas para que esta implementação sirva como uma memória distribuída:

- o uso da implementação de DSM baseada em páginas *TreadMarks* como subsistema de memória compartilhada distribuída acarreta os mesmos problemas já identificados anteriormente para o *TreadMarks*, como o *overhead* gerado pela comunicação necessária para manter os dados compartilhados coerentes e a ocorrência de *false sharing*, onde dados presentes em páginas de memórias são atualizados sem necessidade, pois as modificações em apenas uma parte destes dados gera a atualização da página de memória inteira, gerando toda uma comunicação que não seria preciso se fossem atualizados apenas os dados modificados;

- a utilização do modelo de consistência de liberação preguiçosa oferecido pelo *TreadMarks* tende a ser mais custoso pois exige que os acessos especiais como *acquire* e *release* sejam sequenciais, ou seja, devem aparecer para os nodos do sistema na ordem em que foram especificados na aplicação, não considerando o tempo real de execução e também define nas seções críticas de código (*acquire* e *release*) a escrita ou leitura de uma ou mais variáveis compartilhadas, diferentemente do modelo de consistência de entrada onde existem variáveis de sincronização associadas a cada uma das variáveis compartilhadas;

- a manutenção da coerência dos dados compartilhados por invalidação diminui o tráfego de dados na rede do *cluster*, mas gera um atraso no acesso aos dados compartilhados que precisam ser buscados após uma invalidação, ocasionando um menor desempenho;

- a necessidade de uma máquina virtual JAVA modificada, possibilitando a migração de *threads* proposta por esta implementação de DSM para JAVA limita a execução das aplicações conjuntamente com o *JESSICA*, pois é necessário uma JVM não padrão instalada nos nodos do *cluster*. Geralmente, os sistemas operacionais Linux trazem o ambiente JAVA padrão para o disparo de aplicações JAVA. Isto traz como consequência um menor suporte a heterogeneidade na execução de aplicações porque a JVM modificada só estará disponível via uma nova instalação, bem como uma diferente configuração de variáveis de ambiente.

O *Terracotta* apresenta-se como a implementação mais viável dentre estas analisadas devido as seguintes razões, de maneira oposta as outras implementações:

- não há o uso do protocolo RMI para comunicação entre os objetos com o intuito de diminuir o *overhead* gerado pela comunicação entre os nodos de um *cluster* quando estes executam aplicações distribuídas e dessa forma acelerar a execução destas aplicações que executam com o *Terracotta*;
- não são necessárias modificações no código JAVA da aplicação para se executar a mesma fazendo uso do *Terracotta*;
- faz uso de replicação dos dados compartilhados, garantindo uma maior segurança proporcionada pelos *backups* e também permitindo o acesso simultâneo aos dados compartilhados;
- não existe a necessidade de uso de uma JVM não padrão como no JESSICA, aumentando a facilidade de execução do *Terracotta* em um maior número de sistemas operacionais e possibilitando uma maior heterogeneidade para esta implementação.

A possibilidade de distribuição do processos nos nodos do *cluster* de uma forma transparente e de um balanceamento de carga dinâmico e automático são características comuns às três implementações avaliadas e satisfazem as necessidades de especificação de uma arquitetura de memória distribuída.

5 ARQUITETURA DSM PARA O AMBIENTE D-GM

O estudo baseado na revisão de alguns dos principais sistemas de memória compartilhada distribuída procurou identificar os requisitos referentes a funcionalidade e desempenho de um sistema DSM, os quais devem ser considerados para atender as demandas do Ambiente D-GM. A seção 4.5, no capítulo 4, possibilita uma análise para seleção de qual destes sistemas pode servir como base para a modelagem de uma memória distribuída para o módulo de execução do Ambiente D-GM.

Como principais requisitos considerados relevantes ao Ambiente D-GM destacam-se:

- Mecanismo de escrita e leitura de dados em memória do modelo GM: Verificando se o sistema de memória compartilhada distribuída possui intrinsicamente o mesmo mecanismo de escrita e leitura de dados em memória do modelo GM, o qual determina que para cada variável compartilhada em memória poderá haver múltiplos processos leitores e apenas um único processo escritor por vez;

- Modelo de consistência e tipo de manutenção de coerência usado: Considerando que os processos que compartilham as variáveis em um sistema de memória compartilhada distribuída não enxerguem estados inconsistentes dessa memória, mantendo a coerência das caches de forma viável, sem gerar muita comunicação extra (*overhead*);

- Avaliação atual do sistema a ser usado como modelo: Analisando se o sistema a ser empregado como base para modelagem da memória distribuída do Ambiente D-GM possui uma versão estável para testes e suas aplicações em projetos;

- Linguagem de desenvolvimento: Identificando se a linguagem de desenvolvimento do sistema DSM, que servirá como base para a modelagem, é compatível com o módulo de execução do Ambiente D-GM e se é necessário modificações no código para a implementação da memória compartilhada distribuída;

- Tecnologia de comunicação entre os objetos distribuídos: Determinando se a comunicação entre os objetos distribuídos pelo sistema DSM produz menos *overhead*, procurando garantir um melhor desempenho;

- Possibilidade de usar o sistema DSM como uma alternativa ao EXEHDA: Avaliando se o sistema DSM a ser usado como base para modelagem possui

também características de *middleware*, constituindo-se em uma alternativa ao *middleware* EXEHDA, onde inicialmente foi baseado o módulo de execução do Ambiente D-GM;

A partir da análise dos sistemas DSM, conforme esses requisitos determinou-se o *Terracotta* como melhor opção atual para servir como base para a modelagem da memória distribuída do Ambiente D-GM.

Justifica-se esta escolha com base nas seguintes considerações:

1. O *Terracotta* permite a escolha do mesmo mecanismo de escrita e leitura de dados em memória usado pelo modelo GM, onde apenas um processo pode escrever em uma variável compartilhada enquanto que múltiplos processos podem ler os valores dessa variável compartilhada;

2. É bem utilizado para distribuição de aplicações que necessitam de um maior poder de processamento gerado a partir das demandas;

3. É um sistema estável, está em constante atualização e foi desenvolvido na linguagem JAVA, a mesma linguagem usada no desenvolvimento do módulo de execução do Ambiente D-GM, herdando o seu modelo de memória e consistência;

4. A coerência no sistema *Terracotta* é mantida através da atualização dos objetos replicados e o balanceamento de carga é feito de forma dinâmica e automática;

5. O programador ao utilizar o *Terracotta* não necessita se preocupar com aspectos de comunicação e nem modificar o seu código existente;

6. Além disso o *Terracotta* busca diminuir o *overhead* da comunicação através do seu diferencial, ou seja, a não utilização do RMI para a comunicação entre os objetos; e

7. Possui características de *middleware*, de acordo com (TANENBAUM; STEEN, 2007), um *middleware* é uma camada de *software* entre uma ou mais aplicações e o sistema operacional e serviços de mais baixo nível, cuja finalidade proporcionar um grau de transparência de distribuição, ocultando da aplicação a distribuição dos dados, processamento e controle. Estas definições assemelham-se as definições do *Terracotta* mostradas na seção 4.3.4, permitindo dessa forma que o *Terracotta* seja também considerado um *middleware*.

A partir da definição do *Terracotta* como o sistema DSM escolhido para a modelagem da memória distribuída do Ambiente D-GM, um estudo mais aprofundado torna-se necessário para realizar a integração entre o *Terracotta* e o Ambiente D-GM.

5.1 Sistema de Memória Compartilhada Distribuída - Terracotta

Esta seção tem como objetivo detalhar como ocorre o funcionamento do sistema *Terracotta* quando da execução de aplicações JAVA sob este sistema, bem como especificar a integração entre aplicações JAVA e o sistema *Terracotta* através da execução de um exemplo mostrando e detalhando as seções de configuração necessárias.

5.1.1 Modelo de Memória do Terracotta

O modelo de memória do *Terracotta* busca seguir o mesmo modelo de memória do JAVA, ou seja, é necessário assegurar que todas as mudanças que aconteceram antes que uma *thread* seja autorizada a entrar em um bloco sincronizado sejam aplicadas a memória principal antes de o bloqueio ser concedido.

O *Terracotta* usa um modelo de memória próprio para garantir um comportamento e uma semântica apropriada entre os processos. Por exemplo, se o objeto O for modificado por um processo, chamado *processo1* em algum ponto anterior, outro processo, chamado *processo2* irá obter estas modificações antes de entrar em um bloco sincronizado que vai proteger o objeto O dos outros acessos. Então, as modificações do *processo1* neste caso, acontecem antes do *processo2* começar a trabalhar com o objeto O, permitindo que o *processo2* enxergue estas modificações feitas pelo *processo1*. Uma vez que *locks* são exclusivos, ou seja, somente uma *thread* é permitida dentro de um bloco sincronizado de cada vez, o modelo de memória se mostrou suficiente para definir a ordenação em um ambiente onde as *threads* estão espalhadas através dos processos JAVA.

5.1.2 Funcionamento Interno - Interceptação de Operações

O *Terracotta* funciona no mesmo nível da JVM, inserindo-se entre a lógica da aplicação e a sua memória, procurando por chamadas a *Heap*. Estas chamadas podem ser leitura (*reads*) da memória ou escritas (*writes*) na memória.

Além disso, o *Terracotta* considera dois principais aspectos ao trabalhar com um objeto que são as definições de classe e os dados da instância. O primeiro contém os métodos que implementam as regras do sistema e o modelo do objeto e, o segundo, contém somente os dados na memória. Assim, como as operações de leitura ou escrita em memória são resultantes de instruções JAVA que executam na JVM, uma instância de uma classe escreve na memória através do operador de atribuição (=) e lê dados na memória através do operador de acesso (*accessor*) (.).

5.1.2.1 Exemplo *PlayWithMemory*

Para um melhor entendimento, será demonstrado através do exemplo de código simples *PlayWithMemory* (TERRACOTTA, 2008) como o *Terracotta* procede em tempo de execução. O código é mostrado na listagem 5.1.1.

```
class MyObject {
int i=0;
// ...
int playWithMemory(int input) {
int j;
this.i=input;
j=this.i;
return j;
}}
```

Listagem 5.1.1: Exemplo *PlayWithMemory* - Código JAVA

Em tempo de execução, as poucas linhas de código no método *playwithmemory()* se transformarão em instruções para a JVM. A primeira linha se transforma em uma variável de alocação *stack* de 4 bytes onde a variável *j* será armazenada. A próxima linha

escreve na memória *Heap*. O endereço na memória onde ocorre a escrita é o endereço onde *this* está armazenado. O tamanho do tipo do campo *i* determina um *offset* (deslocamento) em *this*, onde a aplicação pode fazer a escrita. Neste caso *i* é um *int*, então o *offset* deve ser quatro. O argumento para o método *playwithmemory()* será copiado dentro da instância *this*, provocando dessa forma uma escrita de 4 *bytes* no *Heap*.

A próxima linha de código se transforma em uma leitura da memória *Heap*. Novamente, a operação de leitura ocorre no *offset* de 4 *bytes* dentro da nossa instância, onde *i* está armazenado. O valor de *i* é copiado para a variável stack chamada *j*. Na verdade, o acesso ao *Heap* ocorre de forma transparente, linha por linha dentro da JVM de acordo com a execução da lógica de nossa aplicação. Algumas lógicas irão provocar leituras da memória *Heap*, enquanto outras vão provocar escritas. Obviamente, a JVM pode executar muitas outras operações além das operações de leitura e escrita na memória, mas o *Terracotta* está primariamente baseado nestas duas operações de *bytecode*. Outras instruções dentro da JVM, as quais o *Terracotta* pode se associar são as instruções especiais *constructor* e instruções de *threading* tais como *locking* e *unlocking* de objetos. Mas, salienta-se que o conceito do núcleo do *Terracotta* é suficientemente bem representado pelas instruções de leitura e escrita.

O conceito de núcleo por trás do *Terracotta* é esta manipulação de *bytecode*. Quando *playWithMemory()* é chamado e uma operação de leitura na memória *Heap* ocorre com o *Terracotta* habilitado, a operação de leitura irá fazer mais do que somente ler da memória. Pode-se visualizar no pseudo código da listagem 5.1.2, que a lógica da aplicação não é modificada enquanto o *Terracotta* se prende ao objeto.

```
class MyObjectWithTerracottaPseudoCodeForm {
int i=0;
// ...
int playWithMemory(int input) {
int j;
// Terracotta se insere aqui devido a ocorrência de uma
operação de memória.
if(Terracotta.isOutOfDate(this.i)) {
Terracotta.updateFromNetwork(this.i);
}
this.i=input;
// manda deltas para a rede a partir da modificação
do objeto compartilhado
Terracotta.sendDeltasToTheNetwork(this.i);
// O Terracotta se insere aqui também, devido a operação
em memória.
if( Terracotta.isOutOfDate(this.i)) {
Terracotta.updateFromnetwork(this.i);
}
j=this.i;
// nenhum delta é enviado, pois esta é uma operação lógica
de leitura que não escreve em memória.
return j;
}}
```

Listagem 5.1.2: Exemplo *PlayWithMemory* - Pseudo Código

O *Terracotta* não se insere dentro dos arquivos fonte JAVA, ele se insere dentro das classes quando elas são carregadas. Estas classes vêm de arquivos de classes que o compilador escreve no disco na maioria dos casos. O *Terracotta* não entende a sintaxe do Java, porque ele não precisa fazer isto. Ele somente observa as operações lógicas *HEAPREAD()* e *HEAPWRITE()* e intercepta-as, como no pseudo código mostrado anteriormente.

5.1.3 Tecnologia de Comunicação Inter-Processos

As tecnologias de troca de mensagens tradicionais, como o RMI, não possuem uma memória compartilhada disponível para os processos. Como resultado, todos os dados representando o contexto da mensagem deve ser enviado com a mensagem acarretando em redundância de dados devido as múltiplas mensagens, congestionamento da rede e tomando tempo desnecessário do processador.

A tecnologia *Terracotta* DSO (*Distributed Shared Objects*) permite que os dados sejam compartilhados independentemente dos eventos de troca de mensagens. As mensagens inter-processos podem referenciar grandes conjuntos de dados sem fisicamente mandar estes dados através da rede junto com as mensagens. Dessa forma a rede permanece descongestionada e o processador fica livre para realizar trabalhos mais importantes (TERRACOTTA, 2008).

Pode-se observar que o *Terracotta* é uma tecnologia de comunicação, assim como o RMI, com a vantagem de oferecer um *Heap* virtual para armazenamento dos objetos que são compartilhados, distribuídos e modificados por todos os processos ou *threads* das aplicaçõe que executam em um *cluster*.

5.1.4 Possibilidades de Uso do Terracotta

O *Terracotta* pode fazer com que vários computadores funcionem como um único servidor lógico, podendo ser aplicado de várias formas e ter muitas possibilidades de utilização, dependendo da situação. O *Terracotta* trabalha agregado a JVM e busca eliminar os problemas de escalabilidade e disponibilidade quando da execução distribuída de aplicações JAVA e como tipos de uso básicos pode-se citar:

- Cache distribuído:

  Busca fornecer uma solução para as aplicações que necessitam executar em um *cluster* e compartilhar os dados através dos nodos deste *cluster*. Além de armazenar os objetos do usuário, o cache distribuído deve também receber as modificações realizadas pelos processos, e quando isso acontece, as modificações devem ser replicadas através dos nodos do *cluster*. De acordo com o tipo dos dados, pode ser importante que um mesmo objeto de usuário não seja modificado ao mesmo tempo em nodos diferentes;

- Banco de dados offload:

  Tem por objetivo eliminar os gargalos no acesso aos dados em um banco de dados, aumentando o desempenho. Estes gargalos muitas vezes são gerados por muitos acessos simultâneos para a realização de buscas que já foram efetuadas. O uso do *Terracotta* em conjunto com um *software* de ORM (Object-Relational Mapping) como o *Hibernate* (TERRACOTTA, 2008) permite armazenar as buscas realizadas em memória, disponibilizando estas buscas como objetos para outros usuários que buscam a mesma informação;

- Replicação de sessões:

  São características de servidores de aplicações como o *Apache Tomcat* (TERRACOTTA, 2008). Dessa forma, a infraestrutura transparente do *Terracotta* para o compartilhamento de objetos pode fazer a diferença no momento das replicações. Cada sessão replicada pelo servidor de aplicações, como o *Apache*, pode ser, por exemplo, uma nova instância de uma página *Web* presente no servidor. Ao compartilhar objetos nesta sessão replicada, os desenvolvedores devem usar a serialização de objetos para posterior envio ao servidor. O *Terracotta* minimiza esta serialização através do uso dos objetos compartilhados, aumentando desta forma o desempenho da sessão replicada.

