Universidade de Pernambuco
Escola Politécnica de Pernambuco
Departamento de Sistemas e Computação
Programa de Pós-Graduação em Engenharia da Computação

Fernando Antônio Farias Rocha

EasyP: Um Framework Configurável de Suporte à Computação Distribuida

Dissertação de Mestrado

Recife, dezembro de 2009

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

UNIVERSIDADE DE PERNAMBUCO
DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

FERNANDO ANTÔNIO FARIAS ROCHA

EasyP: Um Framework Configurável de Suporte à Computação Distribuida

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

Prof. Dr. Sérgio Castelo Branco Soares
Orientador

Prof. Dr. Ricardo Massa Ferreira Lima
Co-orientador

Recife, dezembro de 2009

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Rocha, Fernando Antônio Farias

EasyP: Um Framework Configurável de Suporte à Computação Distribuida / Fernando Antônio Farias Rocha. – Recife: PPGEC da UPE, 2009.

70 f.: il.

Dissertação (mestrado) – Universidade de Pernambuco. Programa de Pós-Graduação em Engenharia da Computação, Recife, BR–PE, 2009. Orientador: Sérgio Castelo Branco Soares; Co-orientador: Ricardo Massa Ferreira Lima.

1. Sistemas distribuídos. 2. Suporte ao usuário. 3. Sistemas multi-núcleo. 4. Sistemas paralelos. I. Soares, Sérgio Castelo Branco. II. Lima, Ricardo Massa Ferreira. III. Título.

"Além da noite a me proteger
escura como o inferno de pólo a pólo.
Dou graças a quantos deuses possam ser...
Para minha inconquistável alma
nas garras cruéis das circunstâncias,
não recuei nem de mim cresceu um grito,
sob os golpes da sorte que o alcançam.
O meu crânio sangra insubmisso,
para além deste lugar de choro e ira.
Avulta tão-somente o horror das trevas.
E a ameaça dos anos, apesar disso me encontra,
e encontrará, mais destemido.
Não importa quão estreito o portão.
Quão carregada a sentença de castigos.
Sou eu o sonhador do meu destino,
sou eu o capitão da minha alma"

— Willian Ernest Henley "Invictus"

Agradecimentos

Mais uma etapa, mais um desafio vencido. Odeio essa coisa de está agradecendo, até porque alguns realmente não merecem. Mas ainda existem aqueles que merecem meus agradecimentos, aos quais eu serei eternamente grato.

Primeiramente gostaria de agradecer a Sérgio Soares, meu orientador por ter me dado apoio acadêmico e muitos conselhos ao longo deste mestrado, sem contar sua participação no período final da minha graduação. Além dele, André Soares, quase um co-orientador, me ajudando muito na parte de entender como realizar o estudo de caso, mesmo ele estando nas vésperas de entregar a tese de doutorado sempre arrumou tempo para me ajudar...

Agradeço também a participação de Félix Santos e Fernando Castor na banca da defesa com diversos comentários pertinentes nos quais mostraram diversas maneiras de obter um trabalho de melhor qualidade.

Alguns amigos tem que estar presente nesta lista, como Flávio Rosendo, Robson Santana, Renata Medeiros e Policarpo Freitas, amigos que me deram muito apoio neste período. De trabalhos nas disciplinas na faculdade a problemas pessoais, passando por altas trilhas a pé e de bike. Atividades que o mestrado, ajudado por problemas físicos, me cortaram... Talvez agora, com mais tempo livre, eu possa voltar a fazer algo mais divertido para a minha vida.

Amigos da graduação, que me acompanharam durante este tempo de mestrado e irei levar por toda a vida, como PC, Leopoldo Teixeira, Vinicius, Leandro, Thiago, Leopoldo Rabelo. Devo estar esquecendo de alguns, me desculpem... Agradeço a todos vocês pelos momentos durante e após a graduação...

Aos meus pais, irmãos e sobrinhas, um obrigado especial por me suportarem em mais um período sendo um ser insuportável...

Por último, a uma pessoa muito especial na minha vida, não apenas um obrigado, mas um reconhecimento por tudo que ocorreu neste último ano. Indo diversas vezes de momentos maravilhosos a momentos terríveis. Estresses causados devido a este mestrado... E muitas vezes meu único porto seguro, uma única pessoa capaz de, em muitas vezes, me impedir de largar tudo. Minha única razão para conseguir hoje estar finalizando esta dissertação. Muito obrigado Tássia Brandão, acima de tudo minha companheira e amiga. Amo você, por tudo e por nada, para sempre...

> "When I met you, you were so unique
> Had a little thing I'd love to keep
> Every movement carried so much mystique
> I knew right then I'd carry on, to you I knew my heart belonged"

Ah.... Não podia esquecer. Até o doutorado ☺

Sumário

LISTA DE ABREVIATURAS E SIGLAS vi

LISTA DE FIGURAS viii

LISTA DE TABELAS ix

RESUMO x

ABSTRACT xi

1 INTRODUÇÃO 1
1.1 Objetivos 3
1.2 Organização 4

2 FUNDAMENTAÇÃO TEÓRICA 5
2.1 Sistemas Paralelos/Distribuídos 5
2.1.1 Classificação 6
2.1.2 Benefícios e desvantagens dos sistemas distribuídos/ paralelos 7
2.1.3 Sistemas paralelos 8
2.1.4 Sistemas distribuídos 11
2.2 Comunicação entre Processos 13
2.2.1 Memória Compartilhada 14
2.2.2 Troca de Mensagens 15
2.3 Conclusões 17

3 TRABALHOS RELACIONADOS 19
3.1 Aplicações Distribuídas Baseadas Em Trabalho Cooperativo com JavaSpace 19
3.2 SimRWA-D: Uma Abordagem Distribuída para Simulação de Redes Ópticas Transparentes 21
3.3 Distributed scientific computing in Java: observations and recommendations 22
3.4 Ibis: an efficient Java-based grid programming environment 22
3.5 Conservative Simulation using Distributed-Shared Memory 23
3.6 A Model-Driven Approach to Job/Task Composition in Cluster Computing 23
3.7 PVM - Parallel Virtual Machine 24
3.8 An automatic distributed simulation environment 25
3.9 Conclusões 25

4 *EASYP - EASY PERFORMANCE* 28
4.1 Proposta 29
4.1.1 Serviços 32
4.1.2 Ambiente 34
4.1.3 Ambiente multiprocessados 34
4.1.4 Modelo componentes 35
4.1.5 Camada de abstração 38
4.1.6 Política de despacho 40
4.1.7 Usabilidade 42
4.2 Aspectos Arquiteturais do Framework 43
4.3 Validação da Utilização 45
4.3.1 Inicialização do *EasyP* 45
4.3.2 Adaptação do código de processamento 47
4.4 Conclusões 49

5 ESTUDO DE CASO 51
5.1 Análise do Desempenho 51
5.1.1 Hipóteses 52
5.1.2 Analise do experimento 53
5.1.3 Intervalo de confiança 54
5.1.4 Execução do experimento 55
5.2 Conclusões 58

6 CONCLUSÃO 61
6.1 Contribuições 61
6.2 Trabalhos Futuros 62
6.3 Comentários Finais 64

REFERÊNCIAS 66

Lista de Abreviaturas e Siglas

AMD	Advanced Micro Devices
API	Application Programming Interface - Interface de Programação de Aplicativos
CERN	European Organization for Nuclear Research
CN	Computational Neighborhood
CORBA	Common Object Request Broker Architecture
CPU	Central Processor Unit - Unidade Central de Processamento
DBC	Desenvolvimento Baseado em Componentes
E/S	Entrada e Saída
EasyP	Easy Performance
EJB	Enterprise JavaBeans
GHz	Giga Hertz
IBM	International Business Machines
JDK	Java Development Kit - Kit de Desenvolvimento Java
JNI	Java Native Interface
LAN	Local Area Network - Rede em área local
MIMD	Multiple Instruction Multiple Data - Múltiplas Instruções Múltiplos Dados
MISD	Multiple Instruction Single Data - Múltiplas Instruções Únicos Dados
NASA	National Aeronautics and Space Administration
OMG	Object Management Group
PVM	Parallel Virtual Machine - Máquina Virtual Paralela
RMI	Remote Method Invocation - Invocação Remota de Método
RPC	Remote Procedure Call - Chamada Remota de Métodos
SHA1	Secure Hash Algorithm
SIMD	Single Instruction Multiple Data - Única Instrução Múltiplos Dados
SISD	Single Instruction Single Data - Única Instrução Únicos Dados

SMP	Symmetric Multiprocessing - Múltiplo Processamento Simetrico
SPaDES/Java	Structured Parallel Discrete-event Simulation in Java
WAN	Wide Area Network - Rede em área de longa distância

Lista de Figuras

Figura 2.1: Gráfico comparativo com a energia dissipada de alguns processadores 10

Figura 4.1: Serviços que o *framework* pode prover dentro do ambiente. 32
Figura 4.2: Exemplo de ambiente onde temos uma máquina com o papel de Mestre (M) e três máquinas exercendo o papel de Escravas (máquinas a, b e c) 33
Figura 4.3: Imagem representando dois momentos distintos no processamento de um aplicativo. No momento *(a)* temos três máquinas conectadas ao *Master*. No momento *(b)*, podemos visualizar a inserção de três novas máquinas ao ambiente, totalizando seis máquinas conectadas ao *Master* 35
Figura 4.4: Fluxo de atividades do envio de um componente para ser executado remotamente. 38
Figura 4.5: Arquitetura do framework proposto 39
Figura 4.6: O *framework* permite a diversidade de protocolos de comunicações utilizados ao mesmo tempo. 39
Figura 4.7: Ambiente de processamento heterogêneo. 41
Figura 4.8: Diagrama UML de sequência do *framework* proposto. 44
Figura 4.9: Diagrama UML de classes do *framework* proposto. De preto está representada a classe externa do *framework* proposto.. 45

Figura 6.1: A proposta atual comparada com a proposta de modificação futura. Ao invés de enviar componentes e um mesmo conjunto de dados para máquinas diferentes, enviar um único componente e apenas uma única vez o conjunto de dados, evitando o re-envio dos dados pela rede. 63
Figura 6.2: Proposta atual de comunicação centralizada do *EasyP* em (a) e a nova proposta (b), onde um escravo pode se comunicar diretamente com o segundo escravo. 64

Lista de Tabelas

Tabela 2.1: Comparação entre sistemas paralelos e sistemas distribuídos [18] . 7
Tabela 2.2: Consumo de Energia dos Processadores 10

Tabela 3.1: Comparativo entre as características de cada estudo realizado previamente e a nossa proposta. 27

Tabela 5.1: Ambientes utilizados nos experimentos realizados. 52
Tabela 5.2: Resultado da execução da versão modificada do simulador proposto por Durães. Esta versão processa o simulador de maneira sequencial. 56
Tabela 5.3: Resultados na medição de tempo do simulador original proposto por Durães utilizando seu sistema de distribuição com o protocolo Java RMI. Tempo em minutos 57
Tabela 5.4: Resultados na medição de tempo do simulador modificado utilizando o *framework EasyP* e o protocolo de comunicação RMI. Tempo em minutos 58
Tabela 5.5: Resultados na medição de tempo do simulador modificado utilizando o *EasyP* e o protocolo de comunicação RMI. 58
Tabela 5.6: Valor da distribuição $|t_0|$ para o teste da hipótese nula. 58
Tabela 5.7: Intervalo de confidência para a hipótese $H0_3$ 59
Tabela 5.8: Amostras de tempo do simulador modificado utilizando o *framework* proposto. Tempos coletados utilizando o 1º ambiente de simulação. Tempo em minutos 59
Tabela 5.9: Amostras de tempo do simulador modificado utilizando o *framework* proposto. Tempos coletados utilizando o 2º ambiente de simulação. Tempo em minutos 60
Tabela 5.10: Amostras de tempo do simulador modificado utilizando o *framework* proposto. Tempos coletados utilizando o 3º ambiente de simulação. Tempo em minutos 60
Tabela 5.11: Resultado da avaliação da comparação da utilização do *EasyP* com Java RMI e outros protocolos de comunicação. 60

Resumo

Aplicações que necessitam de muito poder computacional podem necessitar que seu resultado seja obtido dentro de um período de tempo determinado, caso contrário a resposta pode não ser mais válida. Uma forma de obter tais respostas dentro de um intervalo válido é utilizar máquinas de alto poder computacional, as quais nem sempre estão disponíveis, seja devido ao seu alto custo ou devido ao compartilhamento da mesma com outros pesquisadores. Uma alternativa é a utilização de diversas máquinas com menor poder computacional e maior disponibilidade. No entanto, para usufruir corretamente deste tipo de ambiente a aplicação precisa ser adaptada para tal, o que não é uma atividade simples. Estes ambientes com diversas máquinas podem ser bastante heterogêneos, tornando a adaptação uma atividade bastante complexa. O objetivo deste trabalho é propor um framework que abstraia esta complexidade de programar uma aplicação para executar em um ambiente distribuído e heterogêneo, diminuindo o esforço. Além disso, foram realizadas analises de desempenho mostrando os benefícios de utilizar o framework.

Palavras-chave: Sistemas distribuídos, suporte ao usuário, sistemas multi-núcleo, sistemas paralelos.

Abstract

Complex applications may need the result in a specific time, otherwise it will not be valid anymore. In order to produce this result in a reasonable time, those applications must be processed in powerfull machines. However, such machines are very expensive and not always available. An alternative is using several less powerfull and less expensive machines, which can be used as a single resource. However, to use all the power of this environment, the application need to be designed considering this kind of environment what is not an easy task. Hence, the main objective of this study is to define a framework that can create an abstraction for those problems, making it easier to design applications for distributed environments. Futhermore, some performance analysis had been made.

Keywords: distributed systems, user support, multi-core systems, parallel systems.

Capítulo 1

Introdução

Quanto maiores são as dificuldades
a vencer, maior será a satisfação.

Marcus Tullius Cicero

O desenvolvimento de aplicações computacionais complexas podem requerer horas ou até mesmo dias para realizar toda a computação necessária. Inicialmente, toda vez que existia a necessidade de uma aplicação ser mais complexa e demorar mais do que o possível para realizar a computação, o desenvolvedor da aplicação colocava esta para ser processada em uma máquina de maior poder computacional.

Porém, estamos chegando no limite físico da evolução das máquinas computacionais, não sendo mais tão simples aumentar o poder de processamento de uma máquina computacional. Inicialmente, ao necessitar de uma máquina com maior poder computacional, o arquiteto criava uma nova máquina com uma frequência do relógio interno maior, fazendo com que os dados trafegassem mais rápidos, com isso, sendo possível realizar mais rapidamente o processamento das instruções. No entanto, isto não está mais sendo possível, devido à interferências eletromagnéticas [1] e geração de calor [6].

Desta forma, a solução encontrada ao passar dos anos, foi a paralelização das funcionalidades do processador. Para isto, houve a duplicação de recursos, permitindo que o computador realizasse mais de uma operação ao mesmo tempo. Essa capacidade de realizar mais de uma operação ao mesmo instante permitiu que a frequência do relógio interno da máquina fosse reduzida a níveis que não causassem mais interferências. Além desta duplicação de recursos em um único computador, devido à redução do custo da aquisição de novas máquinas, as instituições atualmente possuem laboratórios com diversas máquinas, permitindo que estas sejam utilizadas como recursos de processamento em abundância para a realização da computação necessária pelos aplicativos.

Apesar de atualmente haver esta duplicação de recursos de processamento, apenas processar um aplicativo antigo em um ambiente com recursos adicionais não permitirá que este aplicativo seja processado em um tempo menor. Para ocorrer um melhor aproveitamento dos recursos duplicados, o aplicativo deve estar preparado, principalmente possibilitando a execução de instruções em paralelo. No entanto,

[1] A redução da escala de construção dos processadores está chegando ao limite, as trilhas elétricas do circuito estão captando ondas eletromagnéticas geradas por outras trilhas, isto devido a proximidade entre estas.

o desenvolvimento de aplicações que possam usufruir destes recursos duplicados necessita lidar com alguns problemas inerentes a esta duplicação de recursos, como a concorrência e comunicação. Além de problemas que surgem com o gerenciamento desta duplicação de recursos, o aplicativo do usuário se torna mais complexo, prejudicando a manutenabilidade do código do aplicativo. Acrescentando a estes problemas, o fato de que o desenvolvimento deste gerenciamento da duplicação distribuída necessitará de um certo tempo até que esteja pronto para ser utilizado, atrasando ainda mais o cronograma do desenvolvimento do aplicativo do usuário.

Devido a estes problemas no desenvolvimento, nem sempre os aplicativos são desenvolvidos para usufruírem de recursos computacionais duplicados. O que ocorre geralmente são aplicativos não desenvolvidos corretamente que subutilizam os recursos da máquina, isto pelo simples fato da facilidade de se desenvolver aplicativos chamados *sequenciais*, que realizam todo o seu processamento em apenas uma unidade de processamento [8]. Desta forma, a existência de uma ferramenta que diminua esta complexidade, permitirá que mais aplicativos sejam desenvolvidos com a capacidade de usufruir da duplicação dos recursos computacionais.

Assim, um pesquisador que possui sua ferramenta de simulação bastante complexa que é completamente sequencial porém com a capacidade de ser paralelizada, e acrescentando o fato de possuir diversas máquinas disponíveis, poderia usufruir destas máquinas para realizar o processamento paralelo de sua simulação. Permitindo que ele obtenha mais rapidamente os resultados de sua pesquisa e provavelmente aprimorando ainda mais sua linha de pesquisa, pois poderia obter seus resultados mais rapidamente do que o que se esperava inicialmente, possibilitando inovações tecnológicas mais rápidas.

Devido a estes benefícios, diversos estudos [38, 56] já foram realizados previamente em busca de uma ferramenta que possibilite o desenvolvimento de aplicações distribuídas mais facilmente. Porém, sempre existe uma limitação nestes estudos. Até pelo fato de ser um campo de estudo muito vasto, esta limitação sempre existirá. No entanto, algumas destas limitações prejudicam o desenvolvimento e a execução da aplicação. Como por exemplo, não tornando tão simples a utilização das ferramentas propostas, deixando ainda alguma dificuldade na utilização destas.

Alguns estudos [11, 9], além de não abstraírem o suficiente os problemas, limitam a utilização do usuário, não permitindo que esta seja executada em qualquer conjunto de máquinas ou possa utilizar o seu próprio escalonamento de processos. Estas limitações, prejudicam não só no desempenho final da aplicação do usuário, mas também pode limitar de tal forma que não seja possível utilizar a ferramenta em conjunto com o aplicativo do usuário. Um motivo para isto, é o fato de que se a ferramenta limitar a um ambiente estático, e o usuário só tenha uma máquina disponível a todo tempo, mas por curtos períodos de tempo, tenha n outras máquinas, não será possível a utilização destas novas máquinas durante estes períodos, sendo então desnecessário o uso da ferramenta de distribuição.

Devido a estas observações listadas, este trabalho procura desenvolver uma ferramenta que:

- permita uma comunicação entre os processos executados em cada um dos recursos computacionais duplicados;
- dê suporte ao usuário no desenvolvimento de aplicações distribuídas, abstraindo dificuldades de mais baixo nível;

- permita uma distribuição transparente das partes paralelizáveis do software do usuário, sendo estas partes indicadas pelo próprio usuário;
- possibilite ao usuário o processamento de seu aplicativo em qualquer ambiente de máquinas;
- permita a troca do protocolo de comunicação utilizado para troca de informações entre os processos que processam remotamente os pedaços do software do usuário;
- possibilite a utilização de um ambiente de máquinas computacionais mais dinâmico, permitindo o acréscimo ou remoção de núcleos de processamento enquanto o aplicativo do usuário está sendo processado;
- possibilite ao usuário definir o próprio escalonamento de processos utilizado durante o processamento do aplicativo.

Desta forma, a ferramenta proposta, *EasyP* (*Easy Performance*), atuará no domínio de *frameworks* permitindo ao usuário a utilização do *EasyP* para a realização de tarefas mais complexas. A abstração destas tarefas mais complexas, permitirá ao usuário tratar com o *framework* em nível mais alto, mais simples de ser manipulado, deixando para o próprio *framework* a responsabilidade de lidar com todas as tarefas complexas que serão executadas em baixo nível. Quanto mais simples a interface do *framework* com o usuário, mais fácil será a utilização deste, permitindo uma maior disseminação de seu uso, com isto, vários pesquisadores poderão obter o resultado de seus estudos mais rapidamente e de maneira mais simples.

Porém, toda facilidade tem seu preço. Pelo fato de estar abstraindo diversos problemas, permitindo que o usuário não os trate em baixo nível, acontecerá necessariamente alguma perda de desempenho. Ao permitir que o usuário interaja apenas com a interface do *EasyP* em alto nível, é de se esperar que ocorra alguma perda de desempenho no processamento, pois ocorrerão mais operações computacionais até realizar realmente a funcionalidade que foi requisitada. No entanto, esta perda deverá ser analisada para verificar se realmente é útil a utilização da proposta deste *framework*. Pois a utilização do *EasyP*, permitirá uma versão do aplicativo com um melhor desempenho, quando comparado com a versão sequencial, porém quando comparado com uma versão desenvolvida em baixo nível para ser utilizada em ambientes com recursos duplicados, existirá uma perda de desempenho considerável. No entanto, o custo de ter esta versão em baixo nível será muito maior do que ter um aplicativo um pouco mais lento desenvolvido mais facilmente e com menor custo utilizando o *EasyP*.

1.1 Objetivos

Esta dissertação tem como principal objetivo o desenvolvimento de uma ferramenta capaz de abstrair algumas dificuldades que aparecem no desenvolvimento de aplicações distribuídas. Para isto, é necessário o estudo das abordagens previamente existentes, identificando suas falhas e seus pontos fortes, resultando em uma ferramenta que una o máximo dos pontos fortes destas abordagens e o mínimo destas falhas. Como objetivos específicos podemos citar:

- o levantamento bibliográfico dos estudos realizados com o intuito de facilitar o desenvolvimento de ferramentas que dêem apoio ao usuário no desenvolvimento de aplicações distribuídas;

- o desenvolvimento do *EasyP* (*Easy Performance*), uma ferramenta capaz de realizar a abstração necessária para processar o aplicativo do usuário em uma grande gama de ambientes, com o mínimo de esforço gasto na sua utilização, como também realizar o processamento paralelo de partes do aplicativo do usuário. Porém, a utilização desta ferramenta não permite obter o melhor desempenho possível de um ambiente paralelo, devendo assim ser utilizada apenas em aplicativos que se deseja um ganho de desempenho, mas não necessariamente o melhor;

- permitir ao usuário alguns diferenciais, como a definição de sua própria política de despacho de seus componentes, permitindo assim um possível balanceamento de carga na utilização do *EasyP*.

1.2 Organização

No restante desta dissertação será detalhado todo o desenvolvimento do estudo realizado. Primeiramente, no Capítulo 2, serão discutidos algumas características de ambientes e os tipos de comunicação entre processos existentes, ressaltando suas dificuldades e seus benefícios. Após este embasamento teórico a respeito do tema, será levantada a discussão sobre a pesquisa realizada, apresentando alguns dos trabalhos que foram levados em consideração para o estudo realizado, esta discussão estará no Capítulo 3. Com a discussão sobre os trabalhos que guiaram este estudo, será apresentada a idéia da proposta, mostrando as características da idéia do *EasyP* além de alguns detalhes de sua implementação, encontrados no Capítulo 4. Esta explicação da idéia proposta será fechada com o Capítulo 5, onde estará presente um estudo de caso e mais alguns detalhes de utilização do *EasyP*. Encerrando a dissertação, existe o Capítulo 6 que abordará, as contribuições deste estudo, os possíveis trabalhos futuros realizados sobre a proposta do *EasyP* e uma discussão geral sobre as conclusões desta dissertação.

Capítulo 2

Fundamentação Teórica

> Cada um tem de mim exatamente o que cativou, e cada um é responsável pelo que cativou, não suporto falsidade e mentira, a verdade pode machucar, mas é sempre mais digna.
>
> Charles Chaplin

> Vocês riem de mim por que sou diferente. E eu morro de rir de vocês, por que são todos iguais.
>
> Bob Marley

Ao longo deste capítulo será exposto um resumo do levantamento bibliográfico realizado sobre os temas de sistemas paralelos/distribuídos e comunicação entre processos em aplicativos distribuídos. Este estudo se fez necessário para obter embasamento teórico suficiente sobre o objetivo principal desta dissertação, a definição de uma ferramenta para facilitar o desenvolvimento de aplicações distribuídas.

2.1 Sistemas Paralelos/Distribuídos

Antigamente quando uma aplicação computacional exigia um maior desempenho, o projetista da aplicação procurava executá-la em uma máquina com um maior poder computacional. Esta mentalidade ajudou que a lei de Moore [42] continuasse válida até os dias atuais. Lei esta que indica que o poder dos processadores dobrará a cada 18 meses.

Quando havia a necessidade de uma máquina com maior poder de processamento, os pesquisadores resolviam aumentar a freqüência de operação do processador da máquina. Porém, segundo a teoria da relatividade de Einstein [12], um sinal elétrico não pode propagar mais rápido do que a velocidade da luz. Isto implica que para construir um computador que trabalhe em uma freqüência em torno dos 10GHz, os sinais elétricos não poderão percorrer mais do que 2cm em um único ciclo de

relógio. A solução para este problema foi encontrar uma maneira de aumentar o nível de integração dos circuitos internos ao processador, chegando a termos em 2007 o lançamento de processadores produzidos na escala de 45 nanômetros [29].

No entanto, o aumento do nível de integração aplicado na construção dos processadores implica em um grande problema: o alto consumo de energia, tendo como conseqüência a produção de uma grande quantidade de calor que deve ser dissipada por um equipamento extremamente pequeno. Um processador moderno pode gerar mais calor por centímetro quadrado do que um ferro de passar roupa e só consegue operar em seu estado de segurança devido a evolução nos materiais e nas técnicas utilizadas no resfriamento deste.

Outro fato que dificulta o constante aumento do poder computacional dos processadores é que os periféricos não acompanharam a velocidade com que os processadores evoluíram, retardando a resposta da computação realizada. Por exemplo, um processador que opere a uma freqüência em torno dos 3.6GHz pode executar uma instrução a cada $277\text{x}10^{-12}$ segundos mas o sistema pode exigir até 400 vezes mais tempo para extrair a informação da memória principal [24].

Uma maneira de obter um ambiente com um grande poder computacional, sem defrontar estes problemas encontrados em máquinas de alto poder de processamento, é a utilização de diversas unidades de processamento, que individualmente não necessitam ter um grande poder computacional, porém podem realizar várias tarefas de forma paralela. O comportamento de aplicações paralelas foi classificado por Flynn[14] em 1972, embora já tenha se passado mais de 35 anos desta classificação, ela ainda é uma das mais aceitas na comunidade científica. Segundo Flynn, estes sistemas podem ser classificados como: **(i)** *única instrução, único dado* (SISD - *Single Instruction, Single Data*): uma única unidade de processamento executa uma única seqüência de instruções usando apenas um único conjunto de dados; **(ii)** *única instrução, múltiplos dados* (SIMD - *Single Instruction, Multiple Data*): um mesmo conjunto de instruções é executado em diversas unidades de processamento, onde cada unidade executará sobre um conjunto distinto de dados; **(iii)** *múltiplas instruções, único dado* (MISD - *Multiple Instruction, Single Data*): cada unidade de processamento em um conjunto executam uma seqüência de instruções distintas em um mesmo conjunto de dados; e **(iv)** *múltiplas instruções, múltiplos dados*(MIMD - *Multiple Instruction, Multiple Data*): um conjunto de unidades de processamento executam instruções distintas sobre um conjunto distinto de dados.

2.1.1 Classificação

Ambientes com mais de uma unidade de processamento podem ainda ser classificados, de acordo com Fujimoto [18], em dois grandes grupos: **(i)** sistemas paralelos; e **(ii)** sistemas distribuídos. Sistemas paralelos são sistemas computacionais que se encontram fisicamente próximos, tendo suas unidades de processamento contidas em um mesmo compartimento (invólucro do processador ou um mesmo gabinete de máquina), geralmente possuindo uma homogeneidade entre estas unidades de processamento. Sistemas paralelos ainda utilizam uma rede de comunicação dedicada com baixa latência, chegando a no máximo gastar algumas centenas de microssegundos.

Sistemas distribuídos são sistemas computacionais que encontram-se fisicamente mais distantes, podendo estar em uma única edificação ou espalhados por todo

Tabela 2.1: Comparação entre sistemas paralelos e sistemas distribuídos [18]

	Sistemas paralelos	Sistemas distribuídos
Extensão física	Sala de máquinas	Podem ser localizados em apenas um prédio ou estar distribuídos por todo o globo
Processadores	Homogêneos	Geralmente heterogêneos
Comunicação	Switchs customizados	Redes comerciais LAN ou WAN
Latência da comunicação	Menos de 100 microsegundos	Centenas de microsegundos até alguns segundos

o mundo. Ao contrário das unidades de processamento que formam os sistemas paralelos, as que formam um sistema distribuído podem funcionar isoladamente do resto do sistema computacional, já que cada uma desta unidade é vista como um sistema computacional completo [1]. Um resumo desta comparação entre sistemas paralelos Vs. sistemas distribuídos pode ser visto na Tabela 2.1.

Porém, recentemente esta distinção entre estes dois tipos de sistema computacional tem se tornado confusa. Isto, de acordo com Fujimoto [18], é devido a inserção do conceito de *Network of Wortkstations* (Rede de estações de trabalho), que se refere a *clusters* de máquinas com um alto poder computacional, mas com características de ambos os grupos de sistemas computacionais com mais de uma unidade de processamento. Por exemplo, é formado por máquinas computacionais completas, o que caracterizaria como sendo um sistema distribuído, porém conseguem se comunicar em um intervalo semelhante ao gasto em sistemas paralelos, isto devido a utilização de tecnologias de comunicação mais avançadas e específicas para o caso, ao invés de utilizarem protocolos complexos.

2.1.2 Benefícios e desvantagens dos sistemas distribuídos/ paralelos

Comparado à aplicações sequenciais, existem diversas vantagens que surgem ao desenvolver uma aplicação preparada para ser executada em um ambiente computacional paralelo/distribuído. Entre estas vantagens podemos destacar [17, 35, 18]:

- *Desempenho*: existe um limite para processos sendo executados em uma mesma unidade de processamento. Para superar este limite, é necessário usar outras unidades de processamento no ambiente. A inserção de outras unidades de processamento poderá permitir que o tempo total de execução da aplicação diminua;

- *Custo*: em um sistema com mais de uma unidade de processamento existe a redução da relação desempenho/custo;

[1]Sistemas computacionais completos são unidades de processamento que possuem sua própria memória e seus próprios periféricos, podendo assim, funcionar isoladamente sem necessitar de nenhum outro recurso computacional.

- *Escalabilidade*: uma aplicação distribuída pode ser utilizada, sem modificações significativas, em n unidades de processamento, porém nem sempre indicará que este aumento de unidades de processamento resultará em um aumento no desempenho da aplicação;
- *Integração*: permite que aplicações de diferentes empresas se comuniquem;
- *Tolerância a falhas*: em uma aplicação monolítica, se a unidade de processamento falhar, a aplicação só volta a executar após o reinício do processo. Já em uma aplicação distribuída se uma das unidades de processamento falhar, uma outra unidade pode assumir este lugar sem a percepção disto no resultado final;
- *Acesso a recursos remotos*: em uma aplicação distribuída, uma unidade de processamento pode ter acesso a recursos computacionais que se encontrem geograficamente distantes.

Porém, a utilização de sistemas paralelos/distribuídos não apresenta apenas vantagens, existem algumas desvantagens que se tornam presentes ao utilizar aplicações nestes ambientes [17], como:

- *Latência*, a comunicação entre os processos em ambientes com mais de uma unidade de processamento não é realizada diretamente, existe um meio para ser feita esta comunicação. Quanto maior a distância física entre as unidades, maior será a latência na comunicação;
- *Sincronização*, por estar dividindo a aplicação em diversos processos, é necessário garantir que uma ordem de execução seja obedecida, para isto se faz necessário o uso de métodos de sincronização de processos;
- *Falha parcial*, quanto mais tempo uma aplicação permaneça executando e quanto mais unidades de processamento estejam inseridas no ambiente computacional, mais provável será a ocorrência de falhas;
- *Estado global*, devido a quantidade de máquinas existente no ambiente computacional, dificilmente todas as máquinas estarão processando o mesmo estado do sistema.

2.1.3 Sistemas paralelos

Sistemas paralelos são sistemas computacionais formados por mais de uma unidade de processamento [2], sendo essas unidades de processamento homogêneas. Adicionando a característica de que estas máquinas estão localizados fisicamente próximas além de utilizarem uma rede de comunicação customizada fazendo com que a latência seja muito pequena, no máximo na ordem de algumas centenas de milisegundos.

A idéia envolvida na utilização de mais de uma unidade de processamento em uma mesma máquina, é a replicação de tarefas em recursos de processamento replicadas. Desta forma, duas ou mais tarefas podem ser executadas ao mesmo

[2] *Unidade de processamento* é considerado um termo geral que representa um circuito eletrônico capaz de realizar uma computação em um determinado dado.

tempo, sem haver necessidade que estas tarefas compitam por recursos de processamento. Porém, por estarem em uma mesma máquina, estas unidades de processamento podem competir por recursos de E/S (Entrada/Saída).

Estes ambientes podem ser divididos em três grandes grupos: **(i)** ambientes multiprocessados, formados por máquinas computacionais que possuem mais de um processador físico [3]; **(ii)** ambientes multi núcleo, formados por processadores que possuem mais de um núcleo de processamento [4]; e **(iii)** *clusters*, que são aglomerados de máquinas organizadas e se comunicando por uma rede de baixa latência.

2.1.3.1 Sistemas multiprocessados

Com a constante procura por mais desempenho e os limites físicos de construção de processadores mais rápidos cada vez mais próximo, os fabricantes introduziram o conceito de máquinas com mais de um processador[51]. Inicialmente essas máquinas possuíam processadores específicos para as principais funções do sistema, como por exemplo a execução do sistema operacional ou um de seus subsistemas, não permitindo que estes processadores executassem outras aplicações como aplicações do usuário. Porém, esta abordagem foi abandonada rapidamente pelo fato de que estes processadores específicos passavam a maior parte do tempo trabalhando enquanto que os processadores de propósito geral ficavam sem uso.

Surgiram então as máquinas de processadores simétricos. Estas máquinas são formadas por um conjunto de processadores, geralmente idênticos, que possuem a mesma capacidade de computação. A diferença é que qualquer tarefa a ser executada pela máquina pode ser executada por qualquer processador. Assim, esta abordagem é denominada SMP - *Symmetric Multiprocessing*.

Estes processadores compartilham os seus recursos, como memória principal e conexões de entrada e saída. Os processadores conectados entre si através de um barramento ou outro tipo de interconexão, de forma que o tempo de acesso à memória seja semelhante para todos os processadores.

2.1.3.2 Sistemas multi núcleo

A utilização de máquinas com mais de um processador resolve a limitação do número de processos possíveis em apenas uma máquina. Porém, ao colocar mais de um processador operando em uma alta freqüência em um mesmo compartimento, faz com que a produção de calor, o consumo de energia e a relação custo/desempenho desta máquina sofram um aumento bastante considerável quando comparado ao ganho de desempenho proveniente da adição de novos processadores.

Atualmente surgiu uma nova solução para o ganho de desempenho sem haver a necessidade de aumentar a freqüência de operação nem o nível de integração dos processadores. Ao invés de seguir a mesma linha de pesquisa utilizada durante anos, estão fazendo o caminho inverso, diminuindo a freqüência de operação e o nível de integração do núcleo do processador. Desta forma conseguem núcleos de

---

[3] O termo *processador* é utilizado para representar o componente físico que realiza o processamento da máquina, podendo possuir um ou mais núcleos de processamento dentro de seu invólucro.

[4] O termo *núcleo de processador* é utilizado como sendo um circuito eletrônico capaz de realizar as operações aritméticas, lógicas e de movimentação de dados. Geralmente, o termo, é visto como a CPU (*Central Processor Unit* - Unidade Central de Processamento).

Tabela 2.2: Consumo de Energia dos Processadores

Processador	Freqüência do Núcleo (MHz)	Consumo (Watt)
AMD Sempron 2800+	1600	62
AMD Athlon 3000+	1800	67
AMD Athlon 3200+	2000	89
AMD Athlon X2 Dual-Core 3600+	1900	65
AMD Phenom X4 Quad-Core	2100	65

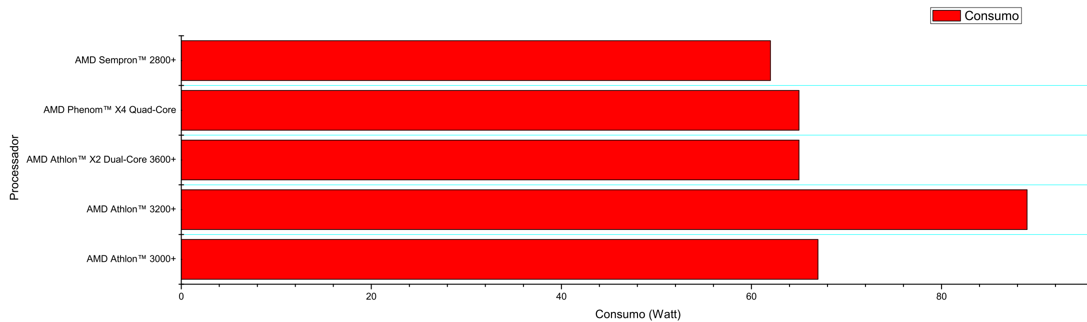

Figura 2.1: Gráfico comparativo com a energia dissipada de alguns processadores

processamento com baixo consumo de energia e com pouca geração de calor. Assim, para conseguir um aumento no desempenho estão adicionando vários destes núcleos em uma única pastilha de processador, podendo gerar ganhos de 30 a 90 por cento sem aumentar, na mesma proporção, a geração de calor e o consumo de energia [6]. A Tabela 2.2 e a Figura 2.1 mostram um pequeno histórico da relação desempenho Vs. consumo de energia dos processadores da AMD [2], nos quais podemos ver que o consumo aumentou enquanto a frequência aumentava em processadores com apenas um núcleo e passou a ficar mais estável em processadores com mais de um núcleo. Apesar de esta tecnologia comercialmente ser algo novo, a nível de pesquisa não se pode dizer que foi descoberta recentemente. Em 1989, Gelsinger [22] previu que na virada do milênio teríamos processadores com mais de um núcleo de processamento.

A grande vantagem desta tecnologia é o fato de conseguir um ganho no desempenho ao utilizar paralelismo em suas aplicações sem aumentar o consumo de energia, dissipação de calor ou espaço físico [1]. Basicamente, a utilização de múltiplos núcleos não faz com que uma máquina opere mais rápido, faz apenas com que os recursos de processamento sejam duplicados, permitindo que mais operações sejam realizadas ao mesmo tempo. Além disto, estes núcleos podem ser vistos como diversos processadores trabalhando em conjunto, porém estão presentes em uma única pastilha de processamento, isto permite uma comunicação entre estes núcleos de processamento muito mais rápida quando comparada a sistemas multiprocessados, em que cada processador está isolado em sua própria pastilha de processamento [28]. Um problema desta abordagem é que aplicações que necessitam de um alto poder de processamento mas não estão preparadas para utilizar mais de uma unidade de processamento não serão beneficiadas quando executadas em máquinas com mais de um núcleo.

2.1.3.3 Cluster

Soluções de alto desempenho como máquinas multiprocessadas ou com múltiplos núcleos, são soluções comercialmente caras, sendo soluções inviáveis para pequenas organizações que necessitam de um ambiente computacional de alto desempenho, porém não possui recursos financeiros para máquinas poderosas [20]. Visto isso, uma solução criada foi a utilização de sistemas computacionais completos (denominados *nós*), financeiramente mais baratos, interligados por uma rede de comunicação, proporcionando ao usuário a visão de um recurso computacional unificado [51].

Em geral, *clusters* podem ser simples, sendo formados por apenas duas máquinas usando qualquer tipo de conexão, ou bastante complexos, formados por milhares de máquinas utilizando uma rede dedicada de baixa latência. Apesar de poder utilizar máquinas de alto desempenho interligadas por uma rede de baixa latência, o mais comum ainda hoje são *clusters* derivados do *cluster Beowulf* [5], proposto pela NASA (*National Aeronautics and Space Administration*) em 1994 com a finalidade de processar as informações espaciais que a entidade recolhia. Este tipo de *cluster* possui as seguintes características:

- a conexão dos nós é realizada através de uma rede privada de comunicação, podendo ser utilizada uma rede *Ethernet*;
- a comunicação é feita utilizando passagem de mensagens;
- existe um nó responsável por controlar todo o *cluster*, principalmente para distribuição de tarefas e processamento;
- o sistema operacional dos nós é baseado no Linux;
- pode-se utilizar computadores comuns, inclusive utilizando computadores obsoletos.

De acordo com Fox [16] uma configuração em *cluster* apresenta três benefícios: **(i)** *escalabilidade*, um *cluster* pode ser formado por diversas máquinas, sendo cada uma delas uma máquina com mais de uma unidade de processamento. Ainda pode ser desenvolvido de maneira que seja possível adicionar ou remover máquinas sem a necessidade de nenhuma grande mudança na configuração do ambiente; **(ii)** *alta disponibilidade*, como cada nó do *cluster* é um sistema completo independente, a falha de um nó não significa que todo o sistema falhou, podendo o resto do sistema continuar executando normalmente; e **(iii)** melhor relação custo/desempenho, devido à facilidade de construir *clusters* com sistemas completos independentes, existe a possibilidade da utilização de algumas máquinas de baixo custo formando um ambiente com poder computacional equivalente a uma máquina de grande porte, porém com um custo muito inferior.

2.1.4 Sistemas distribuídos

Kshemkalyani [35] define sistemas distribuídos como sendo: *"um conjunto de entidades independentes que cooperam entre si para solucionar um problema que não seja possível solucionar individualmente"*. Desta forma, pode se utilizar estes ambientes com mais de uma unidade de processamento como sendo uma máquina com poder computacional total, maior do que uma máquina com uma única unidade

de processamento. Porém, para usufruir deste grande poder computacional, o sistema a ser executado neste ambiente deve estar preparado para utilizar as unidades de processamento disponíveis.

Como exposto anteriormente, um ambiente distribuído possui a vantagem de poder diminuir o tempo de processamento com relação à executar a aplicação em apenas uma unidade de processamento, porém possui suas desvantagens, como o *overhead* devido à comunicação entre as unidades de processamento. Desta forma, Shell [50] cita as 3 principais características de um ambiente distribuído considerado ideal:

- *Desempenho* - todo sistema com mais de uma unidade de processamento insere um *overhead* significativo em relação à aplicação equivalente seqüencial, que seria a comunicação entre os processos, quanto maior a relação entre o tempo de comunicação e o tempo de processamento útil, menor será a eficiência desta aplicação paralela;

- *Escalabilidade* - a principal meta de sistemas com mais de uma unidade de processamento é a escalabilidade linear, o tempo de processamento deve ser inversamente proporcional à quantidade de unidades de processamento envolvidos. No entanto, os sistemas com mais de uma unidade não conseguem esta escalabilidade linear, devido ao custo da comunicação, sendo todos sub-linearmente escaláveis;

- *Manutenabilidade*, sistemas que procuram um melhor desempenho e ser escalável, sacrificam a sua manutenabilidade. Sistemas deste tipo são difíceis de manter e possuem sua vida útil reduzida.

Como os ambientes distribuídos são influenciados diretamente pelo custo de comunicação existente entre as unidades de processamento presentes no ambiente, Tanenbaum [54] classificou estes ambientes de acordo com a sua latência na comunicação: **(i)** *multiprocessadores*, sistemas computacionais nos quais diversas unidades de processamento se comunicam através de memória compartilhada. A comunicação nestes sistemas ocorre em alguns nanosegundos; **(ii)** *multicomputadores*, composto por diversas unidades de processamento onde cada uma possui sua própria memória e estão interligados por uma conexão de alta velocidade. Possuindo uma boa velocidade de comunicação, uma mensagem curta pode ser trocada entre as unidades de processamento em alguns microsegundos; e **(iii)** *computadores distribuídos*, que são compostos por diversos conjuntos de sistemas completos, onde cada sistema possui sua própria memória. Por geralmente este tipo de sistema ser conectado utilizando uma conexão lenta, como a própria internet, e estarem geograficamente mais distantes, as mensagens podem ser trocadas em alguns milissegundos a até alguns segundos.

2.1.4.1 Computação em Grade

Clusters computacionais resolvem a questão da necessidade de supercomputadores em organizações sem muitos recursos financeiros para adquirir máquinas poderosas mas que custam milhares de dólares. Porém, ainda assim pode ocorrer que uma organização não possua recursos financeiros suficientes para construir um *cluster*

que lhe provenha poder computacional suficiente. Pensando nestes casos, surgiu a idéia de *grids computacionais*.

Existem diversas definições para *grids* computacionais: **(i)** o CERN - *European Organization for Nuclear Research*, um dos maiores usuários desta tecnologia, define como *"um serviço de compartilhamento de poder computacional e capacidade de armazenamento através da Internet"* [15]; **(ii)** Plaszczak[45] define como sendo *"a tecnologia que permite a virtualização de recursos, disponibilização* on-demand *e compartilhamento de recursos/serviços entre organizações"*; enquanto **(iii)** a IBM [30] define como *"a capacidade de que usuários (ou aplicações clientes) obtenham acesso a recursos computacionais (poder de processamento, armazenamento, dados, aplicações e outros recursos) como necessário com pouco ou sem nenhum conhecimento de onde estes recursos estejam alocados ou quais tecnologias, hardware ou sistemas operacionais estejam sendo utilizados"*.

No geral, podemos definir *Grids* como sendo um conjunto de recursos computacionais geograficamente remotos, interligados por uma rede de comunicação que pode ser privada, pública ou a própria internet. Estes recursos computacionais podem ser simples máquinas, como também máquinas complexas possuindo diversas unidades de processamento, ou até mesmo, podem ser conjuntos de computadores formando um *cluster* ou outros *grids*.

2.2 Comunicação entre Processos

Para realizar o processamento de um aplicativo em um ambiente com mais de uma unidade de processamento e poder usufruir de todo o poder computacional disponível, é necessário que este aplicativo esteja subdividido em parcelas menores de processamento. Estas parcelas são vistas como subprocessos, que na realidade são processos executando em separado com o objetivo de realizar apenas uma parte da computação necessária pelo aplicativo do usuário.

Quando passamos a ter mais de um processo processando tarefas em comum, é necessário que exista uma comunicação entre estes processos para poderem trocar informações, além de permitir a realização de um processamento final para poder integrar todos os resultados de cada um destes subprocessos e produzir apenas um único resultado. Esta comunicação pode ser realizada tanto a nível de processos, como também em um nível mais alto de granularidade, que seria o caso de não termos processos sendo processados em separado, mas um mesmo processo sendo executado em diversas *threads* [54].

No entanto, a comunicação entre as unidades de processamento é uma tarefa complexa que pode acarretar o surgimento de diversos problemas. Bruschi [8] lista alguns destes problemas, por exemplo:

- *concorrência*: diversas unidades de processamento podem tentar alterar ou ler um mesmo dado ao mesmo tempo;
- *sincronização*: uma determinada computação só poderá executar obedecendo uma pré-condição, que está relacionada com a uma outra computação, sendo executada em outra unidade de processamento;
- *protocolo*: para uma unidade de processamento compreender os dados enviados por uma unidade remota, ambas as unidades devem estabelecer um protocolo

de comunicação, definindo como se dará esta troca de informações.