- Particionamento de carga de trabalho:

  Atualmente conhecido por outros nomes como Computação em *Grid*. Neste tipo de abordagem, é comum a idéia de dividir para conquistar. O *Terracotta* pode fornecer esta arquitetura usando as estruturas de dados JAVA para a comunicação junto com *threads* coordenadas ou comunicação via JMS (JAVA *Message Service*) ou *sockets*. Essa abordagem emprega o paradigma *Master-Worker* para a divisão das tarefas. Assim, uma aplicação, através do nodo *Master*, pode distribuir a sua carga de trabalho computacional, como consultas ou atualização de grandes quantidades de dados, para os *Workers* do sistema, utilizando para isso uma fila (*java.util.queue*) para armazenamento das tarefas a serem executadas.

O objetivo deste trabalho é utilizar o *Terracotta* mais próximo de um cache distribuído aplicado ao Ambiente D-GM com um enfoque em fornecer uma memória compartilhada distribuída. Quando o *Terracotta* é usado para distribuir os dados da aplicação através da implementação de um cache, ele compartilha todas as coleções JAVA para todos os processos JAVA. Nestes casos, os processos das aplicações fazem uso compartilhado de estruturas como *maps*, *lists*, *arrays* e *vectors* pertencentes às ambas classes *Java.util* e *Java.util.concurrent*.

5.1.5 Integrando o Terracotta com Aplicações JAVA

O *Terracotta* pode ser integrado com aplicações POJO (*Plain Old JAVA Objects*), como o módulo de execução do Ambiente D-GM, o VirD-GM. Pode-se definir POJOs como aplicações JAVA que utilizam somente objetos JAVA regulares sem a implementação de interfaces específicas de infra-estruturas de *frameworks*.

Para realizar a integração é necessário a configuração de um arquivo XML chamado *tc-config.xml* com a configuração das classes da aplicação que serão executadas com o *Terracotta*. Este arquivo de configuração pode ser criado por um editor de texto, respeitando o nome do arquivo, que não deve ser mudado. O diretório de localização deve ser a pasta raiz onde estão localizados os arquivos do código fonte da aplicação. Junto com os arquivos de instalação do *Terracotta* é fornecido um template chamado *tc-config-pojo.xml*. Um exemplo é apresentado a seguir instruindo a configuração do *tc-config.xml*.

5.1.5.1 Olá Mundo dos Clusters

O exemplo *OlaMundodosClusters* apresentado nesta seção, adaptado de (TERRACOTTA, 2008), tem por objetivo apresentar os principais arquivos, configurações, objetos e classes usados na implementação de aplicações que fazem uso do *Terracotta*.

Primeiramente, mostra-se como configurar o arquivo *tc-config.xml*, distribuindo a aplicação em duas ou mais JVMs e, neste caso, diminuindo o tempo necessário para sua execução. Abaixo, na listagem 5.1.3, tem-se o código JAVA da simples aplicação que será distribuída em duas JVMs através do *Terracotta*.

```
public class OlaMundodosClusters {
 private static final String message="Ola Mundo dos Clusters!";
 private static final int length=message.length();
 private static char[] buffer=new char[length];
 private static int loopCounter;

 public static void main(String args[]) throws Exception {
  while(true) {
    synchronized(buffer) {
     int messageIndex=loopCounter++ % length;
     if(messageIndex==0) java.util.Arrays.fill(buffer,'\u0000');
     buffer[messageIndex]=message.charAt(messageIndex);
     System.out.println(buffer);
     Thread.sleep(100);
        }
      }
    }
  }
```

Listagem 5.1.3: Exemplo Olá Mundo - Código JAVA

O arquivo *tc-config.xml* possui algumas seções e subseções que devem ser configuradas de acordo com as informações da aplicação e também para determinar configurações de clientes e servidor, considerando que a partir da perspectiva do *Terracotta*, as JVMs executando a aplicação são clientes, embora na visão do desenvolvedor da aplicação estas JVMs sejam conhecidas como servidores da aplicação.

A primeira seção de configuração é a seção *servers*, mostrada na listagem 5.1.4:

```
<servers>
<server host="localhost">
<data>%(user.home)/terracotta/server-data</data>
<logs>%(user.home)/terracotta/server-logs</logs>
<dso>
<persistence>
<mode>permanent-store</mode>
</persistence>
</dso>
</server>
</servers>
```

Listagem 5.1.4: Seção *servers* - Exemplo Olá Mundo

Esta seção pode conter múltiplas configurações de servidores especificadas pela entrada *server host*. Se mais de um servidor for configurado, um destes servidores deve ser o servidor ativo primário. Não há necessidade de especificar esta opção pois os clientes

irão tentar conectar em todos os servidores seguindo a ordem do arquivo de configuração até encontrar o servidor ativo primário.

As entradas *data* e *logs* indicam onde armazenar as informações do servidor. A entrada *data* configura o caminho onde o servidor *Terracotta* irá armazenar informações do *cluster Terracotta*, como os objetos de dados do *cluster* e informações dos clientes conectados. A entrada *logs* configura o caminho para a gravação dos logs do servidor *Terracotta*. O exemplo considera como servidor a máquina local (*localhost*), pois será executado em uma única máquina.

Outra opção de configuração da seção *servers* é se o estado do *cluster* será persistente, opção *permanent-store*, garantindo o armazenamento permanente dos dados ou não, opção *temporary-swap-only*. Caso a opção *temporary-swap-only* seja configurada e o servidor *Terracotta* for reiniciado todos os dados do *cluster* e da aplicação serão inicializados novamente a partir desta reinicialização. Caso a opção escolhida seja *permanent-store*, os dados presentes no *heap* do *cluster* não serão destruídos, ficando disponíveis após uma possível reinicialização do servidor *Terracotta*. Será utilizada a opção *permanent-store* no exemplo com o intuito de observar o funcionamento do *heap* persistente.

A segunda seção do arquivo de configuração, seção *clients*, mostrada logo abaixo, na listagem 5.1.5, diz respeito aos clientes. Na maioria dos casos é preciso somente especificar o local onde o *Terracotta* irá gravar os *logs* das máquinas clientes. Outra configuração possível, nesta seção, é a definição de módulos de *software* necessários para a execução de uma aplicação. Em uma situação destas, deve-se informar no arquivo de configuração, qual módulo a aplicação cliente vai utilizar e onde estão localizados estes módulos. O exemplo avaliado não necessita de módulos.

```
<clients>
<logs>%(user.home)/terracotta/client-logs</logs>
</clients>
```

Listagem 5.1.5: Seção *clients* - Exemplo Olá Mundo

A seção *application* permite as configurações específicas da aplicação como quais classes serão instrumentadas pelo *Terracotta* e quais métodos devem ser estendidos para o gerenciamento de modificações de objetos compartilhados e para os *locks* do *cluster Terracotta*. Os campos de *root/field-name*, apresentados na listagem 5.1.6, definem quais objetos, em termos de variáveis, serão compartilhados (*clusterizados*) pelo *Terracotta* para todas as JVMs que fizerem parte do *cluster*.

```
<application>
<dso>
<roots>
<root>
<field-name>OlaMundodosClusters.buffer</field-name>
</root>
<root>
<field-name>OlaMundodosClusters.loopCounter</field-name>
</root>
</roots>
```

Listagem 5.1.6: Seção *application* - Subseção *root/field-name* - Exemplo Olá Mundo

Mais especificamente, no exemplo *OlaMundodosClusters*, as duas variáveis, *buffer* e *loopCounter* devem ser compartilhadas, pois são de fundamental importância para a correta execução do programa. A variável *array* de tipo *char buffer* é preenchida, letra a letra, com a mensagem ”Olá Mundo dos *Clusters*”. Se não for definida como compartilhada não haverá como duas ou mais JVMs participarem do preenchimento desta variável.

O mesmo acontece com a variável *loopCounter*, responsável por controlar em qual posição do *array* de tipo *char* as letras da variável “*string message*” serão gravadas. Se a variável *loopCounter* não for definida para acesso compartilhado, duas ou mais JVMs trabalhando em conjunto não conseguirão fazer o trabalho de preenchimento da variável *buffer* em menos tempo, havendo dessa forma uma sobre-escrita de caracteres na mesma posição do *array*. Isso acontece porque a variável *loopCounter* indexa as posições do *array* onde serão gravados os caracteres, sendo necessário que a mesma seja atualizada pela segunda JVM que faz parte do sistema. Portanto, é de fundamental importância que a variável *loopcounter* seja definida para acesso compartilhado.

As classes da aplicação são instrumentadas pela subseção *instrumented-classes* da seção *Application*, apresentada na listagem 5.1.7. A instrumentação de *bytecode* é o mecanismo essencial por trás da transparência do *Terracotta*. As classes JAVA são compiladas abaixo das instruções de *bytecode* que devem ser carregadas pela JVM antes que as novas instâncias destas classes sejam criadas. Este processo de carregamento de *bytecode* propicia ao *Terracotta* a possibilidade de inspecionar e manipular o *bytecode* das classes antes de serem usadas pela aplicação.

```
<instrumented-classes>
<include>
<class-expression>OlaMundodosClusters</class-expression>
</include>
</instrumented-classes>
```

Listagem 5.1.7: Subseção Instrumented-Classes - Exemplo Olá Mundo

O *Terracotta* pode ser configurado para instrumentar todas ou somente um subconjunto das classes carregadas na JVM. Instrumentar todas as classes assegura que o *Terracotta* visualizará tudo que a lógica da aplicação faz na memória em tempo de execução.

No exemplo apresentado, todas as classes foram instrumentadas, mas em casos reais e aplicações reais geralmente com muitas classes é interessante por razões de performance, instrumentar somente as classes que irão interagir com os objetos compartilhados. Entretanto, classes que não forem instrumentadas não serão visualizadas pelo *Terracotta* e qualquer modificação feita aos objetos compartilhados de uma classe que não foi instrumentada não será refletida no *cluster*.

O código de uma classe pode manipular objetos compartilhados mesmo que nenhuma instância desta classe tenha sido criada no *cluster*. Assim, deve-se instrumentar não somente as classes que possuem os objetos que serão compartilhados, mas também as classes que manipulam os objetos compartilhados.

O exemplo “OlaMundodosClusters” possui todas as classes instrumentadas e esta configuração pode ser modificada a partir das entradas *include* para instrumentar uma classe e *exclude* para evitar que uma classe seja instrumentada.

A última subseção da seção *Application* do arquivo de configuração, seção *locks*,

mostrada na listagem 5.1.8, é onde se determina os *locks* das variáveis compartilhadas (objetos compartilhados).

```
<locks>
<autolock>
<lock-level>write</lock-level>
<method-expression>
void OlaMundodosClusters.main(..)
</method-expression>
</autolock>
</locks>
</dso>
</application>
```

Listagem 5.1.8: Subseção *locks* - Exemplo Olá Mundo

O arquivo de configuração do exemplo determina o *lock* de escrita (*write*) para todos os métodos das classes instrumentadas. A configuração do exemplo informa ao *Terracotta* que o *lock* compartilhado deve ser aplicado ao método *main* do código OlaMundodosClusters. Os dois pontos na parte dos parâmetros indicam que o métodos chamados *main* podem ser aceitos com zero ou mais argumentos e podem ser de qualquer tipo. O exemplo possui apenas um método *main*.

As configurações possíveis para o nível de *lock* (*lock-level*) são:

- *write*: São *locks* de exclusão mútua que agem como os *locks* do JAVA. Neste caso, é garantido que somente uma *thread* em todo o *cluster* pode adquirir o *lock* por vez em um determinado tempo;
- *synchronous-write*: Este *lock* determina que a *thread* que mantém o *lock* não vai liberar este *lock* até as modificações feitas durante este *lock* forem totalmente aplicadas e enviadas para o servidor *Terracotta*;
- *read*: Permite à múltiplas *threads* leitoras adquirirem o *lock* de uma só vez, mas não é permitido a nenhuma destas *threads* realizar modificações nos objetos compartilhados enquanto estiverem mantendo o *lock* de leitura (*read*). Nenhuma *thread* irá adquirir um *lock* de escrita (*write*) se qualquer uma das *threads* mantiver um *lock* de leitura (*read*). Nenhuma *thread* vai adquirir um *lock* de leitura (*read*) enquanto qualquer uma das threads ficar mantendo um *lock* de escrita (*write*). Se uma *thread* tentar fazer modificações enquanto mantém um *lock* de leitura (*read*), o *Terracotta* irá gerar uma exceção. Segundo (TERRACOTTA, 2008), este nível de *lock* apresenta vantagens de performance em relação aos outros níveis de *lock* quando se tem múltiplas *threads* concorrentes executando que não modificam os dados compartilhados;
- *concurrent*: Este nível de *lock* não protege as seções críticas de código pois os *locks* são sempre concedidos. Não há garantia sobre a ordem nas quais as transações efetuadas pelas *threads* sob os *locks* concorrentes (*concurrent*) são aplicadas. Também não há garantia que outras *threads* modifiquem e apliquem as modificações sobre os dados compartilhados no *Heap*, enquanto um *lock* estiver ativo.

Na seção 5.1.5.2 é mostrado o exemplo desta seção em funcionamento, com intuito de avaliar as capacidades do *Terracotta* e verificar as suas funcionalidades.

5.1.5.2 Terracotta em Ação

Esta seção descreve os testes realizados com o *software Terracotta* em plataforma *Windows*. Posteriormente, no seu funcionamento, integrado ao módulo de execução do Ambiente D-GM, pode-se optar também por utilizar plataforma *Linux*.

As etapas necessárias à instalação do *Terracotta* são:

A Fazer o *download* e instalar os arquivos necessários ao seu funcionamento:

1. *download* do arquivo Terracotta-3.1.1-installer.jar em `http://www.terracotta.org/dl/oss-download-catalog`;
2. *download* do JAVA *Development Kit* ou JDK, pois são necessários o compilador e as ferramentas de desenvolvimento JAVA que não estão presentes no *JAVA Runtime Environment* ou JRE que é a alternativa mais comum para se obter o JAVA. O JDK pode ser baixado em `http://java.sun.com/javase/downloads/index.jsp`;
3. Após o *download* dos arquivos, deve-se proceder com a sua instalação. Essa instalação deve ser feita a partir do *prompt* de comando e executando a seguinte linha de comando "java -jar terracotta-3.1.1-installer.jar" dentro da pasta onde foi feito o *download*. Isto requer o JDK já instalado e as variáveis de ambiente para o compilador JAVA corretamente configuradas.

B Com a etapa anterior concluída, é necessário configurar as variáveis de ambiente para o *Terracotta*, geralmente a pasta de instalação é `C:\Arquivos_de_programas\terracotta` e a variável de ambiente *path* do *Windows* deve ser atualizada com o seguinte caminho `C:\Arquivos_de_programas\terracotta\terracotta-3.1.1\bin`.