A realização da comunicação entre processos tem evoluído com o passar do tempo. As duas idéias mais simples para a realização desta comunicação é a de troca de mensagens e memória compartilhada. Com o aprimoramento da tecnologia surgiu a necessidade de maneiras mais transparentes para o desenvolvedor, surgindo então a chamada remota de procedimento e posteriormente a idéia de objetos compartilhados. Eles apenas abstraem este processo interagindo de maneira mais alto nível com o usuário. Estes tipos de comunicação são listados abaixo [63, 54]:

- *memória compartilhada*: diversas unidades de processamento tem acesso a uma mesma porção de memória. Permitindo, então, a modificação desta memória por uma unidade de processamento a e sua leitura uma outra unidade de processamento b, disponível no ambiente;

- *troca de mensagens*: uma unidade de processamento envia uma porção de dados para uma outra unidade de processamento, a qual deverá ler os dados recebidos e interpretá-los;

- *chamada remota de método*: uma unidade de processamento a solicitará a execução de uma tarefa pela unidade de processamento b, através da chamada de um método do aplicativo executado por b. No entanto, para o desenvolvedor, esta chamada é transparente, se assemelhando a uma chamada local;

- *objetos compartilhados*: representa uma evolução da chamada remota de método. Enquanto na chamada remota de métodos temos a representação de um método remoto como se fosse um método local, em objetos compartilhados existe uma representação local de um objeto remoto, podendo assim atuar diretamente sobre este objeto, mesmo este não sendo um objeto local.

Estes tipos de comunicação são detalhados no restante desta seção.

2.2.1 Memória Compartilhada

Um dos conceitos mais simples de implementar a nível de software com relação à comunicação entre processos é o de memória compartilhada [34]. Esta tecnologia permite que um processo crie um espaço de endereçamento na memória que seja comum a outros processos, permitindo assim que exista a troca de informações entre estes processos.

Esta simplicidade fica a nível apenas do software, principalmente quando está se tratando com ambientes com mais de uma unidade física. Nestes ambientes, o desenvolvimento em hardware do conceito de memória compartilhada é extremamente complexo. Desta forma, apesar de ser mais rápido, e mais simples o desenvolvimento de aplicações baseadas no conceito de memória compartilhada, o seu desenvolvimento em hardware é bastante complexo.

Uma outra vantagem da utilização de memória compartilhada é que os dados colocados na memória só deixarão de existir se forem sobre-escritos. Não ocorrerá do dado ser enviado e não existir mais após a leitura por um segundo processo.

Porém, diversas abordagens já foram realizadas com o intuito de minimizar as dificuldades. Uma destas abstrações em software já desenvolvidas foi o modelo de computação distribuída nomeada de JavaSpace[17]. Este modelo de computação é baseado no espaço de tuplas da linguagem *Linda* [21], desenvolvida por Dr. David Gelernter da Yale University.

Toda a comunicação realizada nesta abordagem é realizada através da escrita e leitura de dados que estão compartilhados em um espaço comum a todos os processos. Um espaço é um repositório compartilhado de objetos acessível pela rede de comunicação. Os processos, então, utilizam este espaço como um mecanismo de persistência de objetos; desta forma, ao invés de realizar uma comunicação direta entre os processos, a comunicação é realizada pela troca de objetos utilizando este espaço compartilhado.

No entanto, ao contrário de um ambiente convencional para armazenamento de objetos, os processos não alteram diretamente os objetos compartilhados. Cada vez que é realizada uma leitura de um objeto, o processo recebe uma cópia deste objeto, utilizando-o localmente. Quando encerrar a sua computação em cima deste objeto, caso seja necessário, basta enviá-lo novamente para o espaço que este irá sobre-escrever o objeto anterior.

As principais vantagens do modelo de implementação de JavaSpace, são [17]: **(i)** *simplicidade*, por possuir poucas operações, este modelo de comunicação não necessita de um grande aprendizado; **(ii)** *expressividade*, mesmo com este conjunto reduzido de instruções, é possível se desenvolver um grande conjunto de aplicações distribuídas; **(iii)** *suporta protocolos não acoplados*, não necessitando saber qual foi o processo que enviou o dado ou quem irá receber a requisição; e **(iv)** *facilidade de uso*, o desenvolvimento de aplicações cliente/servidor é bastante simples, pois características como acesso por múltiplos clientes, concorrência e armazenamento persistente são nativas à API (*Application Programming Interface* - Interface de Programação de Aplicativos) do JavaSpace.

2.2.2 Troca de Mensagens

Além da memória compartilhada, podemos também encontrar a comunicação entre processos sendo realizada através da troca de mensagens [27]. Esta tecnologia é desenvolvida utilizando o conceito de envio e recebimento de um dado. Desta forma, a comunicação é realizada com um processo enviando diretamente a informação para um outro processo que se encontra no aguardo de receber esta informação.

Uma implementação simples desta tecnologia são os *Sockets* [25, 36]. Em Java existe uma implementação desta tecnologia na qual permite criar um canal de comunicação entre dois processos, independente se estes estão ou não situados dentro da mesma máquina. Podendo assim transmitir dados entre as unidades de processamento que se encontram dentro de um mesmo ambiente, contanto que estas estejam conectadas por algum meio de comunicação.

Desta forma, um processo assume o papel de servidor da comunicação, disponibilizando um meio para outro processo se comunicar com ele. Tendo esta comunicação estabelecida, ambos os processos podem se comunicar. No entanto, sempre deverá ser obedecida a limitação de que quando um processo envia uma informação, o segundo processo deverá estar esperando para receber a informação.

Devido a grande complexidade de se desenvolver sistemas computacionais

baseados em troca de mensagens, pois é necessário ter conhecimento de mais baixo nível para poder realizar a comunicação corretamente, ocorreram algumas evoluções e novas tecnologias surgiram, como é o caso da Chamada Remota de Métodos e Objetos Compartilhados.

2.2.2.1 Chamada Remota de Métodos

A comunicação entre processos denominada como Chamada Remota de Métodos(*Remote Procedure Call - RPC*) é vista como uma evolução da troca de mensagens, que permite que um processo realize uma chamada a um procedimento em outro processo [60, 36]. Devido à sua implementação, esta tecnologia é utilizada de forma transparente pelo usuário, tendo o acesso ao objeto remoto, este pode ser lidado como sendo um objeto local.

Em Java temos sua implementação denominada como RMI (Remote Method Invocation - Invocação Remota de Métodos [52]). Sendo uma das abordagens da plataforma Java para a realização de comunicação em ambientes distribuídos, estando presente na API desde a versão 1.1 do JDK (*Java Development Kit* - Kit de Desenvolvimento Java).

O funcionamento da tecnologia RMI é baseada na arquitetura Cliente/Servidor onde o servidor provê os serviços e realiza uma associação a uma porta de comunicação e permanece no aguardo da conexão de algum cliente. No momento que existe a conexão, o processo cliente envia a mensagem de qual método será executado juntamente com os parâmetros caso existam, ficando bloqueado esperando a mensagem de retorno do processo servidor indicando que o processamento necessário encerrou.

2.2.2.2 Objetos Compartilhados

O desenvolvimento baseado em componentes (DBC [53]) procura desenvolver os sistemas computacionais baseados na técnica de re-uso de software [31]. Essa abordagem baseia-se na definição de componentes com interfaces de comunicação bem definidas para poderem ser re-usados posteriormente, possivelmente em ambientes completamente diferentes.

Para o caso de aplicações distribuídas, o DBC sugere uma arquitetura de componentes que permita a comunicação entre os diferentes processos executados. Para isto, se tem a idéia de componentes distribuídos, onde diferentes processos terão condições de acessar um componente remoto como se estivesse acessando um componente local.

Surgiu então o conceito de objetos distribuídos, onde um mesmo objeto poderá ser acessado por diversos processos. Uma das implementações deste conceito é a tecnologia de arquitetura de componentes CORBA (*Common Object Request Broker Architecture* [59, 63]). CORBA não é basicamente um sistema distribuído, mas a especificação de um sistema. Esta especificação foi desenvolvida pelo *Object Management Group* (OMG) [44], uma instituição sem fins lucrativos com mais de 800 membros, a maioria provenientes da indústria. A meta da instituição ao realizar a definição de CORBA foi desenvolver uma arquitetura que possibilitasse a comunicação entre diversas aplicações que estivessem de alguma forma interligadas. Esta arquitetura é responsável por isolar os requisitantes dos provedores de serviço através de uma interface de comunicação bem definida.

Devido a este isolamento provido por CORBA, é possível utilizá-la para comunicação entre processos que estejam em máquinas distintas ou em uma mesma máquina, contanto que exista um meio de comunicação interligando estes processos. Além disto, por ser implementado utilizando uma linguagem de própria e prover uma implementação para diversas linguagens, os objetos de CORBA, podem ser utilizados para a comunicação entre processos destas diferentes linguagens, como Java e C++.

Apesar desta vantagem de poder comunicar aplicativos de diferentes linguagens, esta característica também representa um de seus problemas. A grande dificuldade de CORBA é justamente sua complexidade de desenvolver os objetos na linguagem específica de CORBA e a compilação e uso na linguagem do aplicativo do usuário.

Na tentativa de solucionar os problemas existentes na proposta de objetos distribuídos de Corba, a SUN desenvolveu o padrão *Enterprise JavaBeans (EJB)* [41, 10]. No qual é um padrão mais simples de arquitetura permitindo aos desenvolvedores segurança, transações e persistência de objetos usando um modelo simples de programação e declaração de atributos.

Além de sua portabilidade, os componentes EJB possuem uma interface de comunicação bem definida sendo possível a comunicação entre os componentes através da idéia de serviços, onde um componente pode prover um serviço ou requisitar um outro serviço de um outro componente. Desta forma, a comunicação ocorre através da troca de serviços entre componentes.

Estes componentes EJB ficam então responsáveis por formar toda a aplicação do usuário, necessitando este atender este critério de comunicação entre componentes através de serviços, dificultando a portabilidade de aplicativos já existentes para uma versão distribuída. Enquanto que na proposta do *EasyP*, o objeto antigo do aplicativo pode ser simplesmente encapsulado dentro do padrão de componentes. Além disto, outra diferença é o fato da comunicação entre os componentes EJB ser completamente transparente, impedindo o usuário realizar uma escolha de um determinado protocolo que irá lhe beneficiar de alguma forma devido a organização do ambiente de processamento que será utilizado para o seu caso específico.

Acrescentando a estas diferenças, podemos citar a facilidade e abstração para o usuário. Como por exemplo, ao invés de saber toda a organização e funcionamento dos componentes EJB, o usuário apenas porta os objetos do seu aplicativo para o modelo que estamos propondo e então manda o framework executar, sem se preocupar em quais serviços devem ser implementados ou como se dará a comunicação entre os componentes.

2.3 Conclusões

Neste capítulo vimos os fundamentos do tema estudado, como alguns conceitos do que seriam ambientes paralelos e/ou distribuídos e as possíveis técnicas de comunicação que pode ocorrer entre processos. Cada uma destas teorias abordadas possuem suas particularidades, que quando utilizadas corretamente, será possível tirar toda a sua capacidade de desempenho.

Desta forma, foi necessário este estudo prévio detalhando as particularidades de cada um destes ambientes identificando assim, que a alteração no ambiente incluindo as possíveis modificações nas técnicas de comunicação iriam influenciar diretamente no desempenho do aplicativo. Assim o desenvolvimento do *EasyP* foi direcionado

para poder atuar de acordo com qualquer ambiente ou tecnologia de comunicação que pudesse ser encontrada, porém esta característica prejudicou o desempenho geral da ferramenta *EasyP*. Pois ao ser mais genérica, esta ferramenta teria que lidar com as deficiências de cada um destes ambientes ou tecnologias de comunicação de forma superficial não permitindo que estas sejam tratadas de uma melhor forma.

Assim, surgem duas possibilidades no desenvolvimento de um *framework* de auxílio ao desenvolvimento de aplicativos distribuídos/paralelos: **(i)** um *framework* balanceado para poder executar em qualquer tipo de ambiente/comunicação, perdendo um pouco de desempenho; ou **(ii)** um *framework* que só permitirá executar em um determinado ambiente com um único tipo de comunicação, porém com o melhor desempenho possível. Como escolha para o desenvolvimento do *EasyP* foi tomada a opção de ter uma ferramenta que facilite o desenvolvimento de aplicativos distribuídos sem ter a necessidade do usuário definir nada com relação ao ambiente em que se encontra, sacrificando o desempenho deste *framework*. Porém, foi incrementada sua usabilidade, já que o usuário poderá utilizá-lo em uma gama maior de ambientes, junto à diversos tipo de comunicação, sem a necessidade de configuração prévia.

Capítulo 3

Trabalhos Relacionados

> Não ser ninguém exceto você mesmo, num mundo que se esforça, dia e noite, para torná-lo igual a todo mundo é lutar a pior das batalhas que todo ser humano pode enfrentar; e nunca deixar de lutar.
>
> Edward Estlin Cummings

A constante necessidade de máquinas mais potentes para executar os aplicativos e estes obtem seu resultado em um tempo válido, fez com que o campo de sistemas distribuídos/paralelos tenha sido bastante difundido nos últimos anos. Grande parte das pesquisas neste campo estão voltadas para tirar proveito de apenas uma única tecnologia de comunicação ou de apenas um único tipo de ambiente.

Estes estudos sempre estão focados em extrair o maior desempenho possível das aplicações. Porém, esta característica limita a execução dos aplicativos, não permitindo que possam ser portados para outros ambientes ou utilizarem ambientes mais flexíveis. Além disto, alguns estudos chegam ao ponto de limitar a inserção e a remoção de unidades de processamento em tempo de execução.

Desta forma, estes estudos guiaram inicialmente a realização desta abordagem. Porém, foi realizada uma abstração ainda maior do que a destes estudos ocorridos previamente. Permitindo assim, que se obtivesse como resultado, uma ferramenta mais flexível e ajustável para o uso específico do usuário. No entanto, o preço disto foi a perda de desempenho quando comparado às propostas anteriores.

No restante deste capítulo, são listados alguns estudos no campo de sistemas paralelos/distribuídos os quais os respectivos autores tentam desenvolver uma ferramenta capaz de abstrair as dificuldades de desenvolvimento de aplicativos para estes ambientes. Porém, estes estudos procuram extrair o melhor desempenho do ambiente, mas tornando suas propostas limitadas.

3.1 Aplicações Distribuídas Baseadas Em Trabalho Cooperativo com JavaSpace

O estudo realizado por Mole [40] utiliza um aplicativo para a quebra de senhas que emprega o método da força bruta, as quais foram geradas pelo algoritmo SHA1

(Secure Hash Algorithm). Para realizar a quebra de uma senha por força bruta, o aplicativo deverá testar todas as possíveis senhas até encontrar a senha correta. Porém, analisando este algoritmo, Mole identificou que a validação de uma senha é independente da validação de outra senha, podendo assim, processar a validação de mais de uma senha ao mesmo tempo.

Desta forma, para procurar obter um ganho de desempenho, Mole desenvolveu um protótipo que realiza o processamento distribuído utilizando técnicas de Trabalho Cooperativo com espaço compartilhado. Para realizar o desenvolvimento desta aplicação, Mole utilizou Java e justificou seu uso devido a possibilidade de abstrair a arquitetura das máquinas em que a ferramenta será processada. Devido a esta restrição de projeto, a técnica de espaço compartilhado foi implementada utilizando *JavaSpace* [17].

Neste protótipo, o processo principal gerencia a criação e o processamento de cada uma das sub-tarefas responsáveis por validar cada uma das senhas possíveis. Logo, ao definir uma destas sub-tarefas, esta é colocada no espaço compartilhado de *JavaSpace* e fica no aguardo de que algum processo secundário leia este espaço e capture uma destas sub-tarefas para serem processadas. Ao término do processamento de uma sub-tarefa, o processo secundário retorna para o espaço compartilhado informando se sua validação foi bem sucedida ou não. Caso encontre a senha correta, o processamento acaba e o aplicativo encerra, pois foi completada a computação necessária pelo aplicativo.

Pelo fato destas sub-tarefas serem totalmente desacopladas, não há necessidade do processo principal ter conhecimento de quantos processos secundários existem em atividade. Desta forma, é permitido que novos processos sejam adicionado em tempo de processamento ou que alguns processos sejam removidos também em tempo de execução. Devido a esta característica, Mole ressalta que a escalabilidade do seu protótipo é transparente, não necessitando a modificação de sua aplicação para que altere a quantidade de máquinas existentes no ambiente.

Por possuir a característica de que os processos secundários é que ficam determinados em escolher uma sub-tarefa para processar, esta proposta de Mole apresenta uma característica de balanceamento de carga, já que existindo um processo secundário livre, este fica encarregado de processar qualquer sub-tarefa disponível no espaço compartilhado.

Quando comparado com o estudo desta dissertação, o trabalho de Mole se assemelha pelo fato da escalabilidade transparente, não havendo a necessidade de alteração na ferramenta para que altere o número de máquinas no ambiente. Porém, difere no que diz respeito ao protocolo de comunicação. Mole utiliza apenas uma única abordagem para o protocolo de comunicação utilizado, enquanto a ferramenta proposta ao longo desta dissertação, procura criar uma camada de abstração isolando a ferramenta da comunicação utilizada entre as unidades de processamento. Desta forma, o *EasyP* possui a capacidade de se comunicar utilizando qualquer tipo de protocolo de comunicação. Até mesmo o caso de que não exista uma implementação para tal protocolo, o desenvolvedor poderá desenvolver a conexão deste novo protocolo com esta camada de comunicação do *framework*, passando, assim, a ter suporte para este novo protocolo de comunicação.

3.2 SimRWA-D: Uma Abordagem Distribuída para Simulação de Redes Ópticas Transparentes

Com o intuito de se analisar o comportamento do algoritmo de roteamento RWA em redes ópticas transparentes sem a necessidade de realizar a implementação deste algoritmo fisicamente nos roteadores, Durães criou um aplicativo capaz de realizar esta análise através de simulações, o SimRWA. Posteriormente, Durães desenvolveu outro trabalho de pesquisa devido ao custo computacional envolvido no processamento do SimRWA, este estudo resultou em uma versão do seu trabalho para ser processada em um ambiente distribuído, o SimRWA-D. Este trabalho intitulado "SimRWA-D: Uma Abordagem Distribuída para Simulação de Redes Ópticas Transparentes" [11], permite que seja realizado o processamento da simulação do algoritmo de roteamento do RWA em mais de uma máquina.

Além da redução do tempo de processamento, este estudo ainda resultou na definição de um *framework* genérico para distribuição de processos em um ambiente com mais de uma máquina. Este *framework*, igualmente a proposta de Mole, só trabalha com uma única tecnologia de comunicação, no caso deste trabalho, utiliza apenas RMI para realizar a comunicação entre os processos. Além disto, diferentemente de Mole, a ferramenta proposta por Durães não permite a modificação do ambiente em tempo de execução, ou seja, não permite nem adicionar nem remover máquinas enquanto o aplicativo está sendo utilizado.

Acrescentando a estas características, mais duas falhas foram encontradas, que são a necessidade de existir um outro processo para apenas gerenciar as máquinas existentes no ambiente, conhecido como *Servidor de Nomes*, e a necessidade de preparar previamente o ambiente. Esta preparação prévia se dá pelo fato de que é necessário que todas as máquinas estejam habilitadas a reconhecer a parte do *software* que vai ser processada, para isto todos os processos executados nas máquinas devem ter a definição prévia do que vai processar.

Durães permanece mostrando a mesma limitação de Mole, o manuseamento com apenas uma tecnologia de comunicação. Desta forma, a existência de um ambiente que não permita a utilização de RMI, tecnologia utilizada em seu estudo, para a comunicação, não será permitido a utilização da proposta de Durães. Vemos ainda que Durães apresenta mais uma limitação, a incapacidade de gerenciar a adição ou remoção de máquinas ao ambiente em tempo de execução. Além disto, pode-se acrescentar a necessidade de uma configuração prévia para poder processar o aplicativo de forma distribuída e ainda a necessidade de um processo em separado para o gerenciamento das máquinas existentes no ambiente.

Estas limitações incentivaram e mostraram os possíveis pontos fracos que foram evitados neste trabalho, como permitir a adição e remoção de máquinas em tempo de execução, além deste gerenciamento está integrado ao processo principal do *EasyP*. A proposta desta dissertação também permite uma carga dinâmica, quando necessário, das partes do *software* a serem processados em máquinas remotas.

3.3 Distributed scientific computing in Java: observations and recommendations

O estudo de Sheil foi baseado em um problema de bioinformática utilizando redes neurais [43] e procura analisar o comportamento das proteínas através de simulações. Porém, por ser uma ferramenta com alto custo de processamento, Sheil [50] propôs uma versão distribuída desta simulação.

Assim como Mole, Sheil utiliza a tecnologia de JavaSpace para realizar a distribuição do processamento. Além desta semelhança, a ferramenta proposta por Sheil também permite a inserção e remoção de máquinas durante o processamento do aplicativo, tornando o ambiente dinâmico.

Diferentemente de Durães, a ferramenta proposta por Sheil não necessita de um processo em separado para realizar o monitoramento das máquinas existentes no ambiente. Devido à utilização de um espaço compartilhado de objetos, a ferramenta possui as características muito semelhantes às de Mole, permitindo por exemplo, um balanceamento implícito de carga de processamento.

Por ser muito semelhante à proposta de Mole, a proposta de Sheil apresenta as mesmas características que foram aproveitadas ou rejeitadas no desenvolvimento do conceito envolta da definição do *EasyP*. Por exemplo, a adição e remoção em tempo de execução, a não-necessidade de um processo gerenciando as máquinas existentes e a necessidade de um balanceamento de carga. Apesar do *EasyP* não propor um balanceamento implícito, até porque o balanceamento da proposta de Mole é fraco, foi verificada a necessidade de um balanceamento da carga de processamento, passando a buscar uma forma de permitir que o usuário realize o seu próprio escalonamento do processamento e um melhor e mais eficaz balanceamento de carga.

3.4 Ibis: an efficient Java-based grid programming environment

Nieuwpoort em seu trabalho "*Ibis: an efficient Java-based grid programming environment*" [58], descreve uma ferramenta capaz de realizar uma abstração da camada de comunicação, permitindo que os processos distribuídos possam utilizar qualquer tipo de tecnologia que se deseje. Para isto, Nieuwpoort propõe uma arquitetura em camadas para sua ferramenta, para assim isolar o aplicativo da troca de informações entre os processos.

Porém, este estudo de Nieuwpoort é focado apenas para o uso em GRIDs computacionais. Além de não preparar seu estudo para tecnologias mais atuais como máquinas com vários núcleo de processamento locais, como as máquinas multi-core.

Da proposta de Nieuwpoort, foi retirada a idéia do *EasyP* possuir uma camada de abstração que permitirá a troca de comunicação sem a necessidade de modificar mais nada, nem no aplicativo, nem no ambiente. Desta forma, a ferramenta proposta terá a capacidade de ser executada em qualquer tipo de ambiente computacional. Além disto, foi notada esta dificuldade de lidar com máquinas com mais de uma unidade de processamento local. Assim, é permitido que esta identificação e correto ajuste da ferramenta sejam realizados de forma transparente para o usuário, o qual poderá definir quantas unidades de processamento poderão ser utilizadas e o *framework* que proposto realizará a configuração necessária de forma transparente.

Este trabalho de Nieuwpoort também apresenta a limitação de não poder modificar a quantidade de máquinas do ambiente. Sendo um dos principais pontos de evolução do *EasyP*, pois permite de forma transparente ao usuário que a qualquer momento seja modificado o ambiente em que o aplicativo esteja sendo processado.

3.5 Conservative Simulation using Distributed-Shared Memory

O trabalho de Teo [56] é baseado na evolução de seu trabalho prévio na biblioteca SPaDES/Java (*Structured Parallel Discrete-event Simulation in Java*) [57]. Esta biblioteca foi desenvolvida para realizar a simulação de eventos do mundo real. Para isto, existe um mapeamento das entidades e atividades do mundo real para processos no modelo conceitual e estes processos são representados como sendo processos lógicos a serem simulados.

A versão inicial do SPaDES já é uma versão distribuída na qual utiliza RMI para realizar a comunicação entre os diversos processos que existem dentro do ambiente de processamento. Já na versão desenvolvida no estudo "*Conservative Simulation using Distributed-Shared Memory*", Teo propôs uma modificação utilizando JavaSpace para realizar a comunicação ao invés de utilizar RMI. Desta forma, Teo obteve duas versões separadas do seu sistema onde cada uma utiliza unicamente um tipo de comunicação diferente.