C Após estas configurações, deve-se ir até a pasta onde o código "OlaMundodosClusters.java" foi criado e compilar este código com o comando "javac OlaMundodosClusters". Dessa forma, um arquivo chamado OlaMundodosClusters.class será criado.

D A inicialização do servidor *Terracotta* deve ser feita, no *prompt*, pelo seguinte comando "start-tc-server -f tc-config.xml", especificando o arquivo de configuração *tc-config.xml* detalhado na seção 5.1.5.1. Com o servidor em execução, as instâncias da JVM podem ser disparadas para a execução do código do exemplo "OlaMundodosClusters".

E O código começa a executar a partir da linha de comando "dso-java OlaMundodosClusters" na pasta onde o arquivo OlaMundodosClusters.class foi criado, e em uma outra janela do prompt de comando. Tem-se então uma instância da JVM executando o código do exemplo, e o resultado é mostrado na figura 5.1:

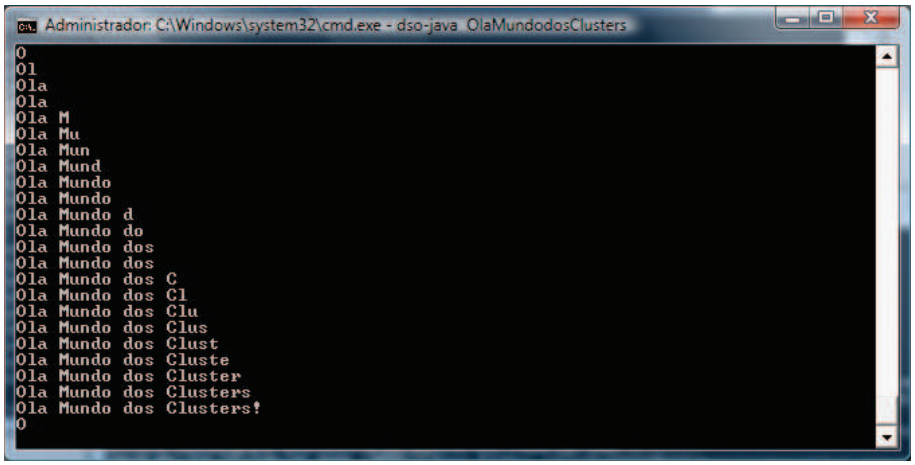

Figura 5.1: Execução com uma instância da JVM

F Ao abrir outro *prompt* de comando e acionar o comando "dso-java OlaMundodosClusters" haverá outra instância da JVM trabalhando para a mesma aplicação, ou seja, duas JVMs irão trabalhar em conjunto para realizar a tarefa mais rapidamente. Os arquivos "start-tc-server.bat" e "dso-java.bat" são arquivos tipo *scripts* de ajuda do *Terracotta*, eles adicionam argumentos, automaticamente, na chamada dos arquivos executáveis simplificando o uso das aplicações com o *Terracotta*. O resultado após o acionamento da segunda JVM é mostrado na figura 5.2:

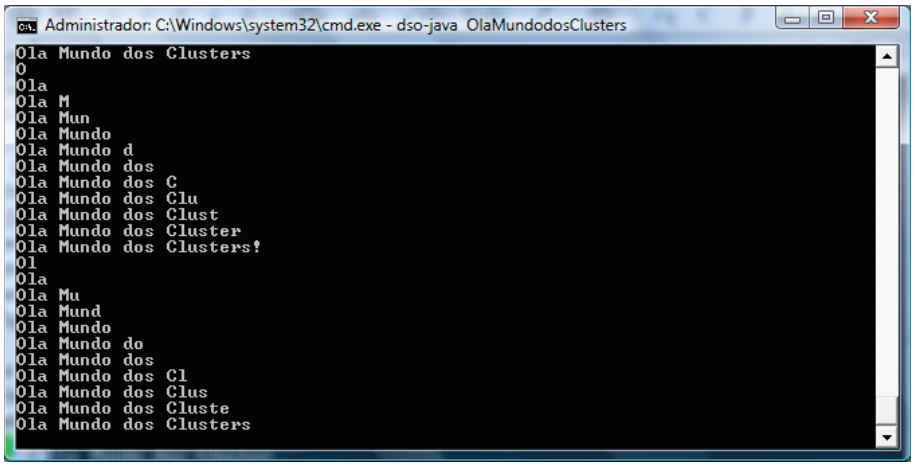

Figura 5.2: Execução com duas instâncias da JVM

Pode-se observar que a execução com duas instâncias da JVM possui um desempenho mais favorável que a execução com apenas uma instância da JVM, reduzindo dessa forma o tempo para preencher a variável *buffer* com a mensagem "Olá Mundo dos Clusters", verificando as funcionalidades do *Terracotta* no tocante ao gerenciamento de duas ou mais JVMs reunindo esforços para execução de aplicações. A partir da verificação do tempo para preenchimento do *buffer* com uma e duas JVMs, concluiu-se que o uso do *Terracotta* nesta aplicação representa um ganho de 40% em tempo de execução. Isto pode ser verificado na tabela 5.1.

Nº de JVMs	*Execuções (x)*	*Tempo Médio de Execução (s)*	*Ganho (%)*
1 s/ *Terracotta*	20	2,39	0
2 c/ *Terracotta*	20	1,44	40

Tabela 5.1: Tempos de execução da aplicação "OlaMundodosClusters"

Os resultados iniciais apresentados neste exemplo, evidenciam que as funcionalidades do *Terracotta* podem vir a contribuir favoravelmente no que diz respeito ao desempenho de execução de aplicações, além das vantagens já citadas anteriormente.

5.2 ShareD-GM: Arquitetura de Memória Distribuída para o Ambiente D-GM

A modelagem da arquitetura ShareD-GM engloba as classes e objetos que fazem parte do módulo de execução VirD-GM, devido a isso as principais classes descritas na seção 2.2.1 serão partes integrantes do referido modelo. O objeto que implementa a memória do Ambiente D-GM deve ser compartilhado pela memória compartilhada distribuída implementada pelo *Terracotta*, evitando que uma cópia deste objeto seja passada como parâmetro para os nodos de um *cluster* quando da execução distribuída, como acontece na implementação atual. Este fator, além de melhorar a dinâmica de execução do VirD-GM por não precisar realizar a transferência de uma cópia do objeto memória do Ambiente D-GM para os nodos por passagem de parâmetros, possivelmente aumentará o desempenho pois diminuirá as *serializações* e *deserializações* de objetos.

5.2.1 Principais Classes do Módulo de Execução VirD-GM

As principais classes do módulo de execução VirD-GM são:

- VirdLauncher: Responsável pela inicialização e escalonamento das tarefas;
- VirdExec: Responsável por enviar os processos para execução nos nodos do *cluster*;
- VirdLoader: Responsável pela interpretação dos arquivos descritores de processos e memória enviados pelos editores do módulo de programação visual VPE-GM, se divide em duas classes, VirdProcLoader para interpretação de processos e VirdMemLoader para interpretação da memória.

Para a concretização da modelagem, estas classes devem ser integradas com os módulos do ShareD-GM em uma arquitetura representando o Ambiente D-GM.

5.2.2 Principais Módulos ShareD-GM

Os principais módulos da memória compartilhada distribuída ShareD-GM são:

- Servidor *Terracotta*: Responsável por gerenciar os objetos compartilhados e disponibilizá-los para os clientes em execução;
- Classes instrumentadas: Define as classes que terão carregadas em seu bytecode no momento da execução do código, as instruções de interceptação de acessos ao *Heap* do *Terracotta*;

- Objetos Compartilhados: Define quais os objetos da aplicação serão compartilhados para todas as classes da aplicação JAVA;
- *Locks*: Define qual o nível de *lock* dos objetos compartilhados pelo servidor *Terracotta*, na maioria dos casos esse nível é o de escrita (*write*) onde somente um processo pode escrever em um objeto compartilhado de cada vez.

A arquitetura da ShareD-GM é apresentada na figura 5.3.

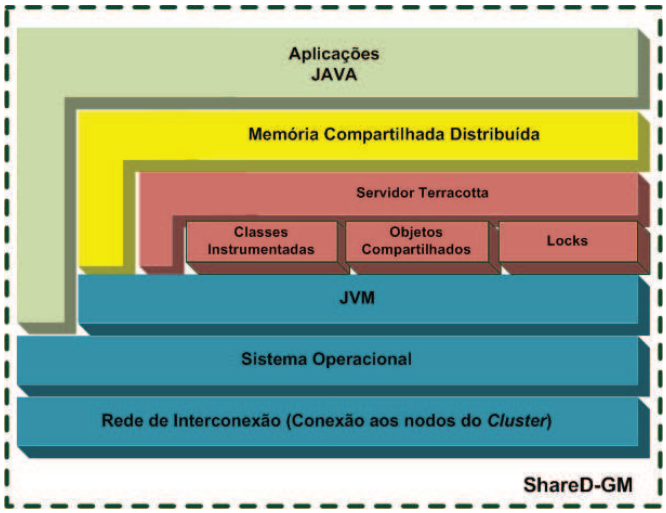

Figura 5.3: Arquitetura ShareD-GM

5.2.3 Modelagem e Implementação da ShareD-GM

A figura 5.4 mostra a arquitetura do Ambiente D-GM integrada à arquitetura da memória compartilhada distribuída ShareD-GM.

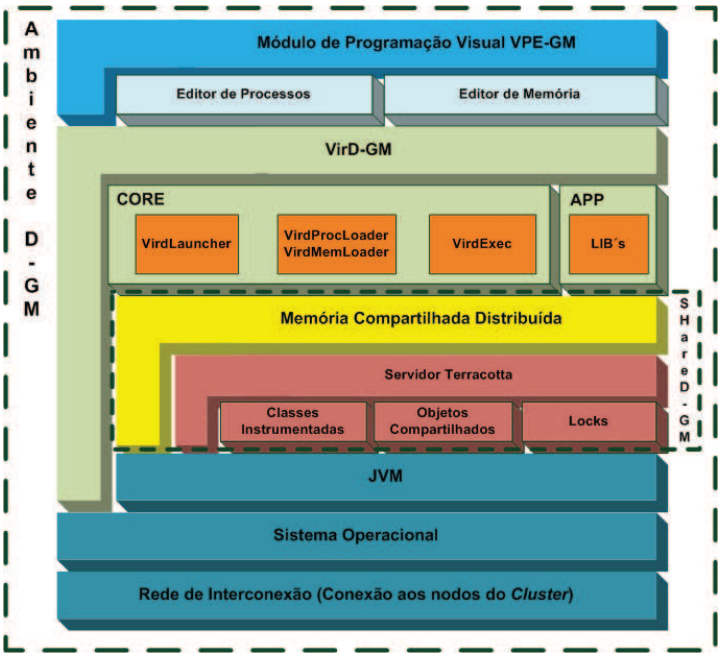

Figura 5.4: Arquitetura de Integração Ambiente D-GM/ShareD-GM

Pode-se visualizar, na figura 5.4, a modelagem realizada através da integração das classes do módulo de execução VirD-GM do Ambiente D-GM com os módulos da memória compartilhada distribuída ShareD-GM.A integração proposta viabilizará que objetos criados pelo módulo de execução VirD-GM sejam compartilhados entre os nodos que fazem parte do sistema de execução em um *cluster*.

A comunicação feita pelo módulo VirD-GM com os nodos disponíveis do cluster, para o envio de tarefas que precisam ser executadas de forma paralela e/ou distribuída, permanece inalterada até o momento. Esse fato, explica a ligação do módulo VirD-GM com o sistema operacional, mesmo após a integração com os módulos da memória compartilhada distribuída ShareD-GM.

Essa integração será alcançada através da configuração do arquivo *tc-config.xml* conforme as seções e subseções apresentadas nas listagens seguintes.

A seção *servers*, listagem 5.2.1, traz a configuração de armazenamento temporário, definindo que o servidor não precisa manter os dados da aplicação específica em execução com o *Terracotta* para uma próxima execução da mesma, também especifica a máquina do *cluster* cujo IP é $200.132.45.235$ como servidor e os locais padrão para armazenamento dos arquivos de dados e *logs*.

```
<servers>
<server host="200.132.45.235">
<data>%(user.home)/terracotta/server-data</data>
<logs>%(user.home)/terracotta/server-logs</logs>
<dso>
<persistence>
<mode>temporary-swap-only</mode>
</persistence>
</dso>
</server>
</servers>
```

Listagem 5.2.1: Seção *servers* - Arquivo de Configuração ShareD-GM

A seção *clients* também mantém a configuração padrão, mostrada na listagem 5.2.2, para os arquivos de *logs* gerados pelos clientes.

```
<clients>
<logs>%(user.home)/terracotta/client-logs</logs>
</clients>
```

Listagem 5.2.2: Seção *clients* - Arquivo de Configuração ShareD-GM

A seção *application*, subseção *root/field-name*, define como variável compartilhada o objeto *VirdLauncher.virdMemory* que implementa a memória distribuída do módulo de execução VirD-GM do Ambiente D-GM. Esta configuração é apresentada na listagem 5.2.3.

```
<application>
<dso>
<roots>
<root>
<field-name>g3pd.virdgm.core.VirdLauncher.virdMemory</field-name
   >
</root>
</roots>
```

Listagem 5.2.3: Seção *application* - Subseção *root/field-name* - Arquivo de Configuração ShareD-GM

A seção de classes instrumentadas (*instrumented-classes*), mostrada na listagem 5.2.4, especifica que as classes *VirdMemory*, responsável por implementar a memória do módulo de execução, *VirdLauncher*, responsável pela inicialização e disparo do VirD-GM, o pacote de classes definido pela máscara *g3pd.virdgm.types.**, cuja função é conter os tipos de dados suportados pela memória do módulo de execução e o pacote de classes *g3pd.virdgm.apps.**, que implementa a biblioteca das aplicações passíveis de execução pelo Ambiente D-GM, poderão ter os seus *bytecodes* inspecionados e manipulados pelo *Terracotta*. Esta configuração foi feita, observando-se quais classes possuem métodos que acessam o objeto compartilhado que implementa a memória do módulo de execução VirD-GM. Se somente a classe *VirdLauncher* que cria o objeto *VirdLauncher.virdMemory* implementando a memória distribuída do Ambiente D-GM fosse instrumentada, as outras classes que compõe o módulo de execução e fazem acesso a memória do VirD-GM não encontrariam o objeto da memória (*VirdLauncher.virdMemory*) compartilhado pelo *Terracotta*.

```
<instrumented-classes>
<include>
<class-expression>g3pd.virdgm.core.VirdMemory</class-expression>
</include>
<include>
<class-expression>g3pd.virdgm.core.VirdLauncher</class-
   expression>
</include>
<include>
<class-expression>g3pd.virdgm.types.*</class-expression>
</include>
<include>
<class-expression>g3pd.virdgm.apps.*</class-expression>
</include>
</instrumented-classes>
```

Listagem 5.2.4: Subseção *instrumented-classes* - Arquivo de Configuração ShareD-GM

A configuração da seção *locks* na listagem 5.2.5, especifica quais os métodos que atuam sobre os objetos compartilhados e que tipo de bloqueio eles utilizarão para acessar estes objetos compartilhados. As expressões *void g3pd.virdgm.core.VirdMemory.updateMemory(Object,Integer)* e *void g3pd.virdgm.core.VirdMemory.writeMemory(Object,Integer)* determinam que estes

métodos, que possuem a função de escrita em memória no código do módulo de execução VirD-GM, tenham um bloqueio de escrita síncrona. Este tipo de bloqueio emula o mecanismo de escrita em memória do modelo GM, implementado no VirD-GM, onde os métodos podem escrever neste objeto um de cada vez e o acesso ao objeto só estará liberado após as mudanças em memória serem efetivadas.

```
<locks>
<autolock>
<lock-level>synchronous-write</lock-level>
<method-expression>
void g3pd.virdgm.core.VirdMemory.updateMemory(Object,Integer)
</method-expression>
</autolock>
<autolock>
<lock-level>synchronous-write</lock-level>
<method-expression>
g3pd.virdgm.core.VirdMemory.writeMemory(Object,Integer)
</method-expression>
</autolock>
</locks>
</dso>
</application>
```

Listagem 5.2.5: Subseção *locks* - Arquivo de Configuração ShareD-GM

O arquivo de configuração completo para a integração ShareD-GM entre o *Terracotta* e o Ambiente D-GM, pode ser visto no Anexo B.

Além do arquivo de configuração *tc-config.xml*, completam a integração a instalação dos arquivos do *Terracotta* na máquina que irá servir de servidor e nas outras que serão os clientes. O código JAVA do módulo VirD-GM também deverá ser modificado, pois variáveis (objetos) que antes eram passados como parâmetros para os clientes, modificados e retornados para o servidor, agora são compartilhados de uma forma global entre todos os métodos das classes.

A seguinte linha de código, apresentada na listagem 5.2.6, faz parte da classe *VirdLauncher* do módulo de execução VirD-GM e é responsável por enviar aos clientes (nodos) para execução, os parâmetros e uma cópia do objeto *virdMemory*.

```
virdExec.send(actionAttr, valueAttr, inputPosAttr, outputPosAttr
   , iterator, virdMemory, host, port);
```

Listagem 5.2.6: Linha de Código Classe VirdLauncher - Módulo de Execução

Com a introdução do *Terracotta*, o objeto *virdMemory*, que implementa a memória distribuída do módulo de execução VirD-GM não seria mais passada como parâmetro, se tornando uma variável compartilhada disponível para todos os clientes (nodos) do *cluster*.

Diferentemente do exemplo mostrado na seção 5.1.5.1, o módulo de execução do Ambiente D-GM (VirD-GM) possui como sistema base atualmente a plataforma Linux, tornando-se necessário a preparação do sistema para a execução de aplicações JAVA com o *Terracotta*. Esta preparação é descrita no anexo C.

6 SHARED-GM: APLICAÇÕES DE TESTE

Este capítulo descreve o comportamento observado com as aplicações de teste desenvolvidas para avaliação da ShareD-GM. O principal objetivo do uso destas aplicações é a exploração das funcionalidades do módulo de execução VirD-GM integrado ao sistema de memória compartilhada distribuída *Terracotta* no que diz respeito à utilização da memória compartilhada durante o disparo, execução e gerência de aplicações distribuídas e/ou paralelas desenvolvidas utilizando o módulo de desenvolvimento do Ambiente D-GM (VPE-GM).

As aplicações, do tipo sintética, não possuem otimizações quanto ao desempenho, suas modelagens tem como objetivo verificar o correto comportamento do mecanismo de memória compartilhada distribuída (DSM) integrado ao módulo de execução VirD-GM, bem como avaliar as melhorias na execução das aplicações na referida integração.

Considerou-se como aplicações de testes e validação da ShareD-GM o algoritmo de *Smith-Waterman* (WATERMAN; SMITH, 1981) e o algoritmo do método de *Jacobi* (BARRETT et al., 1994), dois algoritmos bem conhecidos e utilizados para estes tipos de testes.

Salienta-se que, para os testes descritos neste capítulo, foram realizadas ao todo 20 medições para cada tipo de teste, observando-se uma pequena flutuação nos resultados obtidos para as repetições dos testes. Devido a isso, os valores apresentados nas Seções 6.1.2 e 6.2.2 correspondem a média obtida a partir da medição dos tempos de execução de cada aplicação. Considerou-se a execução sequencial, a execução em 4 processadores e a execução em 8 processadores para o algoritmo de *Smith-Waterman* e a execução em 8 e 16 processadores, além da execução sequencial, para o algoritmo do método de *Jacobi*. Observa-se em cada máquina a existência de 2 processadores. A avaliação dos resultados ocorreu em um cluster que disponibiliza 8 unidades de processamento na sua totalidade, possuindo um conjunto de nodos homogêneos quanto a sua capacidade de processamento. Cada nodo consiste de um processador Intel *Dual Core* 2140 de 1600MHz, com 2048 MBytes de memória RAM.

6.1 Algoritmo de *Smith-Waterman*

O algoritmo de *Smith-Waterman* tem como principal função a procura de similaridades entre subsequências de duas sequências comparadas. Estas similaridades podem indicar evidências de homologias (MENG; CHAUDHARY, 2009) por meio de buscas em bancos de dados de sequências já conhecidas. Este algoritmo é utilizado principalmente para identificação de sequências de DNA desconhecidas, a partir de comparações com

sequências de DNA já existentes e armazenadas em bases de dados.

6.1.1 Metodologia de Implementação

Neste algoritmo, adota-se o método da programação dinâmica para realizar a comparação entre uma sequência desconhecida e as sequências presentes em um banco de dados, atribuindo uma nota a cada comparação. Dessa forma, uma matriz é criada dinamicamente para cada comparação. Considerando-se duas sequências $S = s_1 s_2 ... s_m$ e $T = t_1 t_2 ... t_n$ de tamanhos m e n respectivamente, o algoritmo de *Smith-Waterman* encontra a melhor nota para essa comparação através da construção de uma matriz D de tamanho $(m+1) \times (n+1)$ onde cada entrada dessa matriz indexada pelas variáveis i e j é dada pelas relações recorrentes representadas pelas expressões 6.1 e 6.2 (LIAO; YIN; CHENG, 2004):

$$\begin{aligned} &\text{Para } 0 \leq i \leq m, 0 \leq j \leq n \\ &\qquad \text{Faça } D_{i0} = D_{0j} = 0 \end{aligned} \tag{6.1}$$

$$\begin{aligned} &\text{Para } 1 \leq i \leq m, 1 \leq j \leq n \\ &\qquad \text{Faça} \end{aligned} \tag{6.2}$$

$$D_{ij} = MAX \begin{cases} 0, \\ D_{(i-1)(j-1)} + & Sbt(t_i, t_j) \\ D_{(i-1)(j)} & - \text{ custo de inserção de lacuna} \\ D_{(i)(j-1)} & - \text{ custo de inserção de lacuna} \end{cases}$$

Além da matriz principal, uma matriz de substituição também é desenvolvida com o intuito de avaliar se os caracteres das sequências são iguais ou diferentes, para a construção dessa matriz considera-se a seguinte definição:

$$Sbt(s_i, t_j) = \begin{cases} +2 & se \quad (s_i = t_j), \\ -1 & \text{senão} \end{cases}$$

O custo de inserção de lacuna deve ser adotado como 2, o mesmo se refere ao custo de se encontrar caracteres diferentes na mesma posição das sequências comparadas. Dessa forma, um caractere do tipo "-" deve ser inserido nas sequências resultantes.

Finalizada a etapa de construção da matriz, o próximo passo é achar o maior valor presente nesta matriz e realizar o caminho de volta a partir deste valor máximo, seguindo os ponteiros armazenados para cada valor da matriz (*Backtracking Pointers*).

Os referidos ponteiros tem a função de indicar a posição do próximo valor máximo considerando as três posições anteriores adjacentes a posição atual na matriz.

Considerando-se as sequências S = ACGTAC e T = CACGTTG a matriz construída é mostrada na figura 6.1, bem como o caminho percorrido para se alcançar o alinhamento das sequências.

		C	A	C	G	T	T	G
	0	0	0	0	0	0	0	0
A	0	0	2	0	0	0	0	0
C	0	2	0	4	2	0	0	0
G	0	0	1	2	6	4	2	2
T	0	0	0	0	4	8	6	4
A	0	0	2	0	2	6	7	5
C	0	2	0	4	2	4	5	6

Figura 6.1: Matriz gerada pelo algoritmo SW e o caminho percorrido.

Assim, o alinhamento obtido é mostrado na figura 6.2.

```
- A C G T
C A C G T
```

Figura 6.2: Alinhamento obtido.

O referido algoritmo foi modelado no ambiente de programação visual VPE-GM, figura 6.3, disciplinando-o de acordo com as abstrações do modelo GM. Utilizou-se um construtor Iterativo Paralelo cuja função é executar diversas vezes o mesmo algoritmo em paralelo, modificando apenas os seus parâmetros de entrada e gravando o resultado em posições diferentes de memória.

A estratégia de paralelização adotada é regida pelo paralelismo síncrono, característica inerente ao modelo GM. Nesta abordagem cada par de sequências à ser comparada é responsabilidade de um processo, o qual pode ser executado em paralelo em máquinas diferentes ou concorrentemente em uma mesma máquina.

Segundo a modelagem, a aplicação deve fazer 1.000 comparações de sequências com 32 caracteres cada uma e colocar o resultado das comparações a partir da posição 1 da memória. Todos essas comparações podem ser feitas paralelamente.

Após as comparações, o próximo processo sequencial busca o resultados da sequência com maior similaridade, exibindo-o e encerrando a execução da aplicação.

A referida modelagem realizada no VPE-GM é mostrada na figura 6.3 e apresenta um processo chamado “SW”, referente a uma função responsável pela execução do algoritmo de Smith-Waterman, cuja implementação foi realizada de acordo com as instruções para construção do algoritmo contidas em (WATERMAN; SMITH, 1981). A função “SW” foi adicionada na biblioteca de funções do módulo de execução para dar suporte a modelagem.

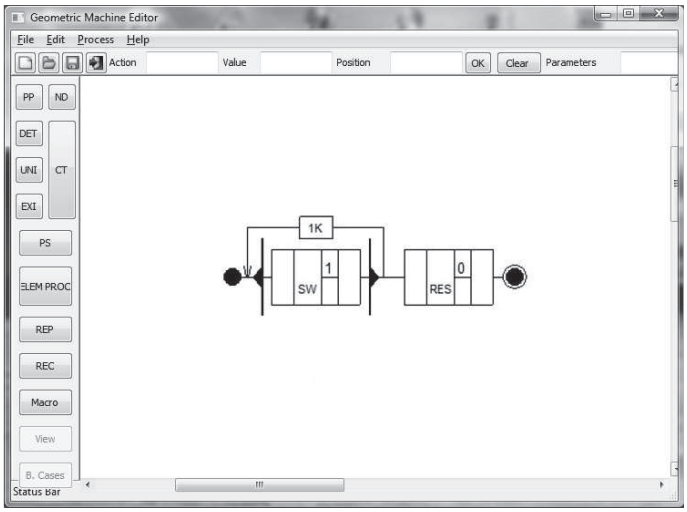

Figura 6.3: Modelagem do Algoritmo de Smith-Waterman

Os arquivos XML gerados pelo módulo de programação visual VPE-GM de acordo com a modelagem da figura 6.3, são mostrados nas figuras 6.4 e 6.5.

```
<process repr="env" tipo="seq" parametro="" pos="1001, 0">
<process repr="terminicio" x="1235.0" y="1003.0" cor="BLACK"></process>
<process repr="env" tipo="seq" parametro="" pos="1001, 0">
<process repr="iterativo" tipo="iterativo" range="1:1001:1" acao="parfor" x="1294.0" y="1003.0"  cor="BLACK">
<process repr="conselem" acao="SW" pos="1000" parametro="1" x="1294.0" y="1003.0" cor="BLACK"></process>
</process>
<process repr="conselem" acao="RES" pos="1" parametro="0" x="1384.0" y="1003.0" cor="BLACK"></process>
</process>
<process repr="termfim" x="1445.0" y="1003.0" cor="BLACK"></process>
</process>
```

Figura 6.4: Arquivo XML descritor de processos.

```
<memoria>
<posicao dimensao="1" tamanho="1001"></posicao>
<valores> String, String, String, String, String, String, String,
String, String, String, ... </valores>
<dados> CACAGGCHCHTTAHAHHGGCTHCCHTTHCHT,  CTCHGTGCACTHTTTGHAACTCTACCCGAAAHTHCCGHTC,
CACHAGTCAACAAGHTTGAHACAHCGGHGACCGGHAHTCH,  GTAGCHGTCCCATAACGACCTTTHCGTATHTHGTGTGTGA,
CCHGHHTGCCGTAHCAHCACAAHTAGAGTTTCHHCAHATT,  CCHCGTCGGTCTCHATHTCHGGACCHTTGHGAHAGCCATG,
TTAGHTGGAGGHATHHATHATTCCAGTTCGGTGCCACHAA,  AHTCCCTTTCGTAGTHCHTHACHTAHCCAAHTAAATTGCT,
CTGAGACTTHCGGACHACCHAGTGHGHTHGCCATATTHGT,
AHGGCGCHTGCHAHCHATHAHGHHHCHHTGTHTCCGGHAH, ... , 1 </dados>
</memoria>
```

Figura 6.5: Arquivo XML descritor da memória (reduzido).

6.1.2 Resultados Obtidos

A partir da criação dos arquivos XML, tornou-se possível a execução da aplicação tanto com a integração ShareD-GM quanto com a implementação anterior do módulo de execução VirD-GM, tabelas 6.1 e 6.2 respectivamente, com o intuito de avaliar as melhorias na execução da aplicação e realizar cálculos de *speedup* relativo.

Tabela 6.1: Tempos de execução do algoritmo de Smith-Waterman (ShareD-GM)

Nº de Proc.	*Alocação de Nodos*	*T. M. de Execução(s)*	*Speedup Relativo*
8	N1, N2, N3, N4	18,345	3,20
4	N1, N2	24,805	2,37
1	N1	58,721	-

Tabela 6.2: Tempos de execução do algoritmo de Smith-Waterman (VirD-GM)

Nº de Proc.	*Alocação de Nodos*	*T. M. de Execução(s)*	*Speedup Relativo*
8	N1, N2, N3, N4	74,623	1,99
4	N1, N2	76,761	1,93
1	N1	148,341	-

As tabelas 6.1 e 6.2 fazem uma síntese das execuções realizadas com as duas implementações, VirD-GM e ShareD-GM, evidenciando um menor tempo de execução devido ao uso do sistema de memória compartilhada distribuída *Terracotta* e dessa forma, favorecendo a arquitetura ShareD-GM. Deve-se salientar que os resultados são relativos a execução do algoritmo com 1, 4 e 8 processadores, considerando 1000 comparações de sequências em cada execução.

6.2 Algoritmo do Método de *Jacobi*

O estudo de caso descrito nesta seção para avaliação da ShareD-GM considera o algoritmo que implementa o método de *Jacobi* (BARRETT et al., 1994). O método de *Jacobi* é um método iterativo para resolução de grandes e esparsos sistemas lineares de equações algébricas (SELAs) (MEHMOOD; CROWCROFT, 2005). Um método iterativo busca a utilização de técnicas para a realização de sucessivas aproximações cujo objetivo é a obtenção de soluções mais precisas em cada passo de iteração, sendo que o número de passos é determinante para a exatidão dos resultados (BARRETT et al., 1994).

A resolução de SELAs pelo método de *Jacobi* é frequentemente aplicada em problemas de domínio discreto existentes principalmente em áreas como ciência e engenharia. Pode-se citar como exemplo a modelagem matemática de sistemas físicos como o sistema climático (MEHMOOD; CROWCROFT, 2005).

6.2.1 Metodologia de Implementação

Na expressão matricial de um SELA dada por $A\vec{x} = \vec{b}$, tem-se A como uma matriz quadrada de ordem n que representa os coeficientes a_{ij} deste sistema, $\vec{x}$ e $\vec{b}$ como vetores coluna de n elementos que representam o vetor das variáveis (referente às incógnitas) e o vetor de termos independentes (valores à direita da igualdade) do sistema, respectivamente. Segundo este método, sempre que $1 \leq i, j \leq n$, o i-ésimo componente do vetor independente $\vec{b}$ pode ser expresso pela eq. (6.3):

$$\sum_{j=0}^{n} a_{i,j} * x_j = b_i \tag{6.3}$$

Isolando o componente do vetor $\vec{x}$ na eq. (6.3), obtém-se o valor de x_i dado na eq. (6.5):

$$x_i = \left(b_i - \sum_{i \neq j} a_{i,j} * x_j \right) / a_{i,i} \tag{6.4}$$

E de forma iterativa, obtém-se a seguinte expressão:

$$x_i^k = \left(b_i - \sum_{i \neq j} a_{i,j} * x_j^{k-1} \right) / a_{i,i}, \tag{6.5}$$

onde k indica o número de iterações na eq. (6.5).