Com esta modificação, Teo demonstra que obteve um pequeno ganho de performance. Isto tendo ocorrido a única alteração de tecnologia de comunicação. Deste estudo de Teo surgiu a idéia de permitir que diversas tecnologias de comunicação sejam possíveis de serem utilizadas, pois dependendo do ambiente poderá ocorrer um pequeno ganho de performance ao utilizar uma comunicação mais adequada. Desta forma, ao invés de ter-se várias aplicações que usem diferentes tipos de comunicação, é proposto uma ferramenta que possa gerenciar esta característica de forma abstrata para o usuário. Assim, para alterar o tipo de comunicação a ser utilizado, bastará para o usuário modificar um parâmetro de inicialização da ferramenta.

Além desta característica, o estudo de Teo demonstra algumas limitações como os outros estudos listados previamente. Não permite a modificação do ambiente em tempo de processamento, além de não permitir um correto balanceamento de carga da aplicação sem alterar a ferramenta, e não ser capaz de executar corretamente em qualquer tipo de ambiente. Ao contrário do *EasyP*, no qual uma mesma versão do sistema permite que seja utilizada a tecnologia de comunicação A ou a tecnologia B.

3.6 A Model-Driven Approach to Job/Task Composition in Cluster Computing

Mais um estudo analisado no decorrer do desenvolvimento desta proposta foi o estudo realizado por Mehta [38]. Em seu trabalho"*A Model-Driven Approach to Job/Task Composition in Cluster Computing*", Mehta propõe um framework capaz de realizar a distribuição de processos para máquinas que possuem recursos

computacionais livres dentro de um ambiente organizado em *cluster*. Porém, não existe a idéia de balanceamento de carga implícito, nem a possibilidade do usuário realizar este balanceamento a nível de código.

Neste trabalho, Mehta propõe o framework *Computational Neighborhood* (CN), no qual utiliza a idéia de *tasks* e *jobs*. No caso, *jobs* seriam um conjunto de *tasks* para serem processadas e os *tasks* são consideradas como threads para serem processadas no CN *Framework*. A definição destes *tasks* e *jobs* dependem do usuário, sendo auxiliados pela API do CN, na qual fica responsável pela criação, controle e coordenação dos *tasks*.

Esta API do CN pode ser dividida em quatro partes: **(i)** *Tasks*, como sendo o processamento a ser realizada; **(ii)** *Jobs*, que é um conjunto de *tasks* que deve ser processada por uma unidade de processamento disponível no ambiente; **(iii)** *JobManager*, responsável pelo gerenciamento dos *jobs*, sendo a interface de comunicação entre os *jobs* e o usuário do framework; e **(iv)** *TaskManager*, que gerencia a execução das *tasks*, ficando transparente à nível de usuário.

O trabalho de Mehta se torna bastante limitado pelo fato de estar focando unicamente no ambiente de *clusters*, além de permitir apenas um único protocolo de comunicação. Desta forma, não deverá ser processado da melhor forma em ambientes que não permitam estas configurações, podendo perder desempenho da execução. Porém, existe esta idéia de definir um conjunto de tarefas a serem processadas para facilitar o envio e reduzir o custo de transporte destes dados pela rede. Diferentemente da proposta de Mehta, o *EasyP* permite a utilização em qualquer tipo de ambiente sem necessitar uma configuração prévia.

3.7 PVM - Parallel Virtual Machine

Uma ferramenta que pode ser comparada à abordagem realizada nesta dissertação é a ferramenta conhecida como PVM (*Parallel Virtual Machine* [20]). Esta ferramenta pode ser vista como uma máquina virtual para realizar processamento paralelo. Além disto, a PVM pode ser vista, também, como um conjunto de outras ferramentas e bibliotecas que possuem a capacidade de emular um *framework* de propósito geral e flexível capaz de realizar computação distribuída em um ambiente com máquinas que possuem arquiteturas distintas. Desta forma, esta ferramenta possibilita que um conjunto de máquinas atuem de forma cooperativa passando a idéia de um único recurso computacional.

Uma das características que diferenciam a PVM de algumas propostas já vistas, é a capacidade de inserir ou remover dinamicamente novas máquinas durante o processamento do aplicativo. Esta capacidade faz com que o usuário possa utilizar por curtos períodos de tempo, máquinas com maior poder computacional, mas que são extremamente disputadas para executarem outros aplicativos.

Outra característica interessante é que a PVM possui *tarefas* como sendo sua unidade de paralelismo. No entanto, não impõe o mapeamento processo-processador, permitindo que diversas tarefas sejam executadas no mesmo processador. Acrescentando a isto, podemos listar como característica, que toda a comunicação realizada no ambiente da PVM é realizada através da troca de mensagens. Esta característica permite uma heterogeneidade entre as máquinas e também pela camada de comunicação. Pois enviando uma mensagem dentro do padrão, está poderá ser lida em qualquer máquina, contanto que reconheça o padrão

da mensagem.

Em sua versão inicial, a PVM limita sua utilização em apenas ambientes Unix e a utilização unicamente de aplicativos desenvolvidos em C, C++ e Fortran. Esta limitação mostra que seu principal alvo são aplicativos científicos que requerem alto desempenho, pois se enquadram justamente nestas limitações. No entanto, já existem grupos de pesquisa realizando *bindings* da PVM para outras linguagens.

Ao comparar a PVM com o *EasyP*, vemos muitas similaridades, como a idéia de que de forma genérica toda a comunicação é tratada como uma simples troca de mensagens. Além disto, a capacidade de gerenciar em tempo de execução do aplicativo a modificação das máquinas no ambiente. Porém, a diferença é o publico alvo, a PVM está voltada para ambiente Unix e aplicativos escritos em C, C++ e Fortran, enquanto o *EasyP* foi desenvolvida para qualquer ambiente computacional e inicialmente para aplicativos em Java.

3.8 An automatic distributed simulation environment

Bruschi [8, 9] desenvolve um estudo na área de aplicativos para ambientes distribuídos. Este estudo procura realizar o desenvolvimento de um ambiente com a capacidade de gerar automaticamente simuladores que poderão ser processados em ambientes com mais de uma unidade de processamento. Uma das características deste ambiente é a possibilidade do usuário, iniciante na área de simuladores, criar seu simulador auxiliado por ferramentas visuais.

Desta forma, este ambiente proposto por Bruschi possui como público alvo desenvolvedores, estudantes e pesquisadores de variados níveis de conhecimento, já que provê a capacidade de montar em alto nível todo o processo de simulação. Isto se é devido a possibilidade de tanto realizar o desenvolvimento destes simuladores através de sua interface gráfica, como também permite que desenvolvedores mais experientes realizem a manipulação do código em baixo nível.

Por esta facilidade de utilização, esta ferramenta pode ser utilizada para o ensino do desenvolvimento de simuladores para pesquisadores iniciais. Como também, esta ferramenta pode ser utilizada por usuários mais avançados que desejem desenvolver uma versão de sua simulação para ser executada em um ambiente distribuído, mas não modificar sua versão sequencial manipulando o código responsável pela distribuição em baixo nível.

Porém a abstração realizada por Bruschi, torna a ferramenta limitada para apenas o desenvolvimento de simulações. Não sendo permitido que utilize este proposta para o desenvolvimento de aplicativos mais gerais. O que não ocorre na proposta do *EasyP*, na qual é possível que o usuário use a ferramenta proposta para a adaptação ou criação de qualquer aplicativo para ser executado em qualquer ambiente que possua mais de uma única unidade de processamento.

3.9 Conclusões

Ao fazermos uma análise geral entre todos estes trabalhos relacionados e a proposta do *EasyP*, encontramos as diferenças técnicas na Tabela 3.1. Porém uma grande diferença não é meramente técnica, se trata da *usabilidade*. A proposta do

EasyP, procura abstrair ao máximo algumas das dificuldades encontradas no desenvolvimento de aplicações distribuídas, além da sua facilidade de uso, tornando-se uma ferramenta prática para se conseguir uma primeira versão distribuída de uma aplicação sequencial sem desprender muito esforço para tal. No entanto, é previsto que esta versão obtida ao utilizar o *framework*, proposto nesta dissertação, não consiga um desempenho equiparável a uma implementação realizada especificamente para o problema em questão , já que a proposta do *EasyP* tornará o aplicativo do usuário mais genérico, podendo ser utilizado em diversos ambientes com diversas configurações, com pouca ou nenhuma modificação no código do aplicativo.

Neste capitulo foram apresentados alguns estudos realizados previamente e comparados com a proposta do *EasyP*. Nesta comparação, foi ressaltada as diferenças entre cada uma destas propostas. Durante a comparação, foram identificados os pontos fortes e fracos de cada uma das abordagens, definindo o escopo do *EasyP*. Para encerrar o capitulo, a Tabela 3.1 mostra um comparativo direto de algumas características encontradas nos trabalhos estudados.

Tabela 3.1: Comparativo entre as características de cada estudo realizado previamente e a nossa proposta.

Critério	Mole	Durães	Sheil	Nieuwpoort	Teo	Mehta	PVM	Bruschi	*EasyP*
Tecnologia	JavaSpace	RMI	JavaSpace	Várias	RMI e uma segunda versão utilizando JavaSpace	Troca de mensagens	Troca de mensagens	Troca de mensagens	Várias
Ambiente	Dinâmico	Estático	Dinâmico	Estático	Estático	Estático	Dinâmico	Estático	Dinâmico
Multi-núcleo	O usuário tem que identificar	O usuário tem que identificar	O usuário tem que identificar	O usuário tem que identificar	O usuário tem que identificar	O usuário tem que identificar	Provê suporte para ambientes multi-processados	O usuário tem que identificar	Provê suporte automático e invisível ao usuário
Linguagem	Java	Java	Java	Java	Java	Java	C, C++ e Fortran	Visual e linguagem própria	Java
Propósito	Genérico	Genérico	Simulações	Genérico	Genérico	Genérico	Genérico	Simulações	Genérico

Capítulo 4

EasyP - Easy Performance

> Nunca diga às pessoas como proceder. Diga-lhes o que deve ser feito e elas surpreenderão você com sua engenhosidade.
>
> George S. Patton

Cada vez mais nos deparamos com aplicativos que necessitam de um alto poder computacional para poder executar corretamente. As vezes o conceito de executar corretamente entendido por poder realizar completamente a computação necessária [4]. Porém, pode ser devido ao fato de que, para obter um resultado considerado válido, este tem que ser alcançado dentro de um determinado tempo [18].

Devido a limitações na tecnologia de fabricação dos processadores, o desenvolvimento de máquinas que pudessem realizar processamento paralelo de instruções foi realizado. Para isto, foi realizado a multiplicação dos recursos de processamento. Ao proceder desta forma, foi possível reduzir a frequência interna do processamento, permitindo o processador atuar em uma faixa aceitável de geração de calor. Mas, a capacidade de processamento se tornou maior, apenas para aplicativos que estão preparados para utilizar corretamente todo este potencial de paralelismo.

Além do fato de termos computadores com recursos computacionais duplicados, podendo executar instruções em paralelo, o preço das máquinas computacionais decaiu muito ao passar dos anos [26, 61]. Desta forma, várias instituições de pesquisa possuem laboratórios com várias máquinas, as quais poderiam ser utilizadas como um recurso computacional distribuído. Desta forma, este conjunto de máquinas pode passar ao aplicativo, a idéia de um único recurso computacional [35].

No entanto, o desenvolvimento de aplicações capazes de utilizar todo o poder de paralelismo existente nos novos processadores ou o reconhecimento de várias máquinas como sendo um ambiente computacional unificado, não é uma tarefa trivial. Diversos problemas surgem ao tentar criar programas paralelos [17].

Devido a estes problemas, diversos pesquisadores, que necessitam de simuladores que demoram horas ou até mesmo dias para executarem [18], mas que o resultado não deixa de ser válido caso demore todo este tempo, preferem realizar o desenvolvimento de um aplicativo mais simples e sequencial [8, 9]. Porém, a existência de uma ferramenta capaz de facilitar o desenvolvimento de aplicativos distribuídos, faria com que estes pesquisadores, chegassem ao resultado de suas pesquisas mais rapidamente, podendo beneficiar vários outros campos de pesquisa [18].

4.1 Proposta

Diversos estudos já foram realizados procurando propor uma ferramenta que facilite o desenvolvimento de aplicativos para serem executados em ambientes com mais de uma unidade de processamento. Alguns destes trabalhos foram contemplados no Capítulo 3. Porém, cada um dos trabalhos abordados nesta dissertação, possui sua limitação. Alguns tem sua usabilidade muito complexa; outros estudos são limitados quando analisados os ambientes em que o aplicativo poderá executar, não permitindo o uso em diversas configurações de ambientes; enquanto outros possuem a limitação de permanecer com o ambiente de forma estática [1] durante todo o período de processamento do aplicativo, impedindo ao desenvolvedor utilizar máquinas mais potentes durante curtos períodos de tempo ao longo do período de processamento.

O estudo aqui apresentado, é mais um na tentativa de facilitar o desenvolvimento de aplicações distribuídas, porém abstraindo as dificuldades encontradas nas outras ferramentas para o mesmo fim. Com isto, foi utilizada uma técnica de reuso de software para realizar a implementação das funcionalidades em um domínio de *frameworks* [32, 3]. A proposta aqui apresentada é a do desenvolvimento de um *framework* horizontal [13] denominado *EasyP* - *Easy Performance* [46], no qual proverá suas funcionalidades a diversos tipos de problemas computacionais, desta forma sendo capaz de abstrair complexidades como: comunicação, independência da máquina, independência do ambiente, capacidade de adicionar ou remover novas máquinas ao ambiente, complexidade de despacho das tarefas a serem executadas e complexidade de uso do próprio *framework* proposto. Assim sendo, irá facilitar o desenvolvimento de aplicativos distribuídos, permitindo que desenvolvedores criem seus aplicativos com capacidade de utilizar todo o potencial provido por um ambiente com mais de uma unidade de processamento. Desta forma, esta ferramenta irá proporcionar que aplicativos em que o tempo de resposta do seu processamento não seja relevante, possam mesmo assim, de uma maneira mais fácil, alcançar o resultado em um tempo menor, permitindo que a análise deste resultado seja realizada mais rapidamente, disponibilizando a possibilidade que ocorram novos avanços científicos ainda mais rapidamente.

Para realizar esta distribuição de processamento dentro do ambiente mais facilmente, o *EasyP* pode disponibilizar três tipos de serviços: **(i)** *Master* que possui a característica de receber os componentes e fazer o gerenciamento para qual outro ponto de processamento será enviado o componente para poder ser processado;**(ii)** *Slave* ficando responsável por realizar o processamento necessário pelo componente; e **(iii)** *Client* que é por onde o aplicativo do usuário irá interagir com o *framework*, enviando por este os componentes para o *Master* e sendo então distribuídos para os *Slaves* processarem. É necessário observar que um ponto de processamento configurado para prover o serviço de *Master* também poderá disponibilizar o papel de *Slave*, permitindo assim que além de distribuir os componentes para serem executados, também execute localmente os componentes.

Analisando a proposta do *EasyP* a nível de ambiente, existe uma característica que nem sempre é encontrada em outros estudos semelhantes, a capacidade do ambiente ser dinâmico. Ou seja, o *EasyP* que está sendo proposto, permite que máquinas sejam adicionadas ou retiradas do ambiente durante a execução do

[1] Ambiente estático ou dinâmico, no contexto desta dissertação, é a capacidade de remover ou adicionar novas máquinas ao ambiente durante a execução do aplicativo.

aplicativo. Esta característica permite que no caso de uma instituição ter diversas máquinas simples que estão disponíveis durante a maior parte do tempo e um conjunto de máquinas com grande poder computacional, mas que sempre estão ocupadas e que para utilizá-las deverá ser realizada uma reserva por um curto período de tempo, comece a execução do seu aplicativo no conjunto de máquinas mais simples e quando puder, mover a execução do seu aplicativo para as máquinas mais potentes e no momento que sua reserva de tempo nestas máquinas acabarem, será possível retornar para apenas o conjunto de máquinas mais simples. Desta forma, será melhor aproveitada as máquinas existentes dentro de uma instituição.

O desenvolvimento de uma ferramenta capaz de realizar estes tipos de abstrações, terá ainda que lidar com outro problema, sua complexidade e a posterior manutenção do *software*. De acordo com Rollings [47], um aplicativo desenvolvido em uma linguagem procedural, pode chegar a 25000 linhas de código antes de apresentar problemas com a sua manutenção. Enquanto um aplicativo desenvolvido em uma linguagem Orientada a Objetos pode chegar a 100000 linhas de código antes de apresentar algum problema parecido de manutenção. Justificando então o uso de uma linguagem Orientada a Objetos como Java ou C++. Porém, analisando estudos similares ao realizado nesta dissertação, foi observado que vários destes estudos utilizaram a tecnologia Java como padrão para desenvolver seus aplicativos distribuídos, acrescentando a isto o fato de que um dos requisitos da ferramenta é justamente permitir que esta seja utilizada em qualquer tipo de máquina, justificando então o desenvolvimento desta ferramenta em Java [39], por ser uma linguagem compilada e posteriormente interpretada em software, sendo independente da plataforma, mesmo esta ainda sendo considerada lenta quando comparada com outras linguagens que são somente compiladas. Esta independência da plataforma, permitirá ao *framework* proposto, executar em qualquer tipo de máquina, independente da sua arquitetura.

A utilização de uma linguagem Orientada a Objetos permitirá a utilização de seus padrões facilitando a quebra do aplicativo em porções de códigos bem definidas e, em certos casos, completamente isoladas, utilizando a idéia de componentes de software [53, 3]. Desta forma todo o processamento realizado pelo *framework* é baseado na teoria de componentes de software, onde o *framework* irá receber o componente a ser processado e irá enviá-lo para uma unidade de processamento livre dentro do ambiente de execução. Como é realizado o envio de qualquer tipo de componente para manipular qualquer tipo de dado, podemos classificar o *framework*, usando a taxonomia de Flynn [14], como sendo um sistema MIMD.

O resultado do processamento de cada componente é armazenado em um espaço compartilhado de dados, no qual todos os componentes e o próprio aplicativo do usuário terão acesso. Este espaço existe a nível do *framework* e é baseado em um armazenamento do tipo chave-valor. Ou seja, cada valor existente neste espaço só pode ser acessado através da chave única que está a ele associada.

A proposta deste *framework* propõe a abstração da utilização da mesma em máquinas com mais de uma unidade de processamento, permitindo que a identificação de quantas unidades de processamento estejam disponíveis seja realizada automaticamente. Com a correta identificação, será inicializado quantas instâncias do *framework* forem necessárias, permitindo a criação de um ambiente de execução com mais de uma unidade de processamento disponível para a execução dos componentes do usuário.

Para poder realiza o envio do componente para uma unidade de execução remota, a ferramenta proposta deverá realizar uma comunicação direta com outros processos dentro do ambiente. Esta comunicação se dá a nível da camada de comunicação do *framework* via troca de mensagens. Porém, entre o *framework* e a tarefa de comunicação, existe uma camada de abstração, permitindo que a comunicação seja implementada como sendo qualquer tipo de serviço, desde memória compartilhada a uma simples troca de mensagens, a nível do *framework* será sempre vista como uma troca de mensagem. Além disto, esta camada de abstração permite a fácil troca do processo de comunicação ou até mesmo, a ocorrência simultânea da comunicação de um determinado tipo com uma unidade de processamento e utilizar um outro tipo de comunicação completamente diferente para se comunicar com uma segunda unidade de processamento. Acrescentando a estes benefícios, podemos citar a capacidade do usuário poder definir sua própria comunicação, desenvolvida para seu ambiente específico, na qual permitirá utilizar todos os benefícios existentes na configuração de seu ambiente.

Além de isolar o aplicativo do usuário da comunicação entre as unidades de processamento existentes no ambiente e isolar também da arquitetura das máquinas, o *framework* permite que o usuário realize um estudo sobre o seu aplicativo e encontre um balanceamento de carga ideal e implemente o algoritmo de despacho de componentes específico para o seu problema. Desta forma, permite que o usuário identifique um melhor balanceamento da execução de seus componentes e defina especificamente, em que ponto de execução cada um destes componentes serão executados. Com isto, o usuário poderá obter um ganho de processamento baseado na política de despacho específica para a organização do ambiente de processamento em questão.

Assim sendo, a idéia do framework aqui proposto é realizar ao máximo a abstração dos problemas de baixo nível que ocorrem ao desenvolver aplicativos distribuídos, permitindo assim uma interface fácil e prática com o usuário. Porém, ainda permitindo à usuários mais experientes realizarem uma configuração do *framework* mais específica para o seu aplicativo em questão. Desta forma, para o usuário fazer uso do *EasyP*, terá basicamente que realizar a componentização do seu aplicativo, seguindo o padrão definido pelo *EasyP*, e então solicitar ao framework que processe os componentes do aplicativo. Porém, a abstração ao ponto que está sendo proposto tem seu preço, o desempenho. Por estar se tentando abstrair diversos problemas, é previsto que a abordagem do *EasyP* obtenha um desempenho pior quando comparado à aplicações que tem sua distribuição de processo realizada especificamente para o problema em questão. No entanto, não terá o custo associado ao se desenvolver a própria interface de distribuição de processamento em baixo nível. Desta forma, foi focado o desenvolvimento da ferramenta na praticidade de uso, para que os desenvolvedores que realizam estudos em aplicativos que não dependem do tempo de resposta, possam ainda assim, obter uma resposta mais rapidamente e poder assim, ter uma análise dos dados mais rápida.

Durante o restante da seção será analisado cada uma destas características da proposta do *EasyP*. Como o modelo de componentes, a camada de abstração, a política de despacho, a dinâmica do ambiente e a questão da usabilidade, na qual foi mantida da forma mais simples possível.

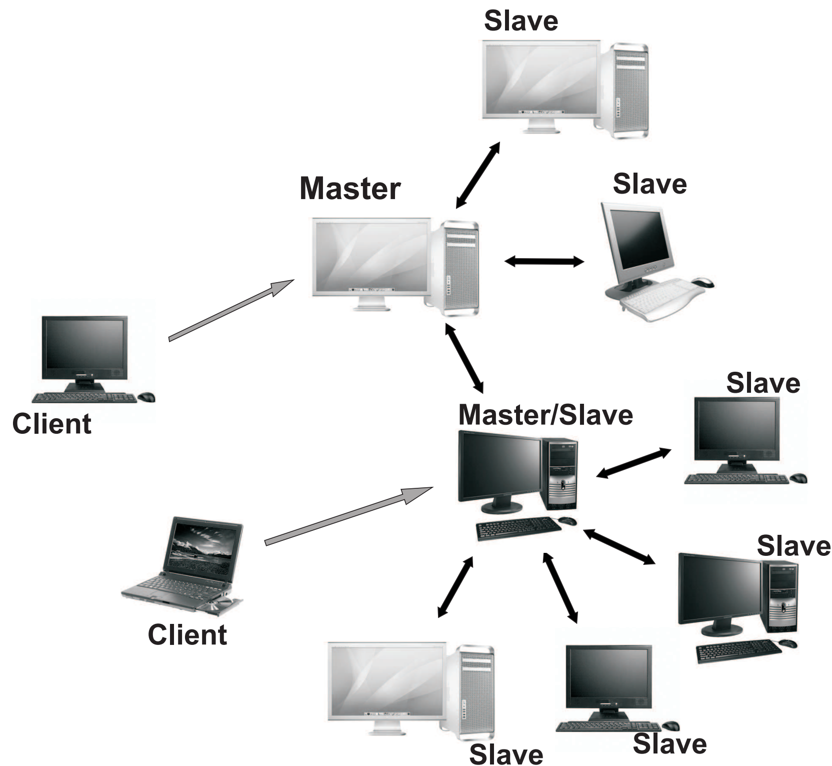

Figura 4.1: Serviços que o *framework* pode prover dentro do ambiente.

4.1.1 Serviços

Dentro do ambiente de processamento do aplicativo, cada unidade de processamento possuirá uma instância do *framework* proposto executando. Em cada uma destas instâncias, o *framework* proverá um serviço, no qual pode ser *Master*, *Slave* ou *Client*. Porém, não são serviços exclusivos, um ponto de processamento pode estar sendo um *Master* como também um *Slave*. Como demonstra a Figura 4.1.

Como pôde ser visto, cada máquina do ambiente assumirá um papel específico provendo um ou mais serviços, onde:

- *Master*: O ponto de processamento fica responsável por receber os componentes e realizar a distribuição para o ponto do ambiente que realizará o processamento do componente;

- *Slave*: O ponto de processamento remoto que está responsável por receber os componentes enviados pelo *Master* e então processá-los;

- *Client*: Serviço do *framework* que se comunica diretamente com o aplicativo do usuário, recebendo as requisições de processamento deste aplicativo e encaminhando os componentes para o ponto do ambiente com o papel de *Master* realizar a correta distribuição destes componentes.

A hierarquia vista na Figura 4.1 é para tentar possibilitar um melhor gerenciamento de um ambiente comunitário. Por exemplo, uma instituição que possui diversos departamentos internos e que cada departamento possui seu próprio laboratório, além da existência de um laboratório exclusivo para processamento de aplicativos da gerência da instituição. Desta forma, uma configuração possível é

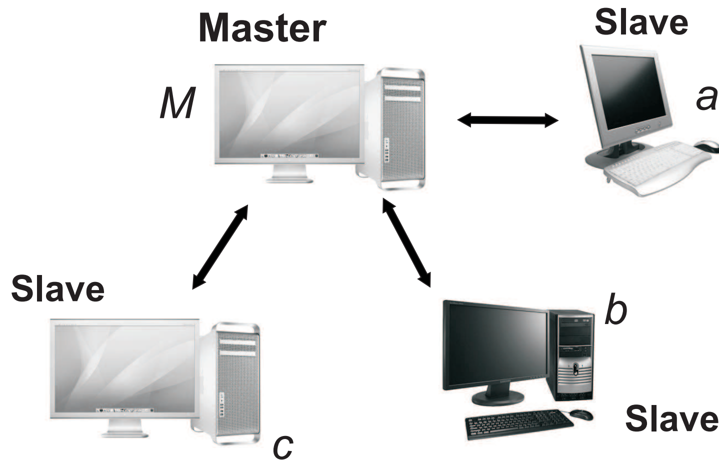

Figura 4.2: Exemplo de ambiente onde temos uma máquina com o papel de Mestre (M) e três máquinas exercendo o papel de Escravas (máquinas a, b e c)

a utilização de vários ambientes isolados para cada departamento e um ambiente englobando todos estes outros sub-ambientes, permitindo à gerência o acesso a todas as máquinas dos laboratórios do departamento, mas não permitindo o compartilhamento entre recursos dos departamentos, nem da utilização do laboratório da gerência por um dos departamentos.

Além destes papéis, existe ainda o serviço de realizar o processamento local dos componentes. Pois cada um destes serviços listados anteriormente representa apenas o papel do ponto de processamento no ambiente. O serviço que é responsável por realizar o processamento do componente é o chamado serviço local. Para realizar a inicialização destes serviços, basta fazer a chamada dos métodos correspondentes no *framework*, listados abaixo:

```
1 public abstract class Framework {
2   public boolean startLocalService(...) {...}
3   public boolean startRemoteMasterService(...) {...}
4   public boolean startRemoteSlaveService(...) {...}
5   public boolean startClientService(...) {...}
6   ...
7 }
```

Estes quatro métodos possuem as seguintes finalidades: **(i)** *startLocalService* - inicializa o serviço local, que fica responsável por criar os processos locais e processar os componentes que forem solicitados; **(ii)** *startRemoteMasterService* - inicializa o serviço *Master* que é responsável por ficar no aguardo do recebimento de componentes para serem distribuídos, além de gerenciar as máquinas com papéis *Slaves*, adicionando ou removendo estas máquinas; **(iii)** *startRemoteSlaveService* - inicializa o serviço *Slave* no qual fica sendo responsável por receber os componentes provenientes do *Master* e enviá-los para o processamento local; e **(iv)** *startClientService* - responsável por inicializar o serviço *Client* que atua diretamente junto ao aplicativo do usuário, recebendo deste os componentes a serem processados e os encaminhando para o *Master* para serem distribuídos.

No entanto, quando montamos um ambiente como ilustrado na Figura 4.2, com um mestre(M) e três escravos (a, b e c), com cada uma destas máquinas de núcleo único, o que realmente acontece é que a instância do *framework* existente na máquina M possui objetos abstratos que representam as máquinas remotas a, b e c. Desta

forma, o *framework* tem a capacidade de tratar localmente as máquinas remotas e apenas no momento de enviar ou receber alguma mensagem para outras máquinas, é que estes objetos abstratos entram em contato com a camada de abstração da comunicação e então se é realizada a transferência de dados com as máquinas físicas e reais do ambiente. Desta forma, as máquinas físicas ficam completamente independentes quando olhamos a nível de software.

4.1.2 Ambiente

Olhando a abstração a nível de ambiente e não mais de máquinas, nos deparamos com uma característica muito comum em outras abordagens, a falta de capacidade de poder gerenciar a adição e a remoção de novas unidades de processamento ao ambiente. Ao longo desta dissertação, um ambiente com esta característica é denominado como sendo um ambiente estático.

Esta falta de elasticidade não permite que em caso de uma instituição possua algumas poucas máquinas de alto desempenho, mas que são extremamente utilizadas, necessitando até a reserva de tempo destas, e muitas máquinas mais simples, porém quase nunca usadas, o seu pesquisador não possa montar um ambiente para execução distribuído com as máquinas mais potentes, pois poderá necessitar de muito tempo de processamento e não terá como reservar este tempo todo. Enquanto que, caso fosse possível montar um ambiente dinâmico para o processamento, o aplicativo poderia ser iniciado nas máquinas mais simples e sempre que as máquinas mais potentes estivessem disponíveis, transferisse o processamento do aplicativo para elas e caso fosse necessário retornar o processamento apenas para as máquinas mais simples.

O *framework* aqui proposto possui esta capacidade de gerenciar dinamicamente a adição e a remoção das máquinas ao ambiente em tempo de execução. Permitindo que em um dado instante, exista n máquinas e em um segundo instante de tempo, o ambiente já esteja formado com m máquinas, onde n não necessariamente deve ser igual a m. Isto pode ser analisado na Figura 4.3, onde estão representados dois momentos do processamento de um aplicativo do usuário.

Desta forma, o usuário do *framework* que estamos propondo poderá ter uma maior flexibilidade para o processamento de seu aplicativo, não ficando preso ao ambiente. Além de que em caso de uma das conexões se encerrarem por algum motivo (usuário desconectou o ponto de conexão com o ambiente ou faltou energia, por exemplo), o *framework* está capacitado para recuperar o componente que havia sido enviado e reenviá-lo para um outro ponto do ambiente para ser processado novamente. Isto é possível pelo fato do *framework* armazenar em memória todos os componentes enviados e seus respectivos destinos e ocorrendo uma verificação de validade dos pontos do ambiente de tempos em tempos, de acordo com a idéia de *timeout* [37]. No momento que é então identificado que um ponto foi desconectado, o *framework* recupera da memória o componente perdido e o reenvia para um outro ponto do ambiente ainda vivo.

4.1.3 Ambiente multiprocessados

Com esta facilidade, de hoje em dia encontrarmos facilmente máquinas multiprocessadas, podemos utilizar aplicações distribuídas de maneira muito mais

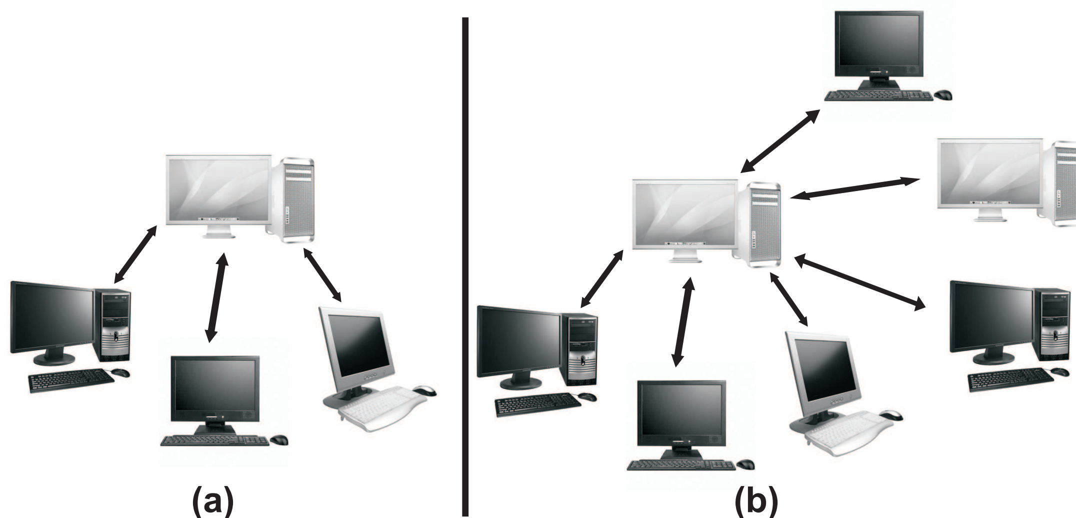

Figura 4.3: Imagem representando dois momentos distintos no processamento de um aplicativo. No momento *(a)* temos três máquinas conectadas ao *Master*. No momento *(b)*, podemos visualizar a inserção de três novas máquinas ao ambiente, totalizando seis máquinas conectadas ao *Master*

simples, não havendo mais a necessidade de termos diversas máquinas físicas completas interligadas por uma rede de comunicação. Porém, é necessário identificar quantas unidades de processamento estão disponíveis no computador e inicializar quantas instâncias do processo sejam necessárias. Esta identificação deverá ser realizada corretamente, pois caso sejam iniciadas uma quantidade menor de processos do que a quantidade de unidades de processamento disponíveis fará com que o sistema computacional seja subutilizado. Enquanto que o contrário irá prejudicar o desempenho, pois teremos mais de um processo concorrendo por um mesmo recurso computacional.

Os diversos estudos realizados anteriormente deixam a cargo do usuário das ferramentas identificarem e inicializarem corretamente as diversas instâncias da ferramenta, ficando mais propício a um má dimensionamento do ambiente. Procurando evitar este problema, a proposta do *EasyP* procura automaticamente identificar e inicializar as instâncias do processo. Porém, está permitido que o sistema seja dimensionado para deixá-lo subutilizado, mas nunca de forma que exista mais processos do que a quantidade de unidades de processamento disponíveis.

4.1.4 Modelo componentes

Muitos sistemas computacionais apresentam partes similares ou até mesmo idênticas, realizando a mesma funcionalidade; nos quais, muitas vezes, são repetidas vezes desenvolvidos do zero [49]. Devido a isto, existem várias propostas para o reuso de partes de sistemas desenvolvidos a priori, técnica denominada *Reuso de Software* [31].

Abstratamente podemos ter o reuso de diversas partes integrantes ao desenvolvimento de software, como reuso de documentação [48] ou reuso de partes funcionais do sistema desenvolvido [7]. Segundo Szyperski [53], a definição de componentes de software é: *"Uma unidade de composição com interfaces*

especificadas de forma contratual e com dependências apenas de contexto e explícitas. Um componente de software pode ser distribuído independentemente e fica sujeito a composição por terceiros".

Quando olhamos para o desenvolvimento baseado em componentes já existentes, reparamos na redução do tempo de entrega do produto ao mercado e no aumento da qualidade deste produto [53]. Este desenvolvimento baseado em componentes só se tornou possível devido a maturidade das tecnologias, que permitiram a construção de componentes e a combinação destas tecnologias para o desenvolvimento de aplicações; bem como o atual contexto organizacional e de negocio, apresentando mudanças em como as aplicações são desenvolvidas, utilizadas e mantidas [7]

Desta forma, o desenvolvimento do framework proposto realiza a distribuição de processamento baseado na idéia de componentes, onde cada componente representará uma porção do processamento a ser realizado. Desta forma, o *framework* realiza a distribuição destes componentes no ambiente com mais de uma unidade de processamento. Para isto, cada um dos componentes de software a serem processados pelo *framework* devem estar bem definidos e atender as especificações impostas pelo *framework*, mas não necessariamente auto contidos. Ou seja, cada um dos componentes devem implementar uma interface Java, na qual pode ser visualizada abaixo, mas poderão necessitar de dados processados por outros componentes. Deixando a cargo do desenvolvedor do aplicativo o correto escalonamento dos componentes para não ocorrer um *livelock* ou um *deadlock* entre os componentes.

```
1 public interface IDistributedComponent {
2   public long getId ();
3   public Object execute ();
4 }
```

Como pôde ser visto, o desenvolvedor que queira utilizar o *EasyP* para a distribuição de componentes deve criar estes implementando dois métodos Java: **(i)** *getId*, no qual deve retornar a chave de identificação de cada componente. Esta chave será utilizada para a recuperação do resultado do processamento do componente diante do *framework*; e **(ii)** *execute*, neste método deve estar localizado todo o processamento realizado pelo componente.

Caso o usuário deseje ganhar um pouco mais de desempenho, deverá implementar uma outra interface que irá prover a capacidade ao usuário de implementar sua própria serialização de seus objetos. Esta interface pode ser vista abaixo e que devemos ressaltar que o usuário implementando esta interface, fica responsável em implementar mais dois métodos, que são responsáveis por converter o objeto em uma *stream* de dados: **(i)** *exportData*, responsável por exportar os dados do componente como uma *stream* de dados, formando um array de bytes; e **(ii)** *importData*, responsável por fazer o papel inverso, receber o *stream* de dados e montar o componente.

```
1 public interface IPerformanceComponent extends IDistributedComponent {
2   public byte[] exportData ();
3   public void importData (byte[] stream);
4 }
```

A implementação dos métodos de importar e exportar dados, permitirá que exista uma conversão de dados mais eficiente para serem transmitidos via canal de comunicação. Porém, a implementação destes dois métodos pode ser bastante

complexa. Desta forma, existe a possibilidade da utilização do mecanismo padrão de serialização de Java, não sendo mais obrigatório implementar estes dois métodos. No entanto, existe uma perda de desempenho quando comparadas a serialização padrão de Java e a possibilidade de desenvolver a serialização específica do componente. Desta forma, caso vá utilizar a serialização padrão de Java, o usuário deverá estender a classe abstrata *AbstractComponent*, na qual só existem dois métodos para serem sobrescritos: **(i)** *getId*; e **(ii)** *execute*.

Como fim, o método *execute* produzirá o resultado do processamento do componente. Este resultado será armazenado em um espaço compartilhado de dados, onde todos os componentes poderão acessar via *framework* do *EasyP*. Desta forma, tendo acesso ao processamento de todos os outros componentes executados pelo *framework*. Este espaço compartilhado é organizado como sendo um mapeamento de chave-valor. Desta forma, para poder realizar o acesso a outros componentes, o componente em questão deverá solicitar ao *framework* o acesso a um determinado resultado através do identificador do componente desejado. Este acesso ocorre inicialmente apenas a nível local do *framework*. Porém, em caso do espaço local não possuir tal resultado, o *framework* fica responsável por procurar o resultado em outras máquinas do ambiente. O acesso a este espaço pode ser de três maneiras: **(i)** *writeResult* (linha 3) para realizar a escrita no espaço de um determinado dado que está associado com uma chave única; **(ii)** *readResult* (linha 4) realizando a leitura de um dado associado a uma chave; e **(iii)** *takeResult* (linha 5) na qual também realiza a leitura de um dado associado a uma chave, no entanto remove do espaço em questão este dado.

```
1 public abstract class Framework {
2   ...
3   public static void writeResult(long componentId, Object result) {...}
4   public static Object readResult(long componentId) {...}
5   public static Object takeResult(long componentId) {...}
6 }
```

Outro fator importante no envio dos componentes para serem executados em alguma unidade de processamento remoto, é o envio da definição do componente. Para um ponto de processamento ter a capacidade de executar um componente recebido pelo canal de comunicação, é necessário que este ponto seja capaz de realizar a montagem deste componente. Para isto, é necessário que este ponto possua a definição da estrutura do componente carregado em memória. Nos trabalhos anteriores, a solução deste problema era resolvido ou preparando o ambiente de execução previamente, adicionando a cada unidade de processamento a definição dos componentes que poderão ser executados; ou a inicialização remota dos aplicativos, carregando remotamente as definições dos componentes a serem executados.

No entanto, a proposta do *EasyP* é detectar se o ponto remoto reconhece ou não o componente e em caso negativo, serializa a definição do componente e o envia pelo canal de comunicação para este ser carregado em memória. Na Figura 4.4, ilustra o processo do envio do componente, onde pode-se verificar que inicialmente é analisada a capacidade do ponto remoto reconhecer o componente a ser enviado, caso não consiga reconhecer a definição do componente é enviada. Quando o ponto remoto de processamento já reconhece o componente, apenas os dados do componente são enviados, sendo então re-montados no ponto remoto e finalmente processados. No caso da verificação inicial já identificar que o ponto remoto de execução é capaz de reconhecer o componente, apenas os dados do componente são enviados, não é mais

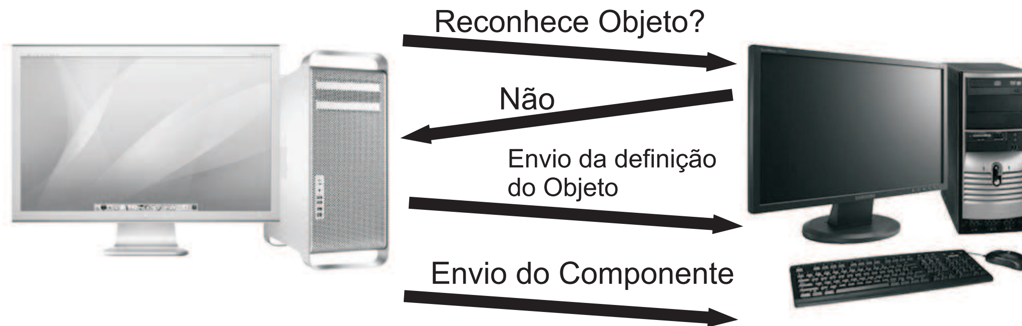

Figura 4.4: Fluxo de atividades do envio de um componente para ser executado remotamente.

enviando a definição para ser carregada previamente.

4.1.5 Camada de abstração

Em um ambiente com mais de uma unidade de processamento, um problema surge logo de inicio, a comunicação entre estas unidades. Como ocorrerá esta comunicação e de que forma será a transmissão de dados. Questões de confiabilidade, segurança e desempenho são levadas em consideração ao se projetar a comunicação entre unidades de processamento [36]. Dependendo da tecnologia de comunicação utilizada, várias limitações podem ser impostas ao aplicativo ou ao ambiente a ser processado.

Vários estudos prévios [11, 40, 57, 56] com o intuito de definir uma ferramenta para a distribuição de processamento são limitadas em relação ao ambiente em que o aplicativo pode executar. Um dos motivos é a escolha do tipo de comunicação proposto em seus trabalhos. No entanto, Niuewpoort [58] propôs uma ferramenta com uma camada de intermédio, entre o aplicativo e o canal de comunicação entre as unidades de processamento. Esta camada de abstração permite que diversos tipos de comunicações sejam realizadas, inclusive, o desenvolvimento de novas implementações do canal de comunicação, permitindo sempre que exista uma versão da comunicação mais apropriada para o ambiente ao qual será processado o aplicativo.

Do estudo de Nieuwpoort, foi retirada a idéia da arquitetura da proposta do *EasyP*, na qual pode ser visualizada na Figura 4.5. Como pode ser visto, é uma arquitetura simples e em camadas [55]. A utilização desta arquitetura permitiu a abstração da camada de comunicação pelo aplicativo, desta forma, o aplicativo vai interagir da mesma forma com o *framework* independentemente de como esteja implementado o canal de comunicação abaixo do *framework*.

Esta arquitetura, embora simples, permitirá com que o usuário possa realizar facilmente a seleção de qual tipo de comunicação será utilizada. Esta troca de comunicação permitirá a comunicação se ajuste da melhor maneira com o ambiente existente e que forneça uma melhor confiabilidade ou uma melhor performance. Além disto, é possível devido a esta camada de abstração, que um ponto do ambiente se comunique com outro utilizando um determinado protocolo de comunicação e com um terceiro ponto usando outro protocolo de comunicação completamente diferente, como pode ser visto na Figura 4.6.

Assim sendo, o *framework* foi desenvolvido para facilmente definir qual será o

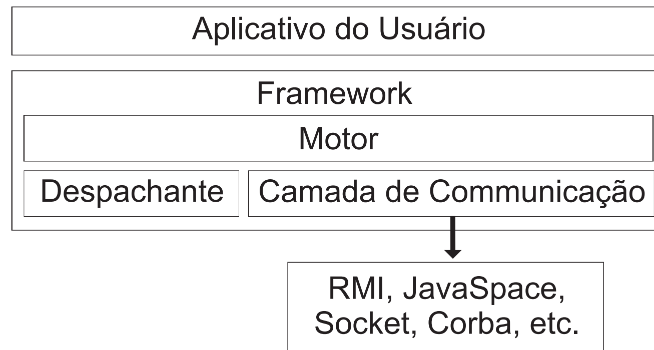

Figura 4.5: Arquitetura do framework proposto

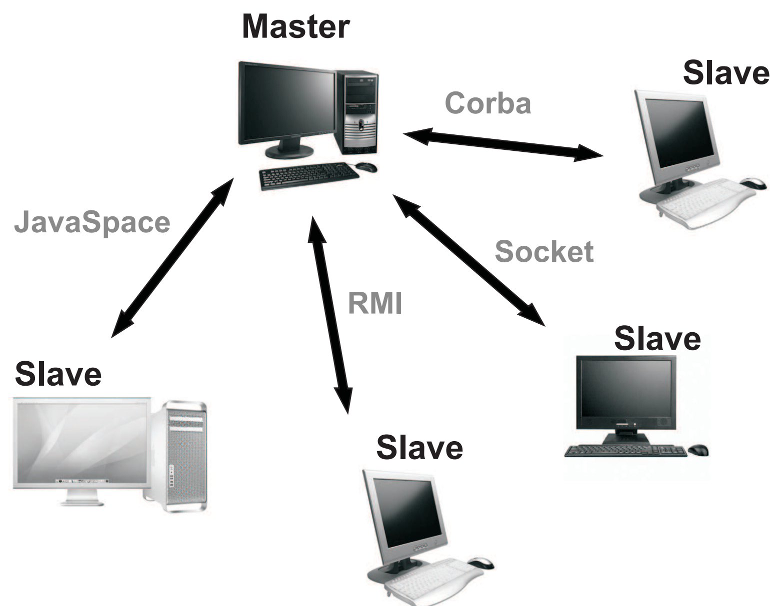

Figura 4.6: O *framework* permite a diversidade de protocolos de comunicações utilizados ao mesmo tempo.

tipo de comunicação que será utilizada na transmissão de dados entre as unidades de processamento. Para isto, basta apenas modificar o parâmetro que identifica a comunicação para que o motor de processamento do *EasyP* já passe a utilizar outra comunicação sem ter a necessidade de alterar mais nada. Este parâmetro é utilizado pelo *framework* para realizar a criação da comunicação. Para isto, é utilizado o padrão de software [19] *Factory Method*. O código da interface da fábrica é definida a baixo. Onde pode ser visto, que o único método existente é o da instanciação do objeto responsável pela comunicação, linha 2.

```
1 public interface ICommunicationFactory {
2   public Communication createCommunication(String communicationType);
3 }
```

Como pode ser visto, a fábrica cria um objeto da classe *Communication*, na qual é uma classe abstrata que implementa a interface *ICommunication* que encontra-se listada a baixo. Na classe abstrata, todos os métodos de recebimento de mensagens já encontram-se implementados, passando a responsabilidade do tratamento da mensagem para o motor do *EasyP*.