Dessa forma, o método de *Jacobi* começa como uma solução aproximada para o sistema de equações, definindo um valor inicial aproximado para o vetor de incógnitas e vai aumentando a exatidão desta solução a cada passo de iteração. Para a execução das iterações o método de *Jacobi* considera o algoritmo a seguir:

```
MetodoJacobi(A, x, b)
inicializa x^(0)
   for k = 1 to N
       for i = 1 to n
       x̄_i = 0
       for j = 1 to n
          x̄_i = x̄_i + a_{i,j} * x_j^{k-1}
          x̄_i = (b_i - x_i)/a_{i,i}
       x^(0) = x̄
       verifica convergência
```

O referido algoritmo foi modelado e implementado conforme os padrões de execução do Ambiente D-GM, sendo necessário para isto a realização da modelagem no módulo de programação visual VPE-GM.

A modelagem no VPE-GM é responsável por disciplinar o algoritmo de acordo com as abstrações do modelo GM e também por permitir a definição de uma estratégia de paralelização, regida pelo paralelismo síncrono, dos processos que definem o algoritmo.

A estratégia adotada foi a divisão em blocos da matriz A de coeficientes e dos vetores de variáveis (incógnitas) e termos independentes, com o objetivo de delegar o cálculo de cada bloco de igual tamanho da matriz para um nodo diferente do *cluster* e consequentemente para processos diferentes, propiciando dessa forma a execução paralela dos cálculos conjuntamente com o paralelismo de dados proporcionado pelo sistema DSM.

A modelagem do algoritmo na VPE-GM é mostrada na figura 6.6 e conforme a estratégia escolhida, a referida modelagem faz uso de Processos Elementares, dos construtores Produto Sequencial, Iterativo Sequencial e Paralelo.

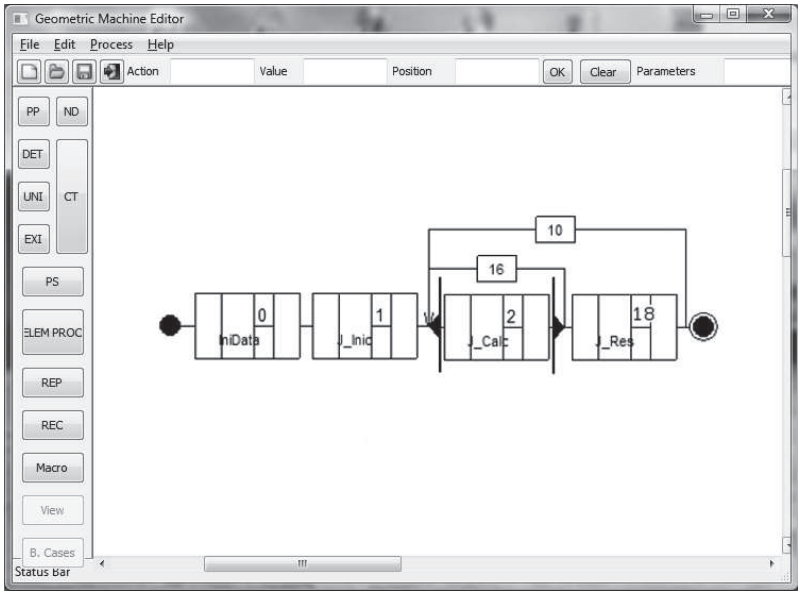

Figura 6.6: Algoritmo de *Jacobi* no VPE-GM

O início do fluxo de execução desta modelagem contém um processo responsável pelo carregamento de valores nas matrizes (IniData) e outro processo com a função de inicializar o vetor com os valores estimados de resultados referentes as incógnitas (JInic), a partir do qual serão realizadas as aproximações.

A parte responsável pelos cálculos do algoritmo tem seu trabalho dividido em dezesseis processos (JCalc), controlados pelo construtor Iterativo Paralelo que determina a execução paralela do mesmo processo apenas variando os parâmetros de entrada. Cada processo acessa um bloco de igual tamanho,definidos a partir dos parâmetros de entrada do processo JCalc, de valores da matriz.

Ao final da execução dos processos, os resultados são reunidos e armazenados no vetor de variáveis (incógnitas) por outro processo (JRes). Terminado este processo, o fluxo de execução segue para executar mais uma vez a parte responsável pelos cálculos com o objetivo de aproximar os resultados com mais uma iteração. Esta modelagem considera dez iterações para aproximação dos resultados.

A modelagem no VPE-GM, mais especificamente no Editor de Processos, gera o arquivo XML descritor de processos, mostrado na figura 6.7.

No Editor de Memória, constrói-se o arquivo XML descritor da memória, mostrado na figura 6.8. O conteúdo do arquivo XML descritor da memória refere-se a uma matriz de tamanho 1280×1280 e foi reduzido para fins de exibição.

Deve-se ressaltar que um arquivo XML descritor da memória para a matriz de tamanho 1920×1920 também foi gerado para a realização dos testes.

```
<save arq="/home/usuario/argvird/JacobiProc" pos="6, 0">
<process repr="env" tipo="seq" parametro="" pos="6, 0">
<process repr="terminicio" x="1072.0" y="1133.0" cor="BLACK"></process>
<process repr="env" tipo="seq" parametro="" pos="6, 0">
<process repr="conselem" acao="IniData" pos="0" parametro="1280" x="1131.0" y="1133.0" cor="BLACK"></process>
<process repr="conselem" acao="J_Inic" pos="18" parametro="0,1" x="1131.0" y="1133.0" cor="BLACK"></process>
<process repr="iterativo" tipo="iterativo"  range="1:11:1" acao="seqfor" cor="BLACK">
<process repr="env" tipo="seq" parametro="" pos="0,18">
<process repr="iterativo" tipo="iterativo" range="0:16:1" acao="parfor" cor="BLACK">
<process repr="conselem" acao="J_Calc" pos="2" parametro="0,18" x="1221.0" y="1133.0" cor="BLACK"></process>
</process>
<process repr="conselem" acao="J_Res" pos="18" parametro="2,17" x="1311.0" y="1133.0" cor="BLACK"></process>
</process>
</process>
</process>
<process repr="termfim" x="1372.0" y="1133.0" cor="BLACK"></process>
</process>
</save>
```

Figura 6.7: Arquivo descritor de processos do algoritmo do método de *Jacobi*.

```
<memoria>
<posicac dimensac="2" tamanhc="1280"></posicac>
<valores>Float</valores>
<dados>
35,47,62,27,1,66,32,82,52,65,54,76,51,19,40,1,73,55,11,24,20,75,81,56,82,97,19,31,24,20,95,39,80,30,85,83,
57,33,8,85,27,57,79,84,63,10,94,6,66,88,15,91,82,100,100,86,17,13,3,29,46,79,95,89,36,76,5,15,96,36,56,17,
6,96,39,38,45,58,69,17,58,100,63,22,94,52,57,20,73,14,34,90,12,67,8,39,66,88,10,77,45,87,34,51,51,15,95,3,
,33,66,5,40,24,76,72,71,32,38,87,43,73,84,63,42,99,66,30,13,79,52,63,67,88,1,37,84,32,57,20,14,96,94,56,4
,70,34,34,22,71,49,83,50,57,20,51,46,31,49,25,38,68,85,11,3,44,12,4,73,12,28,65,88,15,85,10,14,59,18,93,6
,87,82,34,49,93,85,40,80,85,71,17,14,46,100,19,70,48,89,17,100,83,67,89,31,44,17,83,89,85,76,90,6,28,88,7
,40,30,13,24,89,83,37,1,65,7,1,32,93,76,70,56,21,19,60,19,6,14,67,90,30,15,46,2,8,51,5,8,70,17,85,5,13,31,
,30,79,90,69,52,29,20,28,86,87,10,88,69,33,12,86,71,86,6,33,31,85,47,48,76,61,95,89,85,17,74,39,17,86,12,
8,8,43,65,69,97,74,8,87,5,37,23,36,6,32,73,84,63,42,60,38,22,39,24,38,62,64,63,45,90,86,46,60,85,25,76,74,
,66,47,52,56,41,29,37,70,46,58,37,69,12,34,32,38,66,98,95,32,12,83,92,53,15,79,32,43,40,24,72,53,2,89,28,7
,54,48,47,12,49,74,33,96,65,92,59,35,67,67,31,89,51,9,52,75,74,91,68,17,40,26,92,39,46,73,90,69,94,16,18,7
7,45,75,-</dados>
<dados>
10,60,13,18,99,39,34,75,31,73,68,27,63,79,5,50,71,100,37,45,54,64,20,92,34,95,89,79,29,13,29,72,59,45,48,8
0,20,29,49,31,97,74,17,66,55,35,29,65,92,71,72,50,24,47,62,59,31,61,40,89,19,1,38,14,100,28,86,63,25,42,3
49,22,7,30,1,47,77,75,94,25,96,5,2,69,54,66,30,10,53,34,35,31,51,71,28,8,58,60,2,13,29,50,77,15,28,2,49,3
,58,67,45,6,54,8,74,44,50,32,78,23,54,56,49,93,94,78,99,88,71,30,56,21,62,22,53,71,26,72,70,4,28,20,49,24,
```

Figura 6.8: Arquivo descritor da memória para o algoritmo do método de *Jacobi*.

6.2.2 Resultados Obtidos

A partir da criação dos arquivos XML, tornou-se possível a execução da aplicação tanto com a arquitetura ShareD-GM quanto com a implementação anterior do módulo de execução VirD-GM, com o objetivo de avaliar as melhorias na execução da aplicação e realizar cálculos de *speedup* relativo.

As tabelas 6.3 e 6.4 resumem os tempos de execução encontrados, bem como os cálculos de *speedup* relativo.

Deve-se salientar que os resultados são relativos a execução do algoritmo com 1, 8 e 16 processadores e utilizando-se matrizes de 1280×1280 e 1920×1920 com 10 iterações em cada matriz.

Tabela 6.3: Tempos de execução do algoritmo de *Jacobi* (ShareD-GM)

Nº de Proc.	***Alocação de Nodos***	***T. M. Execução(s)***	***Speedup Relativo***	***Matriz***
16	N1, . . . , N8	24,070	1,80	1280×1280
8	N1, N2, N3, N4	25,139	1,71	1280×1280
1	N1	43,105	-	1280×1280
16	N1, . . . , N8	44,234	1,68	1920×1920
8	N1, N2, N3, N4	45,008	1,65	1920×1920
1	N1	74,125	-	1920×1920

Tabela 6.4: Tempos de execução do algoritmo de *Jacobi* (VirD-GM)

Nº de Proc.	***Alocação de Nodos***	***T. M. Execução(s)***	***Speedup Relativo***	***Matriz***
16	N1, . . . , N8	2344,133	1,07	1280×1280
8	N1, N2, N3, N4	2402,109	1,04	1280×1280
1	N1	2507,816	-	1280×1280
16	N1, . . . , N8	5131,148	1,13	1920×1920
8	N1, N2, N3, N4	5258,487	1,10	1920×1920
1	N1	5799,531	-	1920×1920

6.3 Análise dos Resultados

A análise das tabelas apresentadas nas seções 6.1.2 e 6.2.2 que resumem os dados obtidos nos testes desenvolvidos, mostraram que a arquitetura ShareD-GM possui um menor tempo de execução em relação a implementação anterior do módulo de execução do Ambiente D-GM, VirD-GM. Na maioria dos testes realizados e principalmente manipulando matrizes, obteve-se uma redução de tempo de execução significativa em relação a implementação anterior. Este menor tempo de execução verificado se deve ao uso da memória distribuída proporcionada pelo *Terracotta*, minimizando a serialização de grandes quantidades de dados (matrizes) que ocorria na implementação anterior. Portanto, esta análise sugere um desempenho mais favorável à arquitetura ShareD-GM, pela qual ocorre o envio apenas dos dados modificados através da rede e não do objeto inteiro.

Além das otimizações de comunicação interprocessos, maior abstração na programação e possibilidade de melhoria no desempenho, já evidenciaram-se ganhos na primeira prototipação da ShareD-GM, justificando a sua utilização para a execução paralela de aplicações como o método de *Jacobi*, que tem como característica o paralelismo de dados (operações simultâneas nos elementos do *array*) e o algoritmo de *Smith-Waterman*, cuja característica é a independência dos processos a serem executados e a independência dos dados usados nas comparações realizadas.

Observou-se também, durante a execução dos testes, que a adição de mais processadores não necessariamente melhora o desempenho. Esse problema, melhorado pelo *Terracotta*, se deve, provavelmente, ao escalonador de processos implementado no módulo de execução VirD-GM. Nas considerações finais pertinentes ao trabalho, seção 7.4, encontra-se a definição deste problema e uma proposta de solução como continuidade do projeto.

7 CONSIDERAÇÕES FINAIS

Atualmente, tem-se incrementado a pesquisa e o desenvolvimento de ambientes computacionais com suporte à implementações de memória compartilhada distribuída, seja pelas melhorias nas redes de computadores ou pelas melhorias na construção de tais sistemas. Este fato indica que os sistemas DSM são uma abstração poderosa para a comunicação inter-processos.

Muitos dos estudos e projetos avaliados neste trabalho sinalizam com a possibilidade de que o uso de sistemas DSM pode aliviar o programador da preocupação com a comunicação entre os processos, distribuindo os dados compartilhados de forma transparente entre os computadores de um *cluster* e permitindo a execução distribuída de aplicações paralelizáveis.

Outro fator a se considerar, a melhoria do desempenho das aplicações que executam de forma distribuída em um *cluster*, sempre foi o objetivo a ser alcançado por sistemas DSM, sem muito sucesso. A introdução do sistema do *Terracotta* vem contribuir de forma significativa neste quesito, resultados obtidos em testes de desempenho (SUN, 2008) indicaram que o *Terracotta* poderia vir a melhorar também o desempenho do Ambiente D-GM.

Devido a essas possibilidades e também a grande compatibilidade com o módulo de execução do Ambiente D-GM e com a representação de memória do modelo GM, o uso do *Terracotta* para a modelagem, especificação e otimização da arquitetura de memória distribuída do Ambiente D-GM foi fortemente considerado e recomendado.

A avaliação, estudo e revisão das fundamentações do modelo GM e Ambiente D-GM, dos sistemas DSM e suas implementações em *software*, fazendo uma avaliação das características que colaborariam para a implementação da memória distribuída do Ambiente D-GM, contribuiu para uma maior aprendizagem e para consolidar o *Terracotta* como o sistema mais apto para a modelagem da integração entre um sistema de memória compartilhada distribuída e o Ambiente D-GM.

A finalização da modelagem proposta e a implementação da arquitetura ShareD-GM permitiu a realização de testes, com o objetivo de avaliar a mesma, com aplicações tais como o algoritmo de *Smith-Waterman* e o algoritmo do método de Jacobi, verificando uma melhora na execução de aplicações em relação a implementação anterior do módulo de execução VirD-GM.