Assim, a implementação da comunicação deverá definir 11 métodos da interface *ICommunication*: **(i)** *getType* (linha 3), que deverá retornar a chave que identifique o tipo da comunicação a ser realizada; **(ii)** *createCommunicationId* (linha 4), cria a identificação de comunicação do ponto de processamento no ambiente; **(iii)** *startCommunication* (linha 5), tenta inicializar o serviço de comunicação dentro do ambiente distribuído; **(iv)** *finishCommunication* (linha 6), encerra o serviço de comunicação; **(v)** *sendComponent* (linha 7), envia o componente solicitado para ser processado em uma unidade de processamento remota; **(vi)** *sendResult* (linha 8), envia o resultado do processamento de algum componente para outra unidade de processamento; **(vii)** *requestResult* (linha 9), invocado quando uma unidade de processamento necessita de um resultado no qual não foi esta unidade que produziu, então é enviada uma requisição para o ponto no qual processou o componente específico; **(viii)** *notifyProcessStatus* (linha 10), enviar o estado atual do processo remoto; **(ix)** *sendIsAlive* (linha 11), realizar a validação da unidade de processamento, retornando se esta unidade está apta para realizar algum processamento; **(x)** *sendHasClazz* (linha 12), verifica na unidade de processamento remoto se a definição do componente está carregada em memória; e **(xi)** *sendClazz* (linha 13), envia a definição do componente para ser carregado em memória pela unidade remota.

```
1  public interface ICommunication {
2    void setEngine(...);
3    String getType(...);
4    StringBuffer createCommunicationId(...);
5    boolean startCommunication(...);
6    boolean finishCommunication(...);
7    void sendComponent(...);
8    void sendResult(...);
9    Object requestResult(...);
10   void notifyProcessStatus(...);
11   boolean sendIsAlive(...);
12   boolean sendHasClazz(...);
13   boolean sendClazz(...);
14   void receiveComponent(...);
15   void receiveResult(...);
16   Object receiveRequest(...);
17   void receiveProcessStatus(...);
18   boolean receiveIsAlive(...);
19   boolean receiveHasClazz(...);
20   boolean receiveClazz(...);
21 }
```

Desta forma, caso o usuário deseje definir um novo canal de comunicação específico para o seu aplicativo, possibilitando extrair um maior potencial de seu ambiente distribuído, basta implementar esta nova comunicação baseando-se na classe abstrata *Communication* e modificar a fábrica dos objetos responsáveis pela criação do canal de comunicação. Já ficando disponível o uso desta nova comunicação para ser utilizada pelo *framework*.

4.1.6 Política de despacho

Ao realizar a componentização de seu aplicativo, o usuário poderá definir componentes com tempos de processamento diferentes. Juntando a isto a capacidade

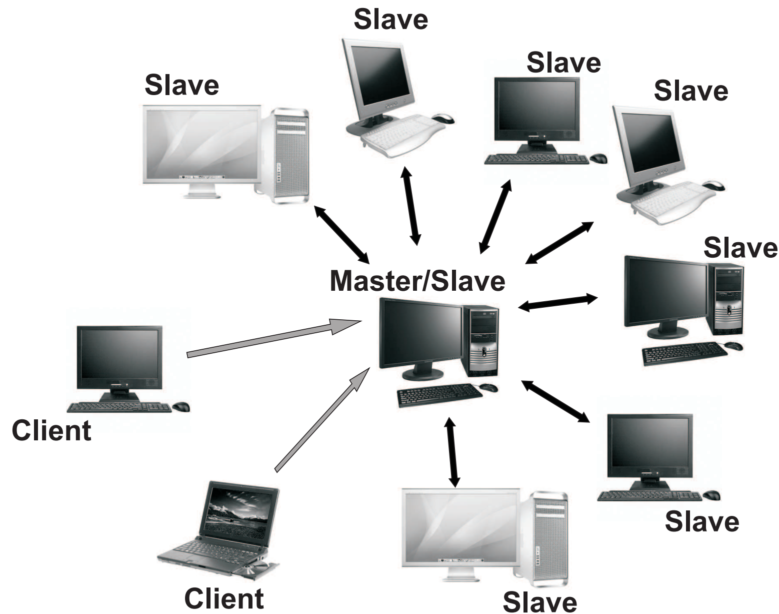

Figura 4.7: Ambiente de processamento heterogêneo.

de encontrar ambientes totalmente heterogêneos como ilustrado na Figura 4.7, na qual pode ser visto um ambiente fictício com a presença de máquinas completamente diferentes. Esta diferença entre máquinas e entre componentes provavelmente acarretará uma má distribuição de carga de processamento. Permitindo então que máquinas mais simples executem componentes mais complexos e máquinas mais poderosas processem componentes mais simples. Esta característica resultará em prováveis momentos em que máquinas permaneçam sem processarem nada enquanto outras máquinas continuem realizando seus processamentos, o que podemos definir como um má balanceamento de carga de processamento [33].

Desta forma, um bom balanceamento da carga de processamento dos componentes poderá resultar em um ganho de performance. Para isto, é necessário realizar um estudo prévio de seu aplicativo e consequentemente, a definição de uma escala de processamento dos componentes a serem executados e suas respectivas máquinas que estarão responsáveis por realizar tal processamento. A Figura 4.5 ilustra um componente da arquitetura que é denominado *Despachante*, no qual é responsável por abstrair a implementação da política de despacho de envio dos componentes a serem executados. Assim, o usuário poderá realizar a implementação de sua própria política de despacho sem a necessidade de alterar mais nada no *framework*. Para realizar esta implementação, o usuário terá que implementar uma interface Java, vista abaixo, e passar esta implementação na inicialização do framework.

```
1 public interface IDispatcher {
2   public void notifyThreadModification(...);
3   public void setThreadsSupport(...);
4   public IThread getThreadToExecute(...);
5   public void setConcurrencyUtil(...);
6 }
```

Como pôde ser visto, existem quatro métodos que devem ser implementados para poder desenvolver sua própria política de despacho, que são: **(i)** *notifyThreadModification* (linha 2) método no qual o *framework* notifica ao

despachante que um ponto de processamento teve seu estado modificado, como por exemplo, um componente terminou de ser processado e o ponto de processamento está livre novamente para receber outros componentes para serem processados; **(ii)** *setThreadsSupport* (linha 3) este método configura o despachante definindo os objetos de suporte e gerenciamento dos pontos de execução, ou seja, os objetos nos quais irão disponibilizar ao despachante o acesso aos pontos de execução; **(iii)** *getThreadToExecute* (linha 4) método principal do despachante no qual vai ser acessado pelo *framework* que irá solicitar o próximo ponto de processamento para executar um componente específico; e **(iv)** *setConcurrencyUtil* (linha 5) um objeto auxiliar do despachante que deve ser configurado para poder realizar operações de concorrência a nível do *framework*, como fazer com que este entre em estado de espera.

Desta forma, o usuário terá apenas que definir a lógica do seu balanceamento de carga de processamento de acordo com os identificadores dos componentes a serem executados. Pois no método de selecionar o próximo ponto de processamento (*getThreadToExecute*) a executar um componente, o despachante saberá para qual componente está sendo realizada a requisição, escolhendo assim, o ponto do ambiente mais apropriado a processar tal componente.

4.1.7 Usabilidade

Como dito anteriormente, a proposta do *EasyP* procura abstrair algumas das dificuldades do desenvolvimento de aplicações distribuídas, como a distribuição do processamento. Porém, o *EasyP* não está focado em desempenho e sim em facilidade de uso. Procurando permitir que usuários que não desenvolviam seus aplicativos adaptados para sistemas paralelos/distribuídos devido a dificuldade, possam usufruir de um ambiente com mais de uma máquina sem muito esforço. Para isto, todas as manipulações com o *framework* foram simplificadas ao máximo, tentando sempre ter uma configuração mínima para o seu uso correto.

Como por exemplo, vimos que para o usuário do *framework* decidir qual tipo de protocolo de comunicação será utilizada, basta modificar o parâmetro de inicialização do serviço. Toda a configuração necessária para que a comunicação ocorra corretamente e a correta identificação dos demais pontos existentes no ambiente é realizado em baixo nível pelo motor do *EasyP* ficando completamente transparente para o usuário.

Além disto, havendo a necessidade do desenvolvimento de um novo protocolo de comunicação para suportar da melhor forma o ambiente, existe uma classe na qual deve apenas ser estendida e implementada corretamente, além de modificar a fábrica da comunicação para refletir essas modificações realizadas. Feitas estas modificações, já estará disponível para ser usado este novo protocolo de comunicação, sem a necessidade de mais configurações.

Enquanto que a nível de ambiente, não existe a necessidade de nenhuma configuração no ambiente já existente para realizar a adição de uma nova máquina. Basta apenas inicializar corretamente esta nova máquina, apontando-a para o *Master* no qual irá conectar-se e esta máquina será reconhecida automaticamente e estará disponível para o processamento dos componentes do aplicativo.

Quando passamos a analisar o aplicativo a ser processado, se este já tiver sido bem desenvolvido a nível de objetos, não se haverá grandes dificuldades para

enquadrá-lo no padrão de componentes do *EasyP*. Pois, é de se esperar que já se tenha o processamento do aplicativo sendo realizado por objetos da linguagem Orientada a Objetos, necessitando apenas que o objeto implemente a interface do padrão definido pelo *framework* e a modificação para colocar todo o processamento do objeto em apenas um método que seria o método *execute*.

Além de que, no caso de um componente necessitar do resultado do processamento de um outro componente, basta este realizar a solicitação do resultado ao *framework*, apenas enviando o identificador do componente. Caso o ponto não conheça o componente, o próprio motor do *EasyP* realiza a busca do componente nos outros pontos de processamento do ambiente. Ao achar, caso este já tenha sido processado, já envia o resultado para o componente solicitante, ou em caso do resultado ainda está sendo processado, o motor do *EasyP* entra em estado de espera, esperando o resultado ser produzido.

4.2 Aspectos Arquiteturais do Framework

O desenvolvimento do *framework* apesar de ter como principal meta a abstração das dificuldades da construção de aplicativos distribuídos/paralelos, outros fatores importantes foram levados em consideração, como: escalabilidade do sistema e ambiente, manutenção do código da ferramenta e sua capacidade de adaptar-se a diversas características dos aplicativos dos usuário.

Para poder ter um *framework* com todas estas características, foi necessário uma etapa prévia de análise e estudo da proposta. Como resultado foi obtida uma arquitetura semelhante ao *Ibis*, proposta por Nieuwpoort [58]. Esta simples arquitetura usada para a proposta do *EasyP* pode ser vista na Figura 4.5. Com esta arquitetura em mãos, foi separada por camada a responsabilidade de realizar cada uma das tarefas. Por exemplo, a camada de *Engine* é responsável por procurar o ponto de processamento que esteja apto a processar o componente e então entregar ao objeto abstrato do *framework* que representa este ponto, caso este ponto seja remoto, o próprio ponto de processamento irá se comunicar com a camada de abstração e para realizar o envio do componente para o ponto de processamento real do ambiente para este então ser processado.

Como pode ser visto na Figura 4.8 que representa o diagrama UML de sequência do envio de um componente para ser processado, inicialmente o aplicativo do usuário entrega ao *framework* o componente para ser processado. Em seguida o *framework* solicita ao despachante que encontre o ponto de processamento que deverá realizar o processamento deste componente em específico e então o motor do *EasyP* entrega ao objeto abstrato, ainda local, que representa este ponto de processamento. Caso este ponto seja remoto, este objeto abstrato entra em contato com a camada de abstração da comunicação para enviar o componente utilizando o canal de comunicação existente. Ao terminar o envio do componente, o *framework* já fica disponível para receber outro componente e realizar o mesmo processo, buscando um ponto de processamento para realizar a computação necessária pelo componente e enviá-lo para ser processado por outro ponto. Ao término deste processamento, o resultado é enviado para o processo mestre para o resultado ser armazenado e torná-lo disponível para outros componentes.

Após ter definido corretamente a sequência das atividades a serem executadas pelo *framework*, o próximo passo foi a definição abstrata da organização do *EasyP*.

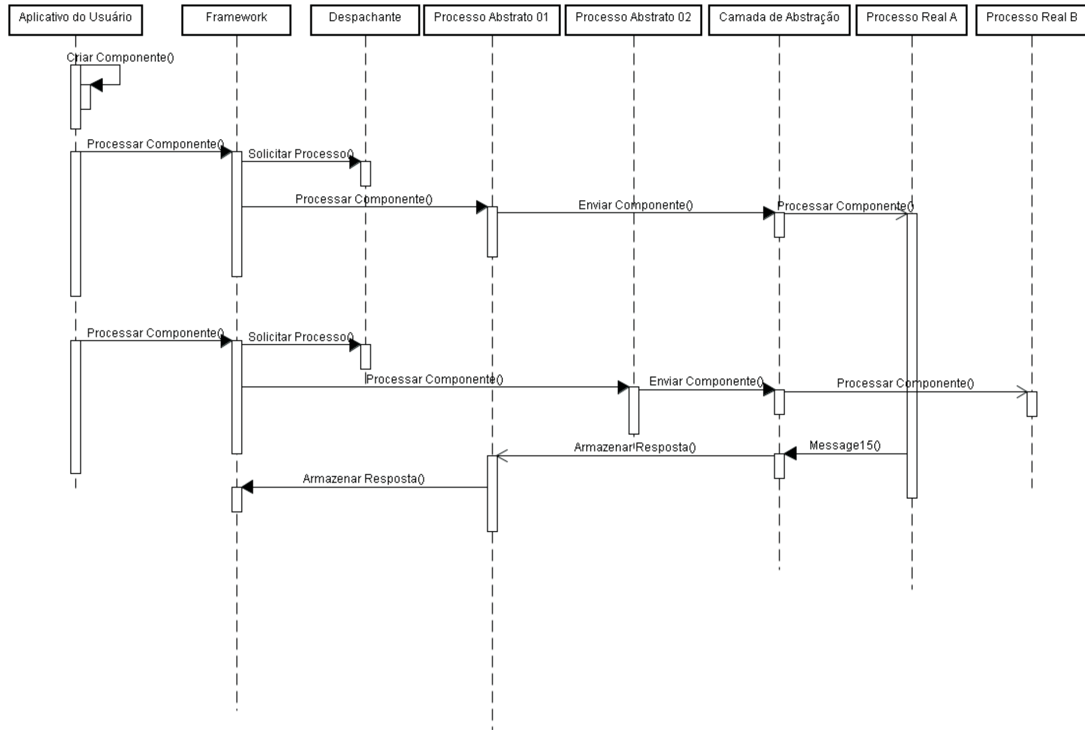

Figura 4.8: Diagrama UML de sequência do *framework* proposto.

Para isto, foi detalhado o diagrama de classes que irá representar o *EasyP*. Este diagrama foi modificado durante o desenvolvimento da ferramenta, mas a versão final é visualizada na Figura 4.9 [2], onde pode ser visto em cinza escuro a interface que permite que o usuário modifique a política de despacho de seus componentes, além de que em cinza claro encontra-se as classes que definem o padrão dos componentes a serem utilizados pelo *framework*

Neste diagrama podemos ver o motor do *framework* (classe *Engine*) como sendo a classe central do *framework* proposto. A partir dela, existe a conexão com todas as outras ramificações e possibilidades de uso do *framework*, como o despachante (classe cinza escuro no diagrama) e a camada de abstração, que no diagrama é visto como a classe *Service*. Outro fato interessante é que todos os processos são representados abstratamente como uma implementação da interface *IProcess*, e que apenas a classe *LocalThread* representa realmente um processo físico, no qual é o único tipo de processo que possibilitará o real processamento do componente. Pois até mesmo o *LocalProcess* é uma implementação abstrata de um processo local, necessitando entrar em contato com a camada de abstração para enviar o componente para ser processado por uma *LocalThread*. Além disto, podemos visualizar as classes que são responsáveis por padronizar os componentes a serem processados pelo *framework*, como a *IDistributedComponent* e *DistributedComponent* (classes cinza claro no diagrama), respectivamente a interface para um componente completamente desenvolvido pelo usuário e uma classe abstrata que facilita o desenvolvimento dos componentes, como por exemplo não havendo a necessidade da implementação da serialização específica do componente.

[2]Para seguir padrões internacionais, todo o desenvolvimento do aplicativo foi realizado na língua inglesa, como pode ser visto no diagrama de classes onde cada classe é denominada em inglês.

Figura 4.9: Diagrama UML de classes do *framework* proposto. De preto está representada a classe externa do *framework* proposto..

Vendo o diagrama de classes, pode ser observado que existem *Threads* de Java no motor do *framework*. Estas *Threads* são responsáveis por ficarem monitorando a conexão de novas máquinas sendo conectadas em tempo de execução. Esta conexão inicial ocorre através de uma conexão simples via *Sockets*, na qual são enviadas as primeiras informações da máquina remota, tornando disponível esta máquina para ser utilizada no processamento de novos componentes.

4.3 Validação da Utilização

O aplicativo proposto por Durães é preparado para ser executado de maneira distribuída, por isso o mesmo já está bem dividido em trechos de código que serão processados remotamente. Desta forma, o custo não vai ser alto, necessitando apenas encapsular este trecho em um componente de acordo com as restrições impostas pela proposta do *EasyP*. Além disto, é necessário a correta inicialização do *framework*.

Além desta inicialização foi necessário identificar onde, na ferramenta de Durães, realiza a distribuição do processamento. Tendo identificado este ponto no código, foi realizado apenas a substituição do código pela chamada ao *framework* enviando os componentes que foram criados anteriormente. Após isto, foi identificado o momento que se fazia uso do resultado do processamento de cada componente e modificado para a leitura destes resultados a partir do *framework*. Procurando sempre, não modificar a lógica do código nem alterar mais nada que não fosse necessário.

4.3.1 Inicialização do *EasyP*

Para poder utilizar o *EasyP* é necessário realizar a correta inicialização deste. Para isto, primeiramente é necessário preparar o ambiente identificando quais serão as

máquinas que irão compor este ambiente de processamento. Com esta identificação pronta é necessário planejar como estarão organizadas as máquinas, se haverá a necessidade de mais de uma máquina prestar o serviço de *master* e quais irão prestar o serviço de *slaves*.

Após todo o planejamento, o próximo passo é realizar a inicialização dos serviços em cada uma das máquinas do ambiente. Para isto, deve ser criado um aplicativo Java que instancia uma representação do *framework* atuando com o seu respectivo papel. Tal inicialização é feita utilizando uma fábrica de objetos do *EasyP*. O trecho de código a seguir mostra dois métodos Java nos quais inicializam uma máquina *master* e outra *slave*, assumindo que o endereço IP da primeira máquina é *172.10.2.1*.

```
1  /* Inicialização do serviço Master */
2  public static void main(String args) {
3    ICommunicationFactory comFactory = new CommunicationFactory();
4    IDispatcher dispatcher = new Dispatcher();
5    Framework framework = FrameworkFactory.create("masterPoint",
         comFactory, dispatcher);
6    int porta = 2000;
7    long timeout = 1000;
8    framework.startRemoteMasterService(porta, timeout);
9  }
10
11 /* Inicialização do serviço Slave */
12 public static void main(String[] args) {
13   ICommunicationFactory comFactory = new CommunicationFactory();
14   IDispatcher dispatcher = new Dispatcher();
15   Framework f = FrameworkFactory.create("slavePoint", comFactory,
         dispatcher);
16   f.startRemoteSlaveService("masterPoint", "172.10.2.1", 2000,
         RMICommunication.TYPE);
17 }
```

Em ambos os casos foram inicializadas uma fábrica para gerenciar a criação das comunicações utilizadas pelo *framework* (linhas 3 e 13) e um despachante para definir em qual unidade de processamento deverá ser processada cada um dos componentes (linhas 4 e 14). Após isto, é preciso criar uma instância do *framework* utilizando a classe *FrameworkFactory* (linhas 5 e 15). Nesta chamada de método, deve ser passado o nome da máquina no ambiente, a fábrica da comunicação e o despachante a serem utilizados. No caso de criar um serviço como *master* é necessário realizar uma chamada ao método *startRemoteMasterService* (linha 8) e passar os parâmetros que definem qual a porta utilizada para a conexão de outras máquinas (linha 6) e o tempo utilizado na validação das máquinas conectadas (linha 9). No caso de um serviço *slave* basta fazer uma chamada ao método *startRemoteSlaveService* (linha 16) passando como parâmetros o nome da máquina no ambiente, o endereço IP da máquina a qual vai se conectar (no exemplo 172.10.2.1), a porta da máquina a qual vai se conectar (porta 2000) e o tipo da comunicação que vai ser utilizada (no exemplo, RMI).

Porém, tais inicializações, só representam a inicialização dos serviços de comunicação entre as máquinas, para poder habilitar uma máquina para realizar o processamento, deve ser inicializado o serviço abaixo.

```
1 int numeroProcessos = 2;
2 framework.startLocalService(numeroProcessos, RMICommunication.TYPE);
```

Nesta chamada deste método, já deve ter sido instanciado corretamente um

objeto do *framework* e então realizar a chamada do método *startLocalService* (linha 2). Este método recebe como primeiro parâmetro a quantidade de unidades de processamento a serem inicializadas na máquina, porém este número é validado pelo *framework* e caso não seja possível inicializar esta quantidade, o número máximo que for possível será utilizado (neste exemplo, caso fosse uma máquina com apenas uma unidade de processamento disponível, iria ser inicializada apenas um processo). O segundo parâmetro é qual será o estilo de comunicação local com estes processos, no exemplo será utilizado RMI.

Na inicialização do cliente do *framework*, deve ser informado o identificador do ponto de processamento ao qual irá conectar, além de detalhes da conexão, como: **(i)** o IP da máquina mestre do ambiente; **(ii)** a porta que será utilizada na comunicação com o mestre; e **(iii)** o tipo da comunicação, no exemplo é utilizado RMI como protocolo de comunicação com o mestre.

```
1 framework.startClientService("masterPoint", "172.10.2.1", 2000,
      RMICommunication.TYPE);
```

No caso de desejar utilizar qualquer outro tipo de comunicação, o usuário deverá apenas alterar o parâmetro de inicialização dos exemplos anteriores, nos quais foi utilizado a comunicação *RMI*, pela comunicação desejada. Além disso, será necessário utilizar apenas uma fábrica da comunicação que permita criar os objetos da comunicação correspondente.

4.3.2 Adaptação do código de processamento

Para o correto uso do *EasyP* é necessário primeiramente definir quais os pontos do aplicativo pode ser distribuído. Como visto no Capítulo 4, após esta identificação é necessário que estes trechos de códigos sejam adaptados para implementarem uma interface Java proposta pela nossa ferramenta. Caso o aplicativo seja desenvolvido orientado a objetos, é provável que toda a computação destes trechos de códigos que podem ser distribuído, sejam representados como um único objeto ou que pelo menos seja representado como um conjunto de objetos que tem sua computação iniciada com a chamada a um método específico. Desta forma, fica fácil a adaptação desses objetos em um componente capaz de ser distribuído pelo *EasyP*.

Como exemplo desta adaptação, será mostrado um código simples que pode ser analisado abaixo. Imagine uma classe Java responsável por realizar um processamento complexo, neste exemplo, uma soma de dois números x e y. Esta classe possui um método de início de seu processamento, novamente realizando a analogia no exemplo, este método é denominado *resultado* (linha 10).

```
1  public class Somar {
2    private int x;
3    private int y;
4
5    public Somar(int x, int y) {
6      this.x = x;
7      this.y = y;
8    }
9
10   public int resultado() {
11     return (x + y);
12   }
13 }
```

Tendo um objeto Java bem definido como o exemplo mostrado, para adaptá-lo para a ser usado com o *EasyP* não é uma tarefa complexa. A maneira mais simples é encapsular este objeto inicial em um segundo objeto que implemente a interface *IDistributedComponent* e *Serializable*. Adicionar o atributo de identificação do objeto com seu respectivo método de ler este atributo (*getId()*) e implementar o método *execute* como sendo uma chamada ao método que realiza o processamento do objeto que existia inicialmente, como pode ser visto a seguir.

```
1 public class SomarComponente implements IDistributedComponent ,
      Serializable {
2   private long id;
3   private Somar somar;
4
5   public SomarComponente(long id, int x, int y) {
6     this.id = id;
7     this.somar = new Somar(x, y);
8   }
9
10  public Object execute() {
11    return somar.resultado();
12  }
13 }
```

Caso o usuário deseje obter um melhor desempenho, é aconselhável que implemente a outra interface, a *IPerformanceComponent*, podendo assim implementar sua própria *serialização* do objeto. Porém, a implementação do componente utilizando esta interface é um pouco mais complexa, necessitando realizar a transformação de todos os dados do componente para um *stream* de bytes. Além de ser necessário realizar o processo inverso, transformando os bytes de volta a um objeto.

Após realizar esta conversão do objeto do aplicativo para um componente de acordo com as necessidades do *framework*, não será mais necessário nenhuma alteração quanto ao objeto, havendo a necessidade de modificar apenas a maneira como será lido o resultado do processamento. Isto porque, ao terminar o envio do componente para o *framework* o aplicativo do usuário estará livre para realizar outras atividades. Desta forma, o aplicativo não recebe diretamente o resultado do processamento do componente que foi entregue ao *framework* para ser processado. Para realizar o acesso a este resultado, o aplicativo do usuário deverá realizar uma solicitação deste resultado ao *framework*, pois, ao terminar o processamento de um componente, o *framework* armazena o resultado em um espaço compartilhado, sendo acessível apenas através do *framework*. Para realizar este acesso, existem duas possibilidades: **(i)** *readResult*, que apenas realiza a leitura do resultado; e **(ii)** *takeResult*, no qual realiza a leitura do dado e apaga este do espaço do *framework*.

Caso um componente necessite do resultado de um segundo componente, o primeiro deve realizar a chamada ao *framework* passando o identificador do componente que o *framework* fica responsável em buscar este resultado e entregá-lo corretamente, abstraindo assim a comunicação entre os componentes. Por exemplo, considere um componente responsável por realizar um processamento dependente de outro componente, como o componente *DividirComponente* que necessita do resultado da soma de outros dois valores, o mesmo deverá realizar uma chamada com o identificador do outro componente, como ilustrado a seguir:

```
1 public class DividirComponente implements IDistributedComponent ,
      Serializable {
2   private long id;
3   private int x;
4   private int n;
5
6   public DividirComponente(long id, int x, int n) {
7     this.id = id;
8     this.x = x;
9     this.n = n;
10  }
11
12  public Object execute() {
13    int div = (int)Framework.readResult(n);
14    return div / x;
15  }
16 }
```

Como visto, este componente realiza a divisão do resultado do componente de identificador igual ao valor da variável n (linha 13) pelo valor da variável x (linha 14). Todos os resultados inseridos dentro do *framework* são armazenados como um objeto Java, desta forma será necessária a conversão (*cast*) para o tipo correspondente (linha 13).

4.4 Conclusões

Neste capítulo foi visto a idéia da proposta do *EasyP* que está sendo apresentado ao longo desta dissertação além de detalhes da sua implementação. O *EasyP* procura ao máximo abstrair as dificuldades encontradas no desenvolvimento de aplicações distribuídas, procurando assim facilitar o desenvolvimento de versões de aplicativos para usarem ambientes com mais de uma unidade de processamento.

Entre as abstrações realizadas, está a camada de abstração existente entre o *framework* e a comunicação realizada entre o aplicativo e o protocolo de comunicação usado na comunicação. Esta camada de abstração permite a troca fácil e a utilização de qualquer tipo de protocolo para realizar a comunicação entre as unidades de processamento. Desta forma o usuário poderá optar por uma tecnologia que lhe dará mais segurança, confiabilidade ou performance, necessitando apenas a troca de um único parâmetro da inicialização do ponto de processamento.

Outra camada da arquitetura no mesmo nível da camada de abstração da comunicação é a do despachante. Este componente do *framework* permite que o usuário defina sua política de despacho e realize um melhor balanceamento de carga de processamento ao distribuir de melhor forma os componentes de seu aplicativo a ser processado.

Além disto, a utilização do *framework EasyP* é simples, necessitando apenas o usuário realizar a quebra do seu aplicativo em partes menores de processamento e encapsulá-los em um componente atendendo as necessidades impostas pelo *EasyP*. Desta forma, estará apto a enviar os componentes do aplicativo para o *Master* do ambiente e este os distribuir para serem executados remotamente.

Outra facilidade imposta pelo *EasyP* vista neste capítulo, foi a capacidade de adicionar ou remover novas máquinas ao ambiente em tempo de execução. Esta característica permite a utilização de máquinas mais potentes, mesmo que

por tempo reduzido, em casos de serem máquinas muito requisitadas. Além do *framework* realizar também o gerenciamento do resultado do processamento de cada componente. Os guardando em um espaço compartilhado e sincronizando este espaço entre todos os outros pontos de processamento do ambiente, sem a necessidade do usuário se preocupar com isto.

Acrescentado a explanação das idéias por trás da proposta, foram apresentados detalhes de sua implementação, como diagramas UML e explicações mais detalhadas de como é possível para o *EasyP* proposto realizar a abstração na qual se propõe.

Capítulo 5

Estudo de Caso

Don't waste time or time will waste you.

Muse - Knights of Cydonia

Para validar a proposta e demonstrar o uso do *EasyP* foi realizado um estudo de caso no qual foi utilizada a ferramenta utilizada no estudo de Durães [11], realizando uma alteração para que esta passasse a utilizar o *EasyP* para a distribuição do processamento. Como visto no Capítulo 3, o estudo de Durães é baseado na simulação do roteamento de redes ópticas transparentes, porém sendo uma versão distribuída de um simulador desenvolvido previamente. Desta forma, por já se tratar de um aplicativo preparado para um ambiente distribuído será mais simples a adaptação, necessitando apenas da modificação do método de distribuição.

Neste estudo de caso realizado, foi necessário uma modificação na aplicação de Durães para que ao invés de utilizar seu sistema de distribuição, passasse a utilizar o *framework* de distribuição do *EasyP*. Esta foi uma modificação pontual, sem a necessidade de alterar diversos trechos do código do aplicativo. Esta modificação vai ser detalhada ao longo deste capítulo.

Uma segunda validação realizada sobre o nosso estudo é justamente o impacto desta modificação no desempenho prévio da ferramenta. Para isto, montamos alguns ambientes de testes e passamos a realizar um estudo comparativo entre o desempenho do aplicativo anterior e após o uso do *EasyP*. Como já mencionado anteriormente, é de se esperar que este resultado mostre uma certa perda de desempenho, pois a proposta *EasyP* apresenta uma maior abstração além de ser para um uso mais genérico.

No restante do capítulo será detalhado mais sobre este estudo de caso, além de caracterizar melhor o uso do *framework EasyP*, como também mostrar um comparativo com o desempenho utilizando e não utilizando o *EasyP*.

5.1 Análise do Desempenho

Para avaliar o impacto no desempenho do *EasyP*, foi realizado alguns experimentos comparativos e uma posterior análise estatística utilizando o *t-test* [1]. Para o

[1] O *t-test* é um teste estatístico que compara o valor médio entre dois conjuntos de dados e analisa se suas diferenças são ou não significativas [62]

Tabela 5.1: Ambientes utilizados nos experimentos realizados.

Simulação	Ambiente	Unidades de Processamento
01	01 Máquina equipada com Intel Core 2 Quad 6600 com 4Gb ddr2-800MHz de memória RAM	01
02	01 Máquina equipada com Intel Core 2 Quad 6600 com 4Gb ddr2-800MHz de memória RAM	04
03	02 Máquinas equipadas com Intel Core 2 Quad 6600 com 4Gb ddr2-800MHz de memória RAM	08

experimento, foi utilizado o simulador desenvolvido no trabalho de Durães [11] como caso base e realizada uma modificação para que não utilize mais o sistema de distribuição idealizado no trabalho de Durães e passasse a utilizar o *EasyP*. Esta alteração foi realizada de forma pontual para que o simulador não sofra influência destas alterações.

Além deste experimento, foi realizada uma modificação na estrutura do simulador para que este não utilize mais nenhum sistema de distribuição e passasse a ser completamente sequencial, utilizando apenas uma única unidade de processamento para todas as suas operações. Esta modificação foi realizada apenas para obtermos um dado informativo de quanto tempo seria necessário para o simulador chegasse ao seu resultado final, caso este tivesse sido desenvolvido de forma completamente sequencial.

Acrescentando a estes experimentos, utilizamos o *EasyP* para realizar comparações com diversos tipos de protocolos de comunicação. Desta forma é possível analisarmos como se comportaria o *framework* ao trabalhar com protocolos de comunicação diferentes. E ainda foi realizado estes experimentos em diversos ambientes computacionais, para permitir a avaliação da evolução do desempenho da ferramenta quando esta é executada em apenas uma unidade de processamento e em ambientes com mais de uma unidade de processamento. Um resumo destas simulações pode ser visualizada na Tabela 5.1.

5.1.1 Hipóteses

Para a realização dos experimentos foram definidas algumas hipóteses comparativas entre as versões utilizadas em cada um dos experimentos. Para cada uma das hipóteses expostas abaixo, teremos três variações, uma para cada um dos ambientes utilizados, como vistos na Tabela 5.1.

A hipótese principal é considerada a *Hipótese Nula*, onde não existirá diferença entre o desempenho da abordagem realizada por Durães (Ti_D [2].)e a abordagem utilizando o *EasyP* (Ti_F [3]). No entanto, o experimento realizado tentará provar que esta hipótese não é verdadeira, pois apesar de sabermos que seria o ideal (não existir diferença entre as duas abordagens), a proposta do *EasyP* será mais lenta, por se

[2]Tempo gasto no experimento utilizando a abordagem de Durães no ambiente i

[3]Tempo gasto no experimento utilizando o *EasyP* no ambiente i.

tratar de uma abordagem mais abstrata do que a realizada por Durães. Desta forma temos a seguinte hipótese nula:

> **Hipótese Nula** - Não existe diferença entre o tempo gasto para o simulador desenvolvido por Durães realizar seu processamento e o tempo necessário para que o simulador modificado que utiliza o *EasyP* encerre seu processamento.
>
> $H0_1 : T1_D \cong T1_F$
>
> $H0_2 : T2_D \cong T2_F$
>
> $H0_3 : T3_D \cong T3_F$

Acrescentando à estas hipóteses nulas, foram definidas *Hipóteses Alternativas* para serem válidas quando a hipótese nula não for aceita. Desta forma definimos duas hipóteses alternativas:

> **Hipótese Alternativa** - O simulador desenvolvido por Durães gastará um tempo diferente do tempo gasto pelo simulador modificado, no qual utiliza o *EasyP*, para encerrar seu processamento.
>
> $H1_1 : T1_D \neq T1_F$
>
> $H1_2 : T2_D \neq T2_F$
>
> $H1_3 : T3_D \neq T3_F$
>
> **Hipótese Alternativa** - O tempo gasto no simulador desenvolvido por Durães será menor do que o tempo gasto pelo simulador modificado, no qual utiliza o *EasyP*, para encerrar seu processamento.
>
> $H2_1 : T1_D < T1_F$
>
> $H2_2 : T2_D < T2_F$
>
> $H2_3 : T3_D < T3_F$

5.1.2 Analise do experimento

Ao analisar os dados colhidos durante o experimento, a principal ação é rejeitar a hipótese nula, mostrando que existe uma diferença significativa entre os valores obtidos. Desta forma, realizamos o *t-test* com as amostras utilizando a abordagem de Durães e utilizando o *framework* aqui proposto.

Considerando $x_1, x_2, \cdots, x_n$ como as amostras (tempo gasto no processamento da simulação) utilizando a abordagem de Durães, e $y_1, y_2, \cdots, y_n$ como as amostras utilizando o framework proposto, é obtido as médias destas duas abordagens com as equações 5.1 e 5.2.

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{5.1}$$

$$\overline{y} = \frac{1}{m}\sum_{i=1}^{m} y_i \tag{5.2}$$

O próximo passo do *t-test* é definir a distribuição t_0 segundo a equação 5.3.

$$t_0 = \frac{\overline{x} - \overline{y}}{S_p\sqrt{\frac{1}{n} + \frac{1}{m}}} \text{ onde } S_p = \sqrt{\frac{(n-1)S_x^2 + (m-1)S_y^2}{n+m-2}} \quad (5.3)$$

Da equação 5.3 tem-se S_x^2 e S_y^2 que são as variâncias das amostras, calculadas de acordo com as equações 5.4 e 5.5.

$$S_x^2 = \frac{(\sum_{i=1}^{n} x_i^2) - n\overline{x}^2}{n-1} \quad (5.4)$$

$$S_y^2 = \frac{(\sum_{i=1}^{m} y_i^2) - m\overline{y}^2}{m-1} \quad (5.5)$$

A hipótese nula espera que não exista diferença entre as médias obtidas no experimento, porém ao rejeitar a hipótese nula, é encontrado uma diferença diferente de zero, o que significaria que estas médias são comparáveis e não iguais. O teste consiste em verificar se $|t_0| > t_{\alpha,f}$, onde $t_{\alpha,f}$ é a distribuição t com um nível de significância α usando f graus de liberdade, onde $f = n + m - 2$. Para este experimento, foi utilizado um nível de significância de 0.05, ou seja, foi utilizado um nível de confiança de 95%, o que significa que em 95% das amostras possuem seus valores próximos aos valores médios.

5.1.3 Intervalo de confiança

Além de apenas analisar os dados coletados, verificando a diferença entre as médias destes valores, é possível analisar o intervalo de confiança entre estas amostras. Esta análise é uma maneira mais efetiva do que apenas informar se duas médias são ou não diferentes. Se este intervalo entre duas amostras for pequeno, significará um alto grau de precisão, em outro caso, se o intervalo de precisão for largo, indicará que o grau de precisão será muito baixo.

Primeiramente deve-se verificar a validade das hipóteses. Caso a hipótese nula seja falsa, não é necessário verificar o intervalo de confiança, pois já significará que os valores já são significativamente diferentes. Porém, caso a hipótese nula seja verdadeira, devemos avaliar o grau de precisão calculando o intervalo de confiança entre as amostras ao invés de apenas informar que os dois valores não são significativamente diferentes. O intervalo de confiança é calculado desta forma:

1. São calculadas as médias $\overline{x}$ e $\overline{y}$, como definido nas equações 5.1 e 5.2

2. São calculadas as variâncias S_x^2 e S_y^2, como está definido nas equações 5.4 e 5.5

3. É calculada a diferença média: $\overline{x} - \overline{y}$

4. É calculado o desvio padrão da diferença média:

$$S = \sqrt{\frac{S_x^2}{n} + \frac{S_y^2}{m}} \quad (5.6)$$

5. É calculado o grau de liberdade:

$$v = \frac{(S_x^2/n + S_y^2/n)^2}{\frac{1}{n+1}(S_x^2/n)^2 + \frac{1}{m+1}(S_y^2/m)^2} - 2 \quad (5.7)$$

6. É calculado o intervalo de confiança para a diferença média:

$$(\overline{x} - \overline{y}) \mp t_{[1-\frac{\alpha}{2};v]}S \tag{5.8}$$

onde $t_{[1-\frac{\alpha}{2};v]}$ é a variação de t dentro de um intervalo $(1 - \frac{\alpha}{2})$ com um grau de liberdade igual a v

7. Se este intervalo de confiança inclui zero, a diferença não é significante à $100(1-\alpha)\%$ de nível de confiança. Similar ao teste da hipótese nula, foi utilizado um grau de significância de 0.05 que corresponde que existe uma probabilidade de que 95% das amostras estejam dentro do intervalo.

5.1.4 Execução do experimento

Como levantado anteriormente, o experimento foi realizado em três ambientes distintos (Tabela 5.1). Para cada um destes ambientes foram realizados testes utilizando o simulador proposto por Durães e variações do simulador utilizando o *EasyP*, tendo apenas o protocolo de comunicação alterado.

Nestes experimentos foi realizado a medição do tempo gasto deis do início da execução até o simulador chegar em seu estado final. Desta forma, estará sendo verificando se o tempo sofre uma modificação significativa quando o simulador passou a utilizar o *EasyP*, ao invés de utilizar uma proposta de um nível de abstração menor, como é o caso da proposta desenvolvida no trabalho de Durães.

5.1.4.1 Comparação entre propostas (Durães Vs *EasyP*)

Para a realização do experimento foram montados três ambientes como detalhado na Tabela 5.1. Em cada um dos ambientes foram coletadas 10 amostras de medição de tempo da execução do simulador proposto por Durães e a versão modificada utilizando o *framework EasyP* com o protocolo de comunicação Java RMI, desta forma tem-se uma comparação direta da utilização do *EasyP* e a proposta de Durães, pois em ambos os casos a única modificação é o sistema de distribuição, já que ambas também utilizaram o mesmo protocolo: Java RMI.

Além deste experimento, a critério de informação, foi realizada uma modificação no simulador de Durães para que este não utilize nenhum sistema de distribuição e seja executado de forma completamente sequencial. O resultado pode ser visualizado na Tabela 5.2, ficando sua média como valor de referência, no qual em qualquer que seja o ambiente, excluindo-se o ambiente 1, a utilização do *EasyP* não poderia apresentar um comportamento mais lento, devido ao fato de que em outros ambientes o aplicativo passaria a ter seu processamento sendo executado de forma distribuída ao invés de possuir apenas uma única unidade de processamento para realizar a computação necessária do aplicativo.

As Tabelas 5.3 e 5.4 mostram os resultados das medições da execução do simulador proposto por Durães utilizando a proposta dele e utilizando o *EasyP*, em cada um dos 3 ambientes. Como pôde ser visto, mesmo utilizando a versão de Durães (diferença média de -59,93%) ou o *EasyP* (diferença média de -62,68%), os resultados foram muito mais rápido do que a versão completamente sequencial. Esta diferença se dá principalmente pelo fato de que em ambas as versões com sistema de distribuição (Durães e *EasyP*), o processamento dos componentes é realizado em um processo separado do processo de distribuição dos componentes,

Tabela 5.2: Resultado da execução da versão modificada do simulador proposto por Durães. Esta versão processa o simulador de maneira sequencial.

Simulação	Tempo (minutos)
01	318,74
02	318,24
03	317,40
04	318,30
05	319,50
06	316,39
07	317,31
08	318,04
09	317,51
10	316,79
Média	317,77

assim por ter sido utilizado um ambiente com um processador com quatro núcleos, existe um ganho de performance considerável por está utilizando os dois núcleos de maneira paralela, ao invés de apenas um de forma completamente sequencial. Porém, mesmo tendo dois processos nas versões com sistema de distribuição, a execução dos componentes é realizada de forma sequencial, ou seja, um componente deve encerrar seu processamento para somente após iniciar o processamento de um outro componente.

Primeiramente deve ser realizado os testes das hipóteses para identificar se as amostras são ou não significativamente diferentes. A Tabela 5.5 resume essa avaliação da hipótese nula. Quando o teste da hipótese nula for falsa, significa que as amostras são significativamente diferentes, caso contrário as amostras podem ser ditas equivalentes. Para poder validar ou não a hipótese nula deve ser calculado o valor de t_0 (Equação 5.3) e compará-lo com o valor tabelado $t_{\alpha,f}$, onde $f = m+n-2$, neste caso, $f = 10+10-2$ e $\alpha = 0.05$. Então foi realizado a comparação dos valores de t_0 (Tabela 5.6) com o valor tabelado $t_{0.05,18}$, que é equivalente a 2,101.

Para realizar uma melhor analise dos resultados, também foi calculado o intervalo de confiança para os casos em que a hipótese nula foi detectada como verdadeira, para ser analisado o grau de precisão da analise realizada. Este intervalo de confiança também foi calculado com um grau de significância de 0.05, e o resultado deste calculo pode ser visto na Tabela 5.7.

Desta análise dos resultados, pode-se avaliar que o *framework* proposto ao longo desta dissertação não acrescentou um *overhead* significativo nos ambientes 1 e 2 quando comparado à proposta de Durães. Porém, quando reduziu muito o tempo de processamento (ambiente 3), o custo da utilização do *EasyP* já passa a impor uma influência negativa, fazendo com que gaste cerca de 334,22% a mais do que o tempo necessário para a versão de Durães encerrar o processamento.

Este *overhead* é inserido, como esperado, em contrapartida ao custo da abstração inserida, como por exemplo a possibilidade de utilizar qualquer tipo de comunicação. Também é acrescentado o fato de ser necessário realizar um maior processamento procurando uma máquina livre no ambiente para processar o componente. Este *overhead* pode ser considerado fixo, já que quando realizada a comparação entre a versão do simulador proposto por Durães e a versão modificada utilizando o *EasyP* a

Tabela 5.3: Resultados na medição de tempo do simulador original proposto por Durães utilizando seu sistema de distribuição com o protocolo Java RMI. Tempo em minutos

Versão proposta por Durães			
Simulação	Cenário 01	Cenário 02	Cenário 03
01	114,81	29,93	3,97
02	114,57	29,98	4,78
03	114,14	29,94	4,72
04	129,68	29,88	4,76
05	113,97	30,01	4,69
06	138,68	30,03	4,70
07	137,08	30,27	4,78
08	134,58	30,15	4,76
09	135,20	30,01	4,77
10	140,49	29,84	4,70
Média	127,32	30,00	4,66
Diferença	+7,77%	-40,21%	-76,97%

diferença foi muito próxima em todas as amostras coletadas, com um valor próximo dos 20 minutos. Desta análise pode-se concluir que independente do ambiente será inserido um custo no desempenho que resultará em um acréscimo de cerca de 20 minutos no tempo de processamento total do aplicativo.

Destes dados, pode-se concluir que a utilização do *EasyP* resultará em versões com melhor desempenho do que a versão sequencial, independentemente do ambiente utilizado. Porém, quando comparado com versões distribuídas desenvolvidas especificamente para o problema em questão, a versão do aplicativo utilizando o *EasyP* acrescentará uma perda de desempenho não significativa enquanto o tempo o total necessário para o processamento da aplicação for maior do que o tempo previsto para o *overhead* inserido, que segundo o experimento realizado é cerca de 20 minutos. No entanto, o custo de desenvolvimento utilizando o *EasyP* e o desenvolvimento do sistema de distribuição específico para o problema, é incomparável. Utilizando o *EasyP* o desenvolvedor terá que realizar poucas alterações no seu sistema para passar a usufruir de ambientes com mais de uma unidade de processamento.

5.1.4.2 Comparação entre os protocolos

Além do experimento comparando a performance entre a proposta de Durães e a versão do simulador utilizando o *framework EasyP* para realizar a distribuição dos componentes a serem processados, foi realizado também algumas comparações entre a utilização do *EasyP* utilizando o protocolo Java RMI e outros três sistemas de comunicação: **(i)** Java Socket; **(ii)** Corba; e **(iii)** JavaSpace. Para a realização destas comparações foram coletadas amostras de tempo do processamento do simulador modificado, utilizando cada um destes sistemas de comunicação em cada um dos ambientes de processamento detalhados na Tabela 5.1.

A modificação do sistema de comunicação é realizado, como visto anteriormente, modificando apenas um parâmetro da inicialização do *EasyP*. Desta forma, o custo para a troca do sistema de comunicação é próximo do zero. Porém, isto levando em

Tabela 5.4: Resultados na medição de tempo do simulador modificado utilizando o *framework* *EasyP* e o protocolo de comunicação RMI. Tempo em minutos

Versão utilizando o *framework* proposto			
Simulação	Cenário 01	Cenário 02	Cenário 03
01	118,00	33,71	19,96
02	118,21	59,82	20,66
03	118,41	56,64	19,90
04	118,01	56,44	20,32
05	119,81	50,52	20,64
06	117,60	48,61	20,01
07	117,18	49,02	20,31
08	118,23	49,34	20,47
09	119,04	48,93	19,96
10	116,89	48,74	19,81
Média	118,14	50,18	20,25
Diferença	-7,21%	+67,24%	+334,22%

Tabela 5.5: Resultados na medição de tempo do simulador modificado utilizando o *EasyP* e o protocolo de comunicação RMI.