7.1 Principais Contribuições

As principais contribuições deste trabalho são detalhadas a seguir conjuntamente com os aspectos específicos que as caracterizam e são identificadas como de dois tipos:

- Concepção e modelagem de uma arquitetura de memória distribuída através da integração entre o módulo de execução do Ambiente D-GM (VirD-GM) e uma implementação em *software* de DSM (*Terracotta*), tendo como objetivo proporcionar uma maior funcionalidade para o ambiente como um todo, melhorando o seu fluxo de execução e também aumentar, se possível, o desempenho de execução de aplicações modeladas por meio de programação visual no Ambiente D-GM. As características que colaboraram para definição, formatação e consolidação desta contribuição são dadas pelos seguintes aspectos específicos:

  - Revisão da literatura relacionada ao Ambiente D-GM, incluindo seus módulos de programação visual (VPE-GM) e execução (VirD-GM), bem como conceitos básicos sobre o modelo de Máquina Geométrica (Modelo GM);
  - Estudo das fundamentações de memória compartilhada distribuída (DSM), principalmente direcionado para as implementações em *software* deste tipo de tecnologia e modelos de programação;
  - Identificação dos principais requisitos do Ambiente D-GM, referentes a definição de uma arquitetura de memória distribuída por meio de uma implementação em *software* de DSM com respeito as funcionalidades e desempenho;
  - Definição do *software Terracotta*, através de análise comparativa, como opção mais viável de implementação em *software* de DSM para integração ao Ambiente D-GM, servindo como base para a modelagem da arquitetura de memória distribuída deste ambiente;

- Implementação e testes da ShareD-GM consolidaram a integração entre o módulo de execução distribuída (VirD-GM) de um ambiente (Ambiente D-GM) destinado a proporcionar a modelagem e execução de aplicações paralelas de uma forma mais simples e o sistema de memória compartilhada distribuída (DSM) *Terracotta* obtendo dessa forma uma arquitetura de memória distribuída para o referido ambiente. Esta integração além de adquirir uma maior funcionalidade para execução de aplicações modeladas para execução no Ambiente D-GM também apresenta um desempenho maior que a implementação anterior para a maioria destas aplicações e agrega um maior número de recursos possibilitando expansões não permitidas antes da integração. Os seguintes aspectos específicos caracterizam esta contribuição:

  - Estudo e realização de testes com o *software Terracotta* com o intuito de verificar as suas capacidades e funcionalidades e possibilidade de integração com o módulo de execução do Ambiente D-GM (VirD-GM);
  - Revisão do código JAVA construído para o módulo VirD-GM para identificação das classes e objetos que vão fazer parte da arquitetura de memória distribuída do Ambiente D-GM, evitando que estes objetos sejam passados como parâmetro para os nodos do *cluster* e diminuindo as

serializações. Este fato, além de simplificar a programação, aumenta a funcionalidade do módulo de execução VirD-GM com possíveis ganhos de desempenho das aplicações modeladas no Ambiente D-GM;

– Especificação do arquivo de configuração necessário para a implementação da efetiva integração entre o *software Terracotta* e o módulo de execução VirD-GM através de parâmetros de controle e determinação de objetos compartilhados;

– Avaliação da ShareD-GM por meio de sua execução provocada pela modelagem de aplicações paralelas no Ambiente D-GM. Para a realização dos testes, considera-se aplicações como o algoritmo de *Smith-Waterman* e do método de Jacobi, envolvendo além destas aplicações o *software Terracotta* e o módulo de execução VirD-GM. Os resultados obtidos com as aplicações sinalizam favoravelmente para o uso de sistemas de memória compartilhada distribuída (DSM), cujo emprego além de simplificar a programação em sistemas multicomputadores pode também majorar o desempenho.

7.2 Outras Contribuições

Pode-se citar além das contribuições técnicas citadas anteriormente, as seguintes contribuições:

- **Proposta Inovadora**: Por propor o uso do *software Terracotta*, ainda pouco usado, mas de grande utilidade e com possibilidades de acelerar as aplicações JAVA que necessitem executar em ambientes de memória distribuída. Espera-se que este trabalho incentive o uso do *software Terracotta* em aplicações científicas, contribuindo para que outros projetos tenham a oportunidade de utilizar este *software* em suas aplicações científicas desenvolvidas na linguagem JAVA;

- **Incentivo ao Uso da Programação Distribuída e Paralela para Execução de Aplicações**: A introdução de uma maior funcionalidade proporcionada pelo uso do *software Terracota* incentiva uma maior utilização do Ambiente D-GM para execução de aplicações da computação científica. A execução distribuída e paralela das aplicações da computação científica pode diminuir o tempo de execução para o retorno de resultados, colaborando, dessa forma, com uma maior produtividade para projetos que façam uso deste tipo de aplicações;

- **Integração entre os Grupos de Pesquisa GMFC e G3PD**: A modelagem e implementação da arquitetura ShareD-GM, a partir da integração entre o módulo de execução, VirD-GM, e a implementação em *software* de memória compartilhada distribuída, *Terracotta*, consolida a proposta de integração entre dois grupos de pesquisa da Universidade Católica de Pelotas (UCPEL): Grupo de Matemática e Fundamentos da Computação (GFMC) e o Grupo de Pesquisa em Processamento Paralelo e Distribuído (G3PD);

- **Desenvolvimento Teórico-Prático na Área de Memória Compartilhada Distribuída**: Proporcionado pela realização desta dissertação e envolvendo principalmente, a execução distribuída de aplicações como o módulo de execução VirD-GM

em um espaço comum de endereçamento, contribuiu para que se tornasse mais efetiva a cooperação entre as linhas de pesquisa, as disciplinas e os projetos referentes aos grupos de pesquisa envolvidos;

- **Formação de Recursos Humanos**: Salienta-se ainda a integração com alunos de IC (Iniciação Científica) e mestrandos, entre as atividades que possibilitaram o desenvolvimento e a análise dos resultados nos testes de avaliação da ShareD-GM.

7.3 Publicações Realizadas

Resumos publicados no CIC:

- **Prêmio Pesquisador em Pós-Graduação**: ZECHLINSKI, Gustavo M.; REISER, Renata Hax Sander; YAMIN, Adenauer Correa. *Projeto D-GM: Uma Proposta para Otimização de Desempenho do VirD-GM*. In: 17º Congresso de Iniciação Científica, 7ª Mostra de Pós-Graduação, UCPel, 2008, Pelotas. Anais do 17º Congresso de Iniciação Científica, UCPel, 2008.

- ZECHLINSKI, Gustavo M.; REISER, Renata Hax Sander; YAMIN, Adenauer Correa. *SHARED-GM: UMA PROPOSTA DE APLICAÇÃO DE SISTEMAS DSM NO AMBIENTE D-GM*. In: 18º Congresso de Iniciação Científica, 8ª Mostra de Pós-Graduação, UCPel, 2009, Pelotas. Resumos do CIC ON-LINE:`http://www.ucpel.tche.br/cic/cdcic2009/trabalhos/trabalhos.html`.

Resumos publicados em Congressos Regionais:

- ZECHLINSKI, Gustavo M.; REISER, Renata Hax Sander; YAMIN, Adenauer Correa. *Uma Proposta de Memória Distribuída Compartilhada para o Projeto D-GM*. In: IX Escola Regional de Alto Desempenho, 2009, Caxias do Sul. Anais IX Escola Regional de Alto Desempenho. Caxias do Sul: UCS, 2009. v. 1. p. 63-64.

- ZECHLINSKI, Gustavo M.; REISER, Renata Hax Sander; YAMIN, Adenauer Correa. *Ambiente ShareD-GM: Uma Proposta de Integracão de Sistemas DSM ao Ambiente D-GM*. In: X Escola Regional de Alto Desempenho, 2010, Passo Fundo. Anais X Escola Regional de Alto Desempenho. Passo Fundo: UPF, 2010. v. 1. p. 1-4.

Resumos publicados em Congressos Nacionais:

- WSCAD-SSC 2010 - XI Simpósio em Sistemas Computacionais/SBAC-PAD 2010, Petrópolis, Rio de Janeiro, 27-30 de outubro de 2010. Artigo: ZECHLINSKI, Gustavo M.; REISER, Renata Hax Sander; YAMIN, Adenauer Correa; PINHEIRO, Anderson B.. *ShareD-GM: Arquitetura de Memória Distribuída para o Ambiente D-GM*.

7.4 Continuidade do Trabalho

Como sugestão para a continuidade do trabalho tem-se a implementação do padrão *Master/Worker* (MATTSON; SANDERS; MASSINGILL, 2004) em JAVA, aproveitando a memória compartilhada distribuída proporcionada pelo sistema *Terracotta* para armazenar as tarefas a serem executadas pelos clientes. A adoção deste padrão tem por objetivo a diminuição da latência gerada pela implementação atual.

Na implementação atual do módulo de execução do Ambiente D-GM (VirD-GM), o nodo mestre fica responsável por enviar, de uma forma sequencial, as tarefas a serem executadas para os nodos clientes, gerando dessa forma uma latência no sistema como um todo.

Considerando-se o seguinte exemplo, têm-se uma visão mais precisa do problema: Um *cluster* com oito nodos, um nodo mestre para a execução do módulo VirD-GM e sete nodos clientes para execução das tarefas a serem executadas paralelamente. Assim, com esta configuração inicial, alguma aplicação modelada no módulo de programação visual VPE-GM começa sua execução. Supondo que esta aplicação gere doze tarefas para serem disparadas em paralelo, o nodo mestre do módulo VirD-GM inicia a distribuição destas tarefas pelos nodos clientes.

Como a distribuição é feita de uma forma sequencial, as tarefas são enviadas para os clientes, que as recebem e começam a execução e retornam os resultados ao término desta. A situação que vai gerar a latência é justamente no envio das tarefas, pois neste momento pode-se ter as primeiras tarefas já concluídas pelos nodos clientes iniciais enquanto que o nodo mestre ainda não terminou de enviar as tarefas para todos os nodos do *cluster*, ficando estes nodos iniciais sem tarefas para processamento pois eles dependem que o nodo mestre lhes envie as tarefas para execução.

No exemplo citado anteriormente pode-se dizer que enquanto o nodo mestre envia a quinta tarefa para o quinto nodo cliente, o primeiro e o segundo nodo já terminaram a execução de suas tarefas e estão aptos a receber mais tarefas para execução, mas o nodo mestre continua o envio até o ultimo nodo livre (sétimo nodo). Assim, somente depois do ultimo nodo receber a sua tarefa, o escalonador reinicia o envio das tarefa restantes para os primeiros nodos até chegar na décima segunda e ultima tarefa. Esta latência dos nodos ociosos pode gerar uma diminuição de desempenho na execução da aplicação.

O padrão *Master/Worker* é bem usado para distribuir a execução de aplicações e quando aplicado ao módulo VirD-GM vai inverter a responsabilidade de delegar as tarefas para execução do nodo mestre para os nodos clientes. Este padrão é particularmente útil em casos onde as aplicações não apresentam dependências entre as suas tarefas quando executadas paralelamente (*embarrassingly parallel problems*) (MATTSON; SANDERS; MASSINGILL, 2004), como é o caso das aplicações que executam no Ambiente D-GM.

O *Master/Worker* consiste basicamente de dois elementos lógicos já mencionados, o elemento *Master* e uma ou mais instâncias do elemento *Worker*. O *Master* inicia a computação com a configuração da aplicação que vai ser executada. A partir disso, uma espécie de *bag of tasks* ou fila de processos para execução é criada, permitindo a busca destes processos para execução pelos *Workers*. O *Master* então aguarda até o término das execuções, reúne os resultados e termina a aplicação.

A abordagem mais direta para implementação da *bag of tasks* é uma fila compartilhada (MATTSON; SANDERS; MASSINGILL, 2004), ou seja, uma estrutura acessível de uma forma global onde as tarefas ou processos possam ser inseridas e removidas quando possível. Mais especificamente, essa fila deve se tornar acessível globalmente

por meio da memória compartilhada distribuída proporcionada pelo sistema *Terracotta*.

Geralmente, os algoritmos para implementação de padrão *Master/Worker* apresentam um balanceamento de carga dinâmico (MATTSON; SANDERS; MASSINGILL, 2004), efetuado no momento em que os *Workers* acessam a *bag of tasks* para obter mais tarefas após o término da execução de sua tarefa atual. Outro aspecto importante é a boa escalabilidade (MATTSON; SANDERS; MASSINGILL, 2004) oferecida por estes algoritmos com um número de tarefas muito superior ao número de *Workers* desde que os custos de tempo das tarefas individuais não tenham uma variação muito grande fazendo com que alguns *Workers* levem muito mais tempo que outros para executarem a referida tarefa.

A figura 7.1 mostra o fluxograma de funcionamento do padrão *Master/Worker*.

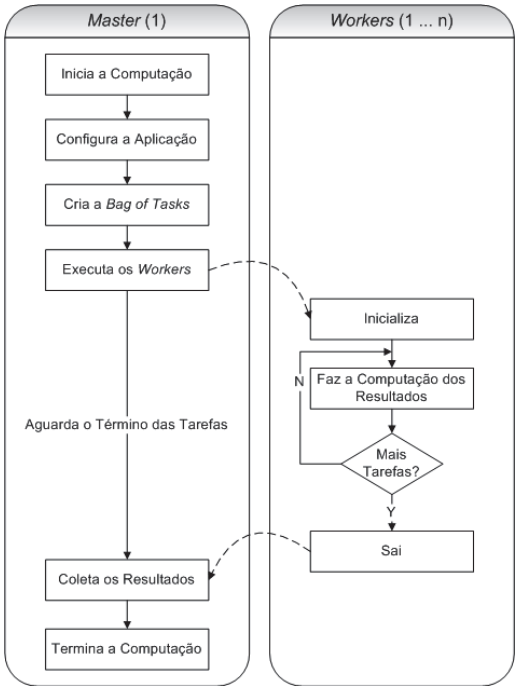

Figura 7.1: Fluxograma *Master/Worker* (MATTSON; SANDERS; MASSINGILL, 2004)

7.4.1 Implementação *Master/Worker* em JAVA

O padrão *Master/Worker* pode ser implementado em JAVA de varias maneiras, mas segundo (TERRACOTTA, 2008) existem três mais difundidas e que variam de acordo com o nível de abstração:

- Primitivas de *threading* e de concorrência - Esta é a abordagem de mais baixo nível proporcionada pela especificação da linguagem JAVA, onde se faz uso dos métodos *synchronized*, *wait*, *notify* e *volatile*. Dessa forma, pode-se afirmar que a partir desta abordagem, as customizações de uma solução podem ser feitas sem limitação. Em contrapartida a implementação por meio destas primitivas é muito difícil e exige muito tempo para ser colocada em prática, sendo um artifício para ser usado eventualmente;

- Classe java.util.concurrent - As abstrações desta classe apresentam um nível de abstração maior que as primitivas de *threading*, tendo portanto uma melhor

aceitação para implementação do padrão *master/worker*. Esta API proporciona implementações de *interfaces* das coleções JAVA com semáforos e barreiras ajustadas especialmente para acesso concorrente. Ela também fornece um *pool* de *threads* chamado *ExecutorService* que possui suporte direto para o padrão *Master/Worker* e pode ser compartilhado pelo *Terracotta*. Apesar das facilidades, esta abordagem apresenta alguns problemas como o *Master* e o *Worker* serem representados pela mesma abstração, impedindo a escalabilidade do *Master* independentemente do *Worker*, isto é, cada nodo vai possuir uma instância tanto do *Master* quanto do *Worker* e a falta de meios de monitoramento das tarefas enviadas aos nodos *Workers*, não havendo métodos para determinar se a tarefa foi completada ou rejeitada devido algum erro para que se possa realizar ações como a reinicialização da tarefa.

- Especificação *CommonJ WorkManager* - O uso desta especificação é apontado por (TERRACOTTA, 2008) como a maneira mais simples e padronizada para a implementação do padrão *master/worker*. A API definida pela especificação, estabelece os padrões para execução concorrente de tarefas em um ambiente JEE (JAVA *Enterprise Edition*). Além de controlar a submissão de tarefas para execução e o retorno de resultados, a API também fornece funcionalidades para o rastreamento do *status* da tarefa, permitindo não só a detecção da falha na execução de uma tarefa, bem como o motivo desta falha, possibilitando que a referida tarefa seja inicializada novamente.

Mais especificamente, para adaptação do padrão *master/worker* ao módulo de execução do Ambiente D-GM, a escolha natural se resume ao uso da especificação *CommonJ WorkManager* como indicada por (TERRACOTTA, 2008), por se tratar da implementação mais simples e completa do padrão *master/worker*. Outros aspectos como volumes de dados muito grandes, falha nas tarefas, *scheduling* e falha nos *Workers* podem ser tratados a partir de adaptações na especificação *CommonJ WorkManager*. Dentre estes aspectos, talvez o mais importante a ser considerado seja o *scheduling*, responsável por definir quais tarefas serão enviadas e para quais nodos do cluster. Algumas soluções como *Round Robin*, *Work load Sensitive Balancing* e *Data Affinity* são propostas para uso conjuntamente com o padrão *master/worker*, mas de acordo com (MATTSON; SANDERS; MASSINGILL, 2004), o uso de uma fila compartilhada de tarefas centralizada, especialmente em ambientes de memória distribuída, pode gerar um gargalo e a solução ótima seria a adoção de uma algoritmo de *scheduling* do tipo *work stealing* como o implementado em (FRIGO; LEISERSON; RANDALL, 1998).

A implementação completa do padrão *master/worker* baseada na especificação *CommonJ WorkManager* que poderá ser adaptada ao módulo de execução do Ambiente D-GM, a configuração necessária para o *Terracotta*, as modificações para sua execução em um ambiente real e algumas opções de algoritmos de *scheduling* podem ser encontrados em `http://jonasboner.com/2007/01/29/how-to-build-a-pojo-based-data-grid-using-open-terracotta.html`.

REFERÊNCIAS

ADVE, S. V.; GHARACHORLOO, K. Shared memory consistency models: A tutorial. In: **IEEE Comput. Mag.** [S.l.: s.n.], 1996. v.29, n.12, p.66–76.

AHUJA, S.; CARRIERO, N.; GELERNTER, D. Linda and Friends. **Computer**, [S.l.], v.19, n.8, p.26–34, Aug. 1986.

APON, A.; BUYYA, R.; JIN, H.; MACHE, J. Cluster Computing in the Classroom: Topics, Guidelines, and Experiences. **Cluster Computing and the Grid, IEEE International Symposium on**, Los Alamitos, CA, USA, v.0, p.476, 2001.

ARAúJO, E. B. **Um Estudo sobre Memória Compartilhada Distribuída**. Trabalho Individual I, UFRGS.

BAL, H.; BHOEDJANG, R.; HOFMAN, R.; JACOBS, C.; LANGENDOEN, K.; RüHL, T.; KAASHOEK, M. Performance evaluation of the Orca shared-object system. **ACM Trans. Comput. Syst.**, New York, NY, USA, v.16, n.1, p.1–40, 1998.

BAL H.E., K. M.; A.S., T. Orca: A Language for Parallel Programming of Distributed Systems. In: **IEEE Transactions on Software Engineering**. [S.l.: s.n.], 1992. v.18, n.3, p.190–205.

BARRETT, R.; BERRY, M.; CHAN, T. F.; DEMMEL, J.; DONATO, J.; DONGARRA, J.; EIJKHOUT, V.; POZO, R.; ROMINE, C.; VORST, H. V. der. **Templates for the Solution of Linear Systems**: Building Blocks for Iterative Methods, 2nd Edition. Philadelphia, PA: SIAM, 1994.

BERSHAD, B. N.; ZEKAUSKAS, M. **Midway**: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Pittsburgh, Penn: Carnegie-Mellon University, 1991.

BIRRELL, A.; NELSON, B. Implementing remote procedure calls. In: **ACM Transactions on Computer Systems**. [S.l.]: ACM Press, 1984. v.2, n.1, p.39–59.

BROWN, C.; MCDONALD, C. Visualizing berkeley socket calls in students' programs. **SIGCSE Bull.**, New York, NY, USA, v.39, n.3, p.101–105, 2007.

C. AMZA A.L. COX, S. D. P. K. H. L. R. R. W. Y.; ZWAENEPOEL, W. Treadmarks: Shared Memory Computing on Networks of Workstations. In: **Computer**. [S.l.: s.n.], 1996. v.29, n.2, p.18–28.

CARTER, J. B. Design of the Munin Distributed Shared Memory System. **Journal of Parallel and Distributed Computing**, [S.l.], v.29, p.219–227, 1995.

COLOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Distributed Systems**: Concept and Design. 3.ed. [S.l.]: Addison-Wesley, 2003.

ESKICIOGLU, M. R. **Software distributed shared memory**: issues and a case study. 2004. Tese (Doutorado em Ciência da Computação) — , Edmonton, Alta., Canada.

FENWICK, K. **A Performance Analysis of Java Distributed Shared Memory Implementations**. [S.l.]: Adelaide University, Department of Computer Science, Distributed and High Performance Computing Group, 2001.

FONSECA, V. S. d. **VirD-GM**: Uma Contribuição Para o Modelo de Distribuição e Paralelismo do Projeto D-GM. 2008. Dissertação de Mestrado em Ciência da Computação — UCPel, Pelotas,RS.

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The Implementation of the Cilk-5 Multithreaded Language. In: IN PROCEEDINGS OF THE SIGPLAN'98 CONFERENCE ON PROGRAM LANGUAGE DESIGN AND IMPLEMENTATION, 1998. **Anais...** [S.l.: s.n.], 1998. p.212–223.

GHARACHORLOO K., L. D. L. J. G. P. G. A. A. J. Memory consistency and event ordering in scalable shared-memory multiprocessors. In: IN PROCEEDINGS OF THE 17TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 1990, Los Alamitos, Calif. **Anais...** IEEE Computer Society Press, 1990. p.15–26.

GOSLING, J.; MCGILTON, H. The java language environment: A white paper. **JavaSoft, Sun Microsystems, Inc**, [S.l.], 1996.

HERLIHY M. P., A. J. M. A correctness condition for concurrent objects. In: **ACM Trans. Program**. [S.l.: s.n.], 1990. v.12, n.3, p.463–492.

KELEHER P., C. A. L.; ZWAENEPOEL, W. Lazy release consistency for software distributed shared memory. In: PROCEEDINGS OF THE 19TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 1992, Los Alamitos, Calif. **Anais...** IEEE Computer Society Press, 1992. p.13–21.

LAMPORT, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. In: **IEEE Trans. Comput.** [S.l.: s.n.], 1979. v.C-28, n.9, p.690–691.

LAMPORT, L. Part II: Algorithms Distrib. Comput. In: **On Interprocess Communication**. [S.l.: s.n.], 1986. v.1, n.2, p.86–101.

LETIZI, O. **An introduction to terracotta distributed shared objects - http**://www.terracotta.org/.

LI, K. IVY: A Shared Virtual Memory System for Parallel Computing. In: IN PROCEEDINGS OF THE 1988 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, 1988, Pennsylvania. **Anais...** State University Press, 1988. p.94–101.

LI, K.; HUDAK, P. Memory coherence in shared virtual memory systems. In: **ACM Transactions on Computer Systems**. [S.l.]: ACM Press, 1989. v.7, n.4, p.321–359.

LIAO, H.-Y.; YIN, M.-L.; CHENG, Y. A parallel implementation of the Smith-Waterman algorithm for massive sequences searching. In: ENGINEERING IN MEDICINE AND BIOLOGY SOCIETY, 2004. IEMBS '04. 26TH ANNUAL INTERNATIONAL CONFERENCE OF THE IEEE, 2004. **Anais...** [S.l.: s.n.], 2004. v.2, p.2817 –2820.

LINDHOLM, T.; YELLIN, F. **The Java Virtual Machine Specification**. [S.l.]: SUN Microsystems Press, 1999.

LIPTON, R. J.; SANDBERG, J. S. **Pram**: A scalable shared memory. [S.l.]: Princeton University, 1988.

LO, V. Operating Systems Enhancements for Distributed Shared Memory. In: **Advances in Computer**. San Diego: Academic Press, 1994. v.39, p.191–237.

MA, M. J. M.; WANG, C.-L.; LAU, F. C. M. JESSICA: Java-enabled single-system-image computing architecture. **J. Parallel Distrib. Comput.**, Orlando, FL, USA, v.60, n.10, p.1194–1222, 2000.

MATTSON, T.; SANDERS, B.; MASSINGILL, B. **Patterns for Parallel Programming**. [S.l.]: Addison-Wesley Professional, 2004.

MEHMOOD, R.; CROWCROFT, J. **Parallel iterative solution method for large sparse linear equation systems**. United Kingdom: University of Cambridge, 2005. (650).

MENG, X.; CHAUDHARY, V. A High-Performance Heterogeneous Computing Platform for Biological Sequence Analysis. **IEEE Transactions on Parallel and Distributed Systems**, Los Alamitos, CA, USA, v.99, n.RapidPosts, 2009.

MUNHOZ, F. N. **Expandindo o VirD-GM para Suporte às Demandas do Projeto D-GM**. Projeto de Graduação II.

PARALLEL-TOOLS. **Concurrent Programming with TreadMarks**. [S.l.: s.n.], 1994.

PARK, I. **Parallel programming methodology and environment for the shared memory programming model**. 2000. Tese (Doutorado em Ciência da Computação) — , West Lafayette, IN, USA. Major Professor-Eigenmann, Rudolf.

PHILIPPSEN, M.; HAUMACHER, B.; NESTER, C. More efficient serialization and RMI for Java. In: CONCURRENCY: PRACTICE AND EXPERIENCE, 2000, Computer Science Department, University of Karlsruhe, Am Fasanengarten 5, 76128 Karlsruhe, Germany. **Anais...** John Wiley and Sons: Ltd., 2000. v.12, n.7, p.495–518.

PRESTES, D. G.; REISER, R. H. S.; CARDOSO, M. B.; ROCHA COSTA, A. C. da. Estendendo o Modelo de Máquina Geométrica a um Ambiente de Programação Visual. In: XXXI CONFERENCIA LATINOAMERICANA DE INFORMATICA, 2005, Pontificia Univers. Javeriana, Cali. **Anales...** [S.l.: s.n.], 2005. v.1, p.273–284.

REISER, R. **A Máquina Geométrica - Um Modelo Computacional para Concorrência e Não-determinismo Usando como Estrutura os Espaços Coerentes**. 2002. Tese — II/UFRGS, POA, RS.

REISER, R.; COSTA, A.; DIMURO, G. A Programming Language for the Interval Geometric Machine Model. In: ELETRONIC NOTES IN THEORETICAL COMPUTER SCIENCE, 2003. **Anais...** [S.l.: s.n.], 2003. v.84, p.1–12.

REISER, R. H. S.; ROCHA COSTA, A. C. da; DIMURO, G. P.; CARDOSO, M. B. Specifying the geometric machine visual language. In: HCC, 2003. **Anais...** [S.l.: s.n.], 2003. p.186–188.

SHEN, X.; ARVIND; SHEN, X. A Methodology for Designing Correct Cache Coherence Protocols for DSM Systems. In: CSG MEMO 398 (A), LCS, MIT, 1997. **Anais...** [S.l.: s.n.], 1997.

STEINKE, R. C.; NUTT, G. J. A unified theory of shared memory consistency. **J. ACM**, New York, NY, USA, v.51, n.5, p.800–849, 2004.

STUMM, M.; ZHOU, S. Algorithms Implementing Distributed Shared Memory. In: **Distributed Shared Memory**: Concepts and Systems. Los Alamitos, California: IEEE Computer Society Press, 1998. p.54–64.

SUN, W. **Clustering with Terracotta**. 2008. Dissertação (Mestrado em Ciência da Computação) — Agder University College.

TANENBAUM, A. S. **Modern Operating Systems**. 1.ed. [S.l.]: Prentice Hall, Inc., 1992.

TANENBAUM, A. S. **Distributed Operating Systems**. [S.l.]: Prentice-Hall, 1995.

TANENBAUM, A. S.; STEEN, M. V. **Distributed Systems**: Principles and Paradigms. 1.ed. [S.l.]: Prentice Hall PTR, 2001.

TANENBAUM, A. S.; STEEN, M. V. **Sistemas Distribuídos**: Princípios e Paradigmas. 2 ed..ed. São Paulo: Pearson Prentice Hall, 2007.

ANGLIN, S. (Ed.). **The Definitive Guide to Terracotta**: Cluster the JVM for Spring, Hibernate and POJO Scalability. [S.l.]: Apress, 2008.

THIRUVATHUKAL, G. K. Guest Editor's Introduction: Cluster Computing. **Computing in Science and Engineering**, [S.l.], v.7, n.2, p.11–13, March/April 2005.

VENNERS, B. **Inside the Java Virtual Machine**. [S.l.]: McGraw-Hill, 1997.

WATERMAN, M. S.; SMITH, T. F. Identification of common molecular subsequences. **J. Mol. Biol.**, [S.l.], v.147, p.195–197, 1981.

YAMIN, A. **Arquitetura Para um Ambiente de Grade Computacional Direcionado as Aplicações Distribuídas, Móveis e Conscientes do Contexto da Computação Pervasiva**. 2004. Tese de Doutorado em Ciência da Computação — Instituto de Informática, UFRGS, Porto Alegre, RS.

ZENGER, M. **JavaParty - Transparent Remote Objects in Java**.

ANEXO A PRINCIPAIS TECNOLOGIAS ENVOLVIDAS

Alguns conceitos e definições sobre as áreas de pesquisa relacionadas diretamente com esta dissertação são apresentados aqui com o intuito de estabelecer as principais tecnologias relacionadas com este trabalho e definindo o contexto onde o mesmo se encontra inserido. A seção A.1 descreve os conceitos de *cluster*, situando o Ambiente D-GM como uma especialização deste tipo de computação paralela. Na sequência, a seção A.2 faz uma introdução sobre o ambiente e a linguagem JAVA, incluindo suas estruturas, ressaltando sua importância na computação distribuída e descrevendo como é feita a comunicação padrão entre os objetos.

A.1 *Clusters*

O termo *cluster* se refere a computadores independentes combinados em um sistema unificado através de uma LAN (*Local Area Network*) e um *software* utilizado para execução de aplicações. Mais genericamente, quando utiliza-se dois ou mais computadores para resolver um problema computacional, pode-se considerar isto um *cluster*. Dessa forma, ao utilizar um conjunto de computadores para resolução de problemas, faz-se uso da computação distribuída. A computação distribuída é caracterizada principalmente pelo emprego de sistemas distribuídos para a realização das computações. Na computação distribuída um problema é dividido em várias tarefas, cada uma resolvida por um computador que compõe o sistema e segundo (TANENBAUM; STEEN, 2001) esse sistema se apresenta aos seus usuários como único e coerente.

Atualmente, os *clusters* são mais comumente usados para HA (*High Availability*), onde os serviços do *cluster* têm alta disponibilidade com o uso de nodos redundantes e impedindo falhas no fornecimento dos serviços. Além disso, tem-se a opção para HPC (*High Performance Computing*), sendo esta a a principal circunstância do Ambiente D-GM, visando prover um poder computacional maior que apenas uma máquina individual poderia propiciar (THIRUVATHUKAL, 2005).

Além de se beneficiar dos *clusters*, o Ambiente D-GM também tira proveito da arquitetura cliente/servidor (COLOURIS; DOLLIMORE; KINDBERG, 2003) definida como um modelo computacional que faz a distinção entre computadores interligados por uma rede de computadores, fixando-os como clientes ou servidores. Os computadores clientes podem enviar requisições de dados para os servidores e aguardarem pela resposta. Os computadores servidores disponíveis podem aceitar tais requisições, processá-las e

retornar os resultados para os computadores clientes.

Os *clusters* são tipicamente construídos usando componentes de hardware COTS (*Commodity-off-the-shelf*) e *software* livre. Computadores COTS, são computadores fabricados por diversas empresas que incorporam componentes baseados em padrão aberto. Estes computadores são baseados em componentes *commodity*, de onde provém a sua denominação. Tais componentes, possuem baixo custo e diminuem as diferenças entre os fabricantes devido a padronização dos processos de fabricação. Com base nestas considerações, pode-se dizer que a computação em *cluster* oferece um recurso computacional de alta performance e baixo custo para instituições educacionais, uma vez que os computadores de um laboratório de estudantes podem ser transformados em um pequeno *cluster*, usufruindo ainda da estrutura de *software* livre disponibilizada pela Internet.

A arquitetura genérica de um *cluster* é mostrada na figura A.1.

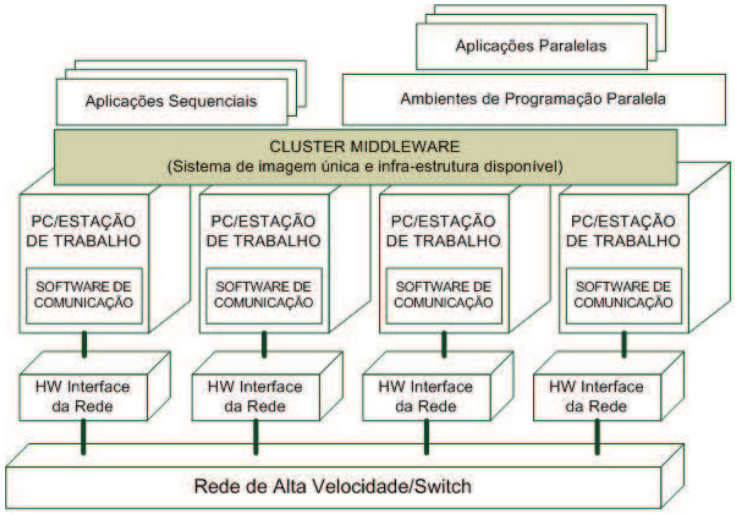

Figura A.1: Arquitetura de um *cluster* (APON et al., 2001)

A.2 JAVA

A linguagem JAVA foi introduzida pela SUN em 1995 e alcançou grande popularidade por estar diretamente relacionada à Internet, ou seja, qualquer computador ligado a internet e equipado com um *browser* habilitado podia executar as APPLET's JAVA. Isto só se tornou possível porque o ambiente JAVA especifica além de um linguagem de programação, uma máquina virtual (FENWICK, 2001). Essa máquina virtual recebe a denominação de JVM (JAVA *Virtual Machine*), sendo os programas em JAVA compilados para ela. Os programas em JAVA compilados para a JVM podem executar em qualquer arquitetura, desde que esta implemente ou tenha configurado a máquina virtual JAVA (LINDHOLM; YELLIN, 1999). Assim, percebe-se que o ambiente JAVA foi concebido visando os sistemas distribuídos, por exemplo os *clusters*, cujos componentes, frequentemente de arquiteturas diferentes, não necessitam de uma programação especial ou sistema operacional específico para execução de aplicações JAVA (GOSLING; MCGILTON, 1996).

Outras características da linguagem JAVA e de grande importância para a computação distribuída são a concorrência e a possibilidade de paralelismo. Estas car-

acterísticas são viabilizadas por duas unidades básicas de execução da linguagem: (i) Processos e; (ii) *Threads*.

As *threads*, conhecidas como processos leves por exigirem bem menos recursos que um processo para a sua criação, estão mais relacionadas a programação concorrente e/ou paralela. Em sistemas de computação com múltiplos processadores ou processadores com múltiplos núcleos de execução é comum a execução de múltiplas *threads* (execução *multithreaded*), sendo esta uma característica que aumenta a capacidade para a execução concorrente e/ou paralela.

Os processos contém internamente um ambiente de execução, ou seja, genericamente um processo é completo, com seu conjunto privado de recursos básicos de tempo de execução. Em outras palavras, um processo tem o seu próprio espaço de memória. As *threads* existem dentro de um processo, cada processo contém pelo menos uma *thread*.

O módulo de execução do Ambiente D-GM, por fazer uso da computação distribuída e da execução *multithreaded* necessita implementar um mecanismo de comunicação entre os objetos criados pelas múltiplas *threads*. O mecanismo atualmente implementado para esta comunicação entre objetos se utiliza de duas estruturas comumente usadas para este propósito:

- A API *Berkeley Sockets* - Uma biblioteca de funções, macros e estrutura de dados que permitem a um programa criar e gerenciar fluxos de comunicação entre dois ou mais processos na mesma máquina ou em uma rede de máquinas. A API *sockets* é uma solução popular e portável para o estabelecimento de comunicação inter-processos, tendo a sua implementação disponível na maioria das linguagens de programação e na maior parte dos sistemas operacionais (BROWN; MCDONALD, 2007).

- O RMI (*Remote Method Invocation*) - É um modelo que permite a computação distribuída com objetos JAVA de uma maneira simples e direta. Em um nível mais baixo, o RMI é o mecanismo RPC (*Remote Procedure Call*) (BIRRELL; NELSON, 1984) para JAVA com algumas vantagens sobre o tradicional RPC porque ele é parte da abordagem orientada a objetos do JAVA. 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. A estrutura do RMI permite que um objeto chame um método em um objeto remoto, ou seja, permite que um objeto chame um método em um objeto que está em outra JVM e geralmente em outra máquina. A abordagem cliente/servidor é usada pelo RMI para descrever as invocações. O objeto que faz a chamada ao método no objeto remoto é o cliente e o objeto remoto é o servidor (FENWICK, 2001).

Outro mecanismo importante e responsável pelo sucesso da heterogeneidade do ambiente JAVA, a JVM possui uma estrutura dividida em módulos de *software* e organizados de acordo com a figura A.2.

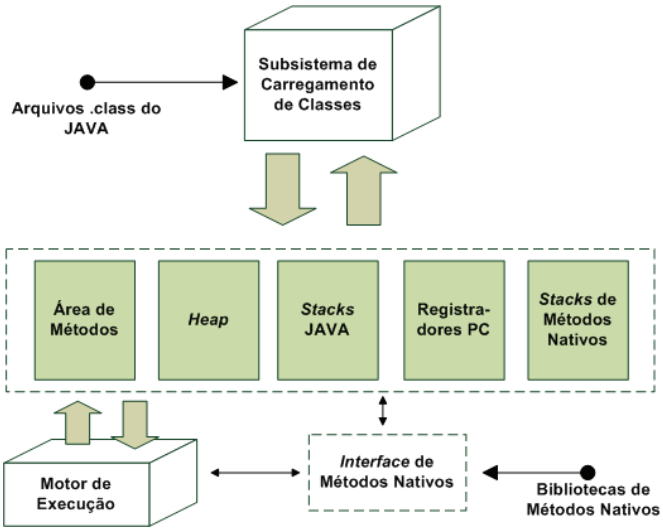

Figura A.2: Estrutura da JVM (FENWICK, 2001)

Dessa forma, a principal função da JVM é carregar os arquivos de classe e executar os *bytecodes* que eles contém (VENNERS, 1997). Para a realização deste procedimento, os módulos da JVM possuem tarefas específicas:

- *Class loader*: Subsistema de carregamento de classes;
- Motor de Execução (*Execution Engine*): Responsável pela execução das instruções contidas nos métodos e nas classes carregadas;
- Área de métodos (*Method Area*): Área de armazenamento, compartilhada por todas as *threads*, para o código compilado da linguagem;
- *Heap*: Área de dados de tempo de execução onde a memória para todas as instâncias de classe e *arrays* é alocada (LINDHOLM; YELLIN, 1999). Esta área é criada na inicialização da JVM e compartilhada por todas as *threads*;
- Coletor de Lixo (*Garbage Collector*): É um sistema automático de gerenciamento de recursos, cuja função é desalocar os objetos que não são mais usados pela aplicação;
- Registradores PC (*PC Registers*): São os contadores de programa de cada *thread* que começa a executar e indica sempre a próxima instrução à executar;
- *Stacks* JAVA: armazena o estado das chamadas de métodos JAVA da *thread* incluindo as variáveis locais, os parâmetros com os quais esse método foi chamado, o valor de retorno se houver e os cálculos intermediários. Cada método invocado pela *thread* gera um *frame* no *stacks* JAVA, que é descartado após o término deste método;
- *Stacks* de Métodos Nativos (*Native Methods Stacks*) e Interface de Métodos Nativos (*Native Method Interface*): Dois módulos referentes aos métodos nativos, cujas funções são respectivamente: Armazenar o estado dos métodos nativos que não são implementados na linguagem JAVA e realizar a interação com a biblioteca dos métodos nativos.

ANEXO B ARQUIVO DE CONFIGURAÇÃO SHARED-GM

```
<?xml version="1.0" encoding="UTF-8"?>

<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.terracotta.org/schema/terracotta
    -5.xsd">

<servers>
<server host="g3pd">
<data>%(user.home)/terracotta/server-data</data>
<logs>%(user.home)/terracotta/server-logs</logs>
<dso>
<persistence>
<mode>temporary-swap-only</mode>
</persistence>
</dso>
</server>
</servers>

<clients>
<logs>%(user.home)/terracotta/client-logs</logs>
</clients>

<application>
<dso>
<roots>
<root>
<field-name>VirdLauncher.virdmemory</field-name>
</root>
</roots>
```

Listagem B.0.1: Arquivo de Configuração Completo ShareD-GM - Parte 1

```
<instrumented-classes>
<include>
<class-expression>g3pd.virdgm.core.VirdMemory</class-expression>
</include>
<include>
<class-expression>g3pd.virdgm.core.VirdLauncher</class-
   expression>
</include>
<include>
<class-expression>g3pd.virdgm.apps.*</class-expression>
</include>
</instrumented-classes>

<locks>
<autolock>
<method-expression>
void g3pd.virdgm.core.VirdMemory.updateMemory(Object,Integer)
</method-expression>
<lock-level>synchronous-write</lock-level>
</autolock>
<autolock>
<method-expression>
void g3pd.virdgm.core.VirdMemory.writeMemory(Object,Integer)
</method-expression>
<lock-level>synchronous-write</lock-level>
</autolock>
</locks>

</dso>
</application>
</tc:tc-config>
```

Listagem B.0.2: Arquivo de Configuração Completo ShareD-GM - Parte 2

ANEXO C PREPARAÇÃO DO AMBIENTE

Neste anexo são apresentadas as etapas de configuração necessárias para a efetiva execução da ShareD-GM no sistema operacional Linux. Embora esta preparação tenha sido feita para a ShareD-GM em específico, ela pode também ser aplicada a qualquer aplicação que use o sistema *Terracotta* como sistema de memória compartilhada distribuída.

As etapas necessárias à preparação do *Terracotta* para execução no sistema operacional Linux são:

A Fazer o *download* e instalar os arquivos necessários ao seu funcionamento:

1. *download* do arquivo Terracotta-3.1.1-installer.jar em `http://www.terracotta.org/dl/oss-download-catalog`;
2. *download* do JAVA *Development Kit* ou JDK para Linux, opção *Self Extracting Linux*, pois são necessários o compilador e as ferramentas de desenvolvimento JAVA que não estão presentes no *JAVA Runtime Environment* ou JRE que é a alternativa mais comum para se obter o JAVA. O JDK pode ser baixado em `http://java.sun.com/javase/downloads/index.jsp`;
3. Após o *download* dos arquivos, deve-se proceder com as instalações, que devem ser feitas a partir do *prompt* de comando, dentro da pasta onde se localiza o arquivo e executando as seguintes linhas de comando: JDK - `chmod a+x jdk-6u<version>-linux-i586.bin` para ajuste da permissão de execução do arquivo e `./jdk-6u<version>-linux-i586.bin` para execução da instalação do JDK propriamente dita. Caso possua direitos de super usuário (*root*), o JDK pode ficar instalado em `/usr/local`, caso contrário o JDK ficará instalado no diretório *home* do usuário. *Terracotta* - `java -jar terracotta-3.1.1-installer.jar` para a instalação dos arquivos do sistema *Terracotta*. Isto requer o JDK já instalado e as variáveis de ambiente para o compilador JAVA corretamente configuradas, portanto é conveniente realizar primeiro a instalação do JDK.

B Ajuste das variáveis de ambiente:

1. Configurar a variável de ambiente `PATH` do Linux com a localização do sistema *Terracota*, isto deve ser feito a partir do comando: `export PATH=$PATH:/home/usuario/terracotta/bin` ou `export PATH=$PATH:/usr/local/terracotta/bin` caso o usuário seja *root*;

2. Configurar a variável de ambiente `TC_INSTALL_DIR`, cuja função é indicar ao sistema operacional Linux a localização da instalação do sistema *Terracotta*, para isto, o seguinte comando deve ser executado: `export TC_INSTALL_DIR=/home/usuario/terracotta` ou `export TC_INSTALL_DIR=/usr/local/terracotta` caso o usuário seja *root*;
3. Configurar a variável de ambiente `TC_CONFIG_PATH` com o endereço IP e a porta do servidor *Terracotta* para que os nodos clientes do *cluster* possam buscar o arquivo de configuração *tc-config.xml* no servidor e também realizar a comunicação. Esta configuração deve ser feita a partir do comando `export TC_CONFIG_PATH =200.132.45.235:9510` usando nosso endereço IP específico;
4. Outra variável de ambiente importante a se configurar é a `CLASSPATH` onde se torna necessário indicar a localização do arquivo de classes compactado (*vird-gm.jar*) do módulo de execução VirD-GM. Para isto, o seguinte comando deve ser executado: `export CLASSPATH=CLASSPATH:/home/usuario/vird-gm.jar` considerando que o arquivo *vird-gm.jar* esteja localizado no diretório *home* do usuário.

C Inicialização do servidor: Segue o mesmo padrão do ambiente Windows, onde iniciamos a execução do servidor conjuntamente com o arquivo de configuração através do seguinte comando `./start-tc-server.sh -f tc-config.xml` cuja diferença reside no formato do arquivo que é do tipo sh (*shell script*) e não *.bat* como no Windows;

D Inicialização dos clientes: Ocorre a partir do comando `./dso-java.sh g3pd.virdgm.core.VirdLauncher` seguindo o mesmo padrão usado no sistema Windows e mudando apenas o tipo de arquivo para *shell script*. Os clientes fazem uso da variável de ambiente `TC_CONFIG_PATH` para obter o endereço IP do servidor, permitindo dessa forma a comunicação entre clientes e servidor e também a transferência do arquivo *tc-config.xml* para estes clientes;

E Execução da VirD-GM: A VirD-GM é executada pelo seguinte comando `dso-java.sh g3pd.virdgm.core.VirdLauncher mem.xml proc.xml clients:número de nodos`. Os arquivos *.xml* são os arquivos exportados do módulo de programação VPE-GM a partir da modelagem de uma aplicação e contém as informações que definem a memória da aplicação, quais os processos ou aplicações da biblioteca de aplicações que devem ser executados e de que forma, paralelamente ou sequencialmente e finalmente o número de clientes que serão usados para a execução dessa aplicação.

Assim como no ambiente Windows, os arquivos *start-tc-server.sh* e *dso-java.sh* são arquivos fornecidos pela instalação do *Terracotta* evitando a utilização de muitos comandos na execução das aplicações. Salienta-se também que o ajuste das variáveis de ambiente devem ser feitas tanto no servidor quanto nos clientes e a instância da VirD-GM que recebe os parâmetros e controla os nodos clientes deve preferencialmente executar na mesma máquina onde o servidor *Terracotta* está em execução.