$H0_1$	$H0_2$	$H0_3$
VERDADEIRO	VERDADEIRO	FALSO

consideração que o sistema de comunicação a ser utilizado já está desenvolvido. Os tempos coletados em cada uma das execuções do simulador podem ser visualizados nas Tabelas 5.8, 5.9 e 5.10.

Também foi realizado o *teste-t* para verificar se a diferença entre o tempo médio das amostras coletadas da utilização do *EasyP* utilizando diversos protocolos de comunicação são significativos. Esta comparação procurou validar a hipótese de que não existe diferença entre a utilização de um outro protocolo de comunicação e a utilização do protocolo Java RMI. O resultado do teste pode ser visto na Tabela 5.11.

Como resultado do teste, foi verificado que não existiu diferença significativa em alterar a tecnologia de comunicação entre Java RMI, Java Sockets, CORBA e JavaSpace. Podendo então ser uma abordagem útil para poder utilizar o *EasyP* em diversos ambientes sem existir a necessidade de grandes configurações.

5.2 Conclusões

Neste capítulo foi realizada uma análise do custo de utilizar o *EasyP* como ferramenta de distribuição de componentes, para que estes sejam processados de forma distribuída. Foi verificado que não existe um custo muito grande para passar

Tabela 5.6: Valor da distribuição $|t_0|$ para o teste da hipótese nula.

$H0_1$	$H0_2$	$H0_3$
0,309	1,803	433,646

Tabela 5.7: Intervalo de confidência para a hipótese $H0_3$

$H0_3$
-370,61; -370,08

Tabela 5.8: Amostras de tempo do simulador modificado utilizando o *framework* proposto. Tempos coletados utilizando o 1º ambiente de simulação. Tempo em minutos

Simulação	RMI	Corba	Socket	JavaSpace
01	118,00	118,34	117,99	117,77
02	118,21	118,18	117,56	118,08
03	118,41	118,75	117,29	118,25
04	118,01	120,16	117,77	118,50
05	119,81	118,32	119,58	118,12
06	117,60	118,44	118,90	117,80
07	117,18	122,68	118,10	118,02
08	118,23	122,50	117,98	117,75
09	119,04	121,05	118,46	118,04
10	116,89	122,70	117,80	117,12
Média	118,14	120,11	118,14	117,95
Diferença	-	+ 1,68%	0%	- 0,17%

a utilizar o *EasyP*, permitindo então um ganho de desempenho considerável, já que poderá utilizar diversas máquinas para realizar o processamento necessário, ao invés de apenas uma.

Quando realizada a comparação entre a utilização do *EasyP* e a proposta de sistema distribuído de Durães, era de se esperar que existisse uma perda de desempenho, o que foi constatado ao ser realizada a coleta de amostras de tempo de algumas simulações, resultando em um *overhead* de cerca de 20 minutos em comparação com a versão de Durães. Ao longo das análises realizadas pode ser visto que este *overhead* permaneceu constante, porém ao ser utilizado um teste estatístico, foi verificado que em casos de que o tempo total da simulação é cerca de 2 vezes o tempo deste *overhead*, a utilização do *EasyP* não acarreta em uma diferença significativa. No entanto, quando passa-se a utilizar um ambiente com muitas máquinas, o tempo total irá continuar decaindo até chegar ao ponto do tempo total de processamento ser próximo do *overhead* inserido pela utilização do *EasyP*. Neste caso a perda de desempenho já passa a ser significativa, como foi demonstrado no experimento, no qual foi encontrado uma perda comparativa com a versão de Durães, de cerca de 334,22%.

Além disso, a troca de protocolo de comunicação não gera dificuldades e como esperado gera uma pequena diferença no desempenho ao utilizar a comunicação A ou a comunicação B. Porém, esta diferença não é estatisticamente significativa, podendo então utilizar qualquer uma das quatro tecnologias testadas no experimento, resultando em um desempenho equiparável.

Tabela 5.9: Amostras de tempo do simulador modificado utilizando o *framework* proposto. Tempos coletados utilizando o 2º ambiente de simulação. Tempo em minutos

Simulação	RMI	Corba	Socket	JavaSpace
01	56,76	56,54	49,69	52,66
02	59,82	51,77	50,13	42,38
03	56,64	48,34	38,29	49,72
04	56,44	51,54	46,66	47,50
05	50,52	48,94	43,49	47,84
06	48,61	51,69	49,38	53,71
07	49,02	50,03	47,86	44,19
08	49,34	48,33	44,86	48,11
09	48,93	52,42	43,26	48,93
10	48,74	49,09	47,98	52,73
Média	50,18	50,87	46,16	48,78
Diferença	-	+ 1,37%	- 8,01%	- 2.79%

Tabela 5.10: Amostras de tempo do simulador modificado utilizando o *framework* proposto. Tempos coletados utilizando o 3º ambiente de simulação. Tempo em minutos

Simulação	RMI	Corba	Socket	JavaSpace
01	19,96	20,32	20,14	19,97
02	20,66	20,55	19,60	20,12
03	19,90	20,34	19,84	20,49
04	20,32	20,99	20,01	19,80
05	20,64	21,70	19,82	19,86
06	20,01	20,47	19,50	19,96
07	20,31	20,79	19,82	19,68
08	20,47	20,49	19,70	19,85
09	19,96	21,17	19,64	20,10
10	19,82	20,82	19,97	20,97
Média	20,20	20,76	19,80	20,08
Diferença	-	+ 2,77%	- 1,98%	- 0,59%

Tabela 5.11: Resultado da avaliação da comparação da utilização do *EasyP* com Java RMI e outros protocolos de comunicação.

Ambiente	Corba	Socket	JavaSpace
01	VERDADEIRO	VERDADEIRO	VERDADEIRO
02	VERDADEIRO	VERDADEIRO	VERDADEIRO
03	VERDADEIRO	VERDADEIRO	VERDADEIRO

Capítulo 6

Conclusão

> Nós não estamos nos retirando...
> Estamos avançando em outra
> direção.
>
> Douglas McArthur

> Não lute uma batalha se você não
> puder tirar proveito de sua vitória.
>
> Erwin Rommel

Durante este capítulo, serão apresentadas as principais contribuições realizadas por esta dissertação e alguns possíveis aprimoramentos no *EasyP*, permitindo uma evolução deste *framework* proposto. Além disto, será apresentada uma discussão final sobre o tema abordado ao longo deste estudo.

6.1 Contribuições

Durante o estudo realizado ao longo deste mestrado, foram notadas algumas contribuições relevantes na escrita desta dissertação, estas estão listadas a seguir:

- Um estudo comparativo entre os ambientes e algumas tecnologias de comunicação utilizadas em sistemas distribuídos e paralelos. Mostrando suas características, benefícios e falhas quando comparados. Este estudo revelou algumas das dificuldades enfrentadas ao desenvolver um aplicativo que possa ser processado nestes ambientes. Entre estas dificuldades, é possível citar a falta de flexibilidade ao desenvolver um aplicativo para o ambiente A e posteriormente utilizar este mesmo aplicativo no ambiente B.

- Um levantamento bibliográfico sobre os trabalhos realizados no campo de sistemas distribuídos, principalmente em ferramentas com o intuito de prover um suporte ao usuário no desenvolvimento de seus aplicativos distribuídos.

- O desenvolvimento de uma ferramenta, o *EasyP*, capaz de auxiliar o desenvolvimento de aplicativos capazes de serem processados em ambientes

computacionais com mais de uma única unidade de processamento. Esta ferramenta foi desenvolvida com o intuito de reduzir o custo do desenvolvimento de aplicações distribuídas. Desta forma, com um menor esforço investido, os desenvolvedores poderiam possuir versões de seus aplicativos para executarem em ambientes distribuídos.

- Um estudo estatístico comparativo entre a utilização do *EasyP* e uma ferramenta desenvolvida em um nível mais baixo de abstração, que é o caso da ferramenta proposta por Durães [11]. Com este estudo estatístico, foi possível perceber que a perda de desempenho por utilizar o *EasyP* não é significativa, em casos que o tempo total não de processamento não fica próximo do *overhead* inserido pelo *EasyP*.

- Um estudo estatístico comparativo entra quatro tecnologias de comunicação distintas, permitindo verificar a diferença de desempenho na utilização de cada uma. Com esta análise estatística, foi possível verificar que não houve uma diferença significativa entre as tecnologias de comunicação utilizadas em três ambientes distintos.

6.2 Trabalhos Futuros

Durante o estudo realizado para o desenvolvimento desta proposta de mestrado, foram identificados diversos pontos de desenvolvimento que permitiriam um melhor aproveitamento do *framework* proposto. Porém, nem todas estas características foram desenvolvidas até o momento. Nesta seção estão listadas algumas destas características que poderão ser acrescentadas ao longo do tempo em novas versões desta ferramenta.

A primeira característica observada foi a inserção de técnicas de tolerância a falhas. No momento já existe uma única técnica inserida na ferramenta que é o monitoramento dos pontos remotos com o intuito de verificar se este ponto ainda se encontra em atividade. Caso ocorra alguma falha, o componente que havia sido enviado será recuperado e re-enviado para um outro ponto de processamento para ser processado novamente. Desta forma, não ocorrem perdas de informações caso uma das máquinas seja desligada durante o funcionamento do *framework*. A idéia seria prover mais serviços de tolerância a erros, permitindo que o sistema que utilize o *EasyP* possa se tornar mais estável.

Uma outra característica é o fato de que existem diversos aplicativos que não são desenvolvidos em Java, além de existirem diversos aplicativos legados nos quais demoram muito tempo para o seu processamento completo, mas que não existe nenhum incentivo para uma nova versão destes aplicativos voltados para sistemas distribuídos. Estes aplicativos podem já estar organizados em componentes de software, podendo assim serem distribuídos usando a ferramenta, caso não ocorresse essa incompatibilidade do código legado e Java. Para isto, existe o intuito de se desenvolver uma nova interface de componentização capaz de interagir com este código legado. Neste caso deverão ser utilizadas tecnologias Java de código Nativo, como o JNI (*Java Native Interface*) [23]. Com isto, teríamos mais abrangência de uso do *framework* proposto, permitindo que aplicações antigas, ou até mesmo que as mais novas desenvolvidas em linguagens mais poderosas como C, possam usufruir do framework aqui proposto.

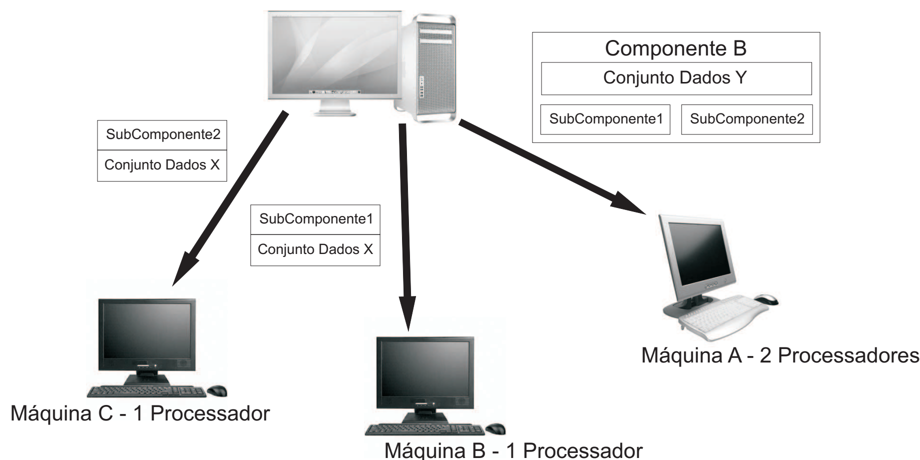

Figura 6.1: A proposta atual comparada com a proposta de modificação futura. Ao invés de enviar componentes e um mesmo conjunto de dados para máquinas diferentes, enviar um único componente e apenas uma única vez o conjunto de dados, evitando o re-envio dos dados pela rede.

Outra característica foi observada ao analisar algumas simulações de algoritmos na área de Inteligência Artificial e também o trabalho proposto por Mehta [38]. Nestas simulações, existem diversos casos onde temos vários conjuntos de componentes que trabalham sobre um conjunto de dados único para cada um dos conjuntos de componentes. Por exemplo, existiria um conjunto de componentes A que processa um determinado conjunto de dados X, onde cada um dos componentes utiliza sempre este mesmos dados de X acrescentando apenas um pequena informação a mais; e teríamos um outro conjunto de componentes B que processa o conjunto de dados Y sob mesmas circunstâncias. Desta forma, seria possível enviar um conjunto de componentes para uma máquina com mais de uma única unidade de processamento, reduzindo o custo de envio do conjunto de dados, já que seria enviado apenas uma vez, ao contrário do envio de cada um destes sub-componentes e seu respectivo conjunto de dados para máquinas separadas, duplicando o custo do envio pela rede de comunicação. Assim, a proposta de modificação futura é o desenvolvimento de uma interface de conjunto de componentes, onde teríamos um componente macro que representaria a execução de cada um destes sub-componentes.

A idéia da proposta desta modificação pode ser visualizada na Figura 6.1, onde inicialmente existe um *Componente 1* que processa utilizando um *Conjunto de Dados X* e um segundo *Componente 2* que necessita do mesmo *Conjunto de Dados X*. Ao enviarmos estes componentes *1* e *2* para máquinas separadas enviamos um mesmo bloco de dados para máquinas diferentes ocorrendo um re-envio de dados sem necessidade. Após a modificação proposta, poderá ser enviado um conjunto de *Componentes B* que contém o *Conjunto de Dados X* e os dois componentes *1* e *2* para serem processados em uma única máquina com mais de uma unidade de processamento, evitando o re-envio dos dados.

Uma última modificação é com relação ao estilo de comunicação entre as máquinas. Atualmente, o *framework* realiza uma comunicação centralizada, como pode ser visualizada na Figura 6.2, onde uma máquina escrava para se comunicar com outra máquina escrava é necessário passar sua comunicação pela máquina

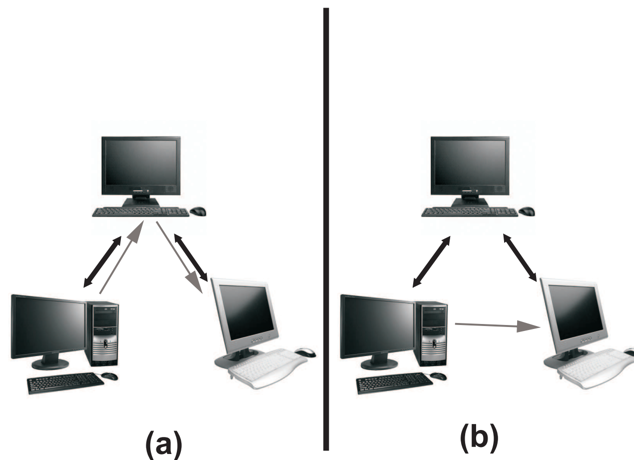

Figura 6.2: Proposta atual de comunicação centralizada do *EasyP* em (a) e a nova proposta (b), onde um escravo pode se comunicar diretamente com o segundo escravo.

mestre. A modificação proposta é a realização de uma comunicação não centralizada, onde seria possível uma máquina escrava se comunicar diretamente com outra escrava sem o intermédio da máquina mestre, reduzindo, assim, o custo de comunicação e simplificando esta troca de informações entre as máquinas escravas.

Além dessas características, mais testes são necessários, como por exemplo em um conjunto maior de máquinas, além de utilizar outros aplicativos para realizar a comparação. Com estes novos testes poderão ser encontradas novas falhas e melhorias na proposta do *EasyP*.

6.3 Comentários Finais

Ao longo desta dissertação foi realizada a apresentação do *EasyP*, um *framework* para facilitar o desenvolvimento de aplicações distribuídas. A motivação desta proposta foi a existência de diversos aplicativos complexos que necessitam de um grande poder computacional e seu resultado pode exigir que seja encontrado dentro de um tempo determinado, caso contrário poderá perder a validade. Uma alternativa para este problema é a utilização de máquinas computacionais mais poderosas, mas esta é uma alternativa muitas vezes inviáveis, devido ao alto custo para adquirir estas máquinas.

Uma outra solução é a utilização de um ambiente com diversas máquinas mais simples, resultando em um ambiente de processamento unificado. No entanto, o desenvolvimento de aplicativos para usufruirem deste ambiente é uma tarefa complexa. Desta forma, foi proposto o *framework EasyP*, uma ferramenta que procura justamente facilitar o desenvolvimento de aplicações para usufruirem de todo o potencial existente em um ambiente computacional formado por mais de uma máquina, em especial as máquinas com processadores *multi-core*, uma realidade atualmente.

Um dos principais objetivos do *EasyP* é permitir que o aplicativo possa ser utilizado em qualquer ambiente. Para isto, foi necessário que este *framework* se tornasse abstrato o suficiente para não sofrer influência da plataforma na qual está sendo processado. Além disto, outras abstrações foram necessárias, como

a abstração da comunicação entre as máquinas, para que se tornasse possível a utilização de qualquer tipo de comunicação, tornando-se completamente desacoplado da máquina.

Devido a esta abstração do *hardware*, o desempenho do *EasyP* seria afetado, passando a possuir um desempenho pior quando comparado à ferramentas desenvolvidas em um nível de abstração menor. Esta expectativa foi confirmada quando foi realizado um teste estatístico comparando os tempos gastos no processamento do simulador proposto por Durães [11] e uma versão modificada utilizando o *EasyP*. No entanto, como resultado do teste estatístico foi comprovado que no caso do tempo total ser superior ao *overhead* inserido pelo *EasyP*, esta diferença de desempenho não é significativa. Porém, no caso do tempo total ser próximo ao tempo de *overhead* inserido pelo *EasyP*, a diferença passa a ser significativa.

Além disto, também foi mostrado o custo de utilização do *EasyP*. As pequenas alterações necessárias foram exemplificadas, demonstrando o baixo esforço necessário para utilizar o *framework*. Em contrapartida, a utilização do *EasyP* fornece um ganho de performance quando comparada ao processamento de um aplicativo puramente sequencial.

Desta forma, para o desenvolvimento de um aplicativo distribuído que necessita melhorar seu tempo de resposta, porém não ao máximo, a utilização do *EasyP* permitirá este desenvolvimento ser muito menos complexo. No entanto, quando necessário realizar esta melhoria ao máximo, o *EasyP* não é mais aconselhável, passando a ser indicado o desenvolvimento do sistema de distribuição específico para o problema em questão, e portanto otimizado.

Referências

[1] Advanced Micro Devices AMD. Multi-core processors the next evolution in computing. Amd multi-core technology whitepaper, AMD, 2005.

[2] Advanced Micro Devices AMD. http://www.amd.com/, agosto 2008.

[3] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, and Fred Long. Technical concepts of component-based software engineering, volume 2, August 2009.

[4] Bob Bates and Robert A. Bates. *Game Design, 2nd Edition.* Course Technology Press, Boston, MA, United States, 2004.

[5] Cluster Beowulf. http://www.beowulf.org/, agosto 2008.

[6] Mary Brandel. Dual-core debate. Technical report, Network World, December 2004.

[7] A. Brown and K. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, 1998.

[8] Sarita Mazzini Bruschi. *ASDA - Um Ambiente de Simulação Distribuída Automático.* PhD thesis, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, Brasil, 2002.

[9] Sarita Mazzini Bruschi, Regina Helena Carlucci Santana, Marcos José Santana, and Thais Souza Aiza. An automatic distributed simulation environment. In *WSC '04: Proceedings of the 36th conference on Winter simulation*, pages 378–385. Winter Simulation Conference, 2004.

[10] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 246–261, 2002.

[11] Gilvan Martins Durães, André Castelo Branco Soares, Sérgio Castelo Branco Soares, and William Ferreira Giozza. Simrwa-d: Uma abordagem distribuída para simulação de redes Ópticas transparentes. In *5th International Information and Telecommunication Technologies Symposium (I2TS)*, volume 1, pages 1 – 11, 2006.

[12] Albert Einstein. *Relativity: the Special and General Theory.* Holt and Company (English edition, 1920), 1916. Republished by Dover, 2001.

[13] Mohamed E. Fayad and Douglas C. Schmidt. Object-oriented application frameworks (special issue introduction). *Communications of the ACM*, 40(10):39–42, October 1997.

[14] Michael Flynn. Some computer organizations and their effectiveness. *IEEETC:JOURNAL*, 21(9):948–960, 1972.

[15] CERN European Organization for Nuclear Research. http://gridcafe.web.cern.ch/gridcafe/, agosto 2008.

[16] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *SOSP*, pages 78–91, 1997.

[17] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces(TM) Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[18] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley Interscience, New York, January 2000.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[20] G. A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Menchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. MIT press, 1994.

[21] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[22] Patrick P Gelsinger, Paolo A. Gargini, Gerhard H. Parker, and Albert Y. C. Yu. Microprocessors circa 2000. In *IEEE Spectrum*, volume 26, pages 43–47. 1989.

[23] Rob Gordon. *Essential JNI: Java Native Interface*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[24] Chona S. Guiang, Kent F. Milfeld, Avijit Purkayastha, and John R. Boisseau. Memory performance of dual-processor nodes: comparison of intel xeon and AMD opteron memory subsystem architectures, 2003.

[25] Elliote Rusty Harold. *Java Network Programming*. O'Reilly, 1997.

[26] Christopher J. Hemingway and Tom G. Gough. The value of information systems teaching and research in the knowledge society. *Informing Science The International Journal of an Emerging Transdiscipline*, 3(4):167–184, 2000.

[27] D. Austin Henderson, Jr. and Theodore H. Myer. Issues in message technology. In *SIGCOMM '77: Proceedings of the fifth symposium on Data communications*, pages 6.1–6.9, New York, NY, USA, 1977. ACM.

[28] Intel. Intel multi-core processor architecture development backgrounder. Intel whitepaper, Intel, May 2005.

[29] Intel. Introducing the 45nm next-generation intel core microarchitecture. Intel whitepaper, Intel, November 2007.

[30] Bart Jacob, Michael Brown, Kentaro Fukui, and Nihar Trivedi. Introduction to grid computing. Redbook, IBM, December 2005.

[31] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse. Architecture, Process and Organization for Business Success.* Addison-Wesley, 1997.

[32] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(5):22–35, 1988.

[33] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 36–43, New York, NY, USA, 2004. ACM.

[34] Leonidas Kontothanassis, Robert Stets, Galen Hunt, Umit Rencuzogullari, Gautam Altekar, Sandhya Dwarkadas, and Michael L. Scott. Shared memory computing on clusters with symmetric multiprocessors and system area networks. *ACM Trans. Comput. Syst.*, 23(3):301–335, 2005.

[35] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, New York, NY, USA, 2008.

[36] James F. Kurose and Keith W. Ross. *Redes de Computadores e a Internet: Uma abordagem top-down.* Addison Wesley, São Paulo, trad. 3 ed. edition, 2006.

[37] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6(2):254–280, 1984.

[38] Neeraj Mehta, Yogesh Kanitkar, Konstantin Läufer, and George K. Thiruvathukal. A model-driven approach to job/task composition in cluster computing. In *IPDPS*, pages 1–8. IEEE, 2007.

[39] Java Sun Microsystems. http://java.sun.com/, Agosto 2009.

[40] Vilson Luiz Dalle Mole, Jefferson Gustavo Martins, and Humberto Bertoluzzi. Aplicações distribuídas baseadas em trabalho cooperativo com javaspace. In *IV Congresso Brasileiro de Computação*, volume 1, 2004.

[41] Richard Monson-Haefel and Bill Burke. *Enterprise JavaBeans.* O'Reilly & Associates, Inc., pub-ORA:adr, fifth edition, 2006.

[42] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[43] Qian Ning and T. Sejnowski. Predicting the secondary structure of globular proteins using neural network models. *J. Mol. Biol.*, 202:865–884., 1988.

[44] The Object Management Group (OMG). http://www.omg.org/, setembro 2009.

[45] Pawel Plaszczak and Richard Wellner. *Grid computing: the savvy manager's guide.* Elsevier/Morgan Kaufmann, pub-ELSEVIER-MORGAN-KAUFMANN:adr, 2005.

[46] Fernando Rocha, Sérgio Soares, André Soares, and Ricardo Lima. An adaptable framework for distributed and parallel applications. In *POOSC '09: Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, pages 1–7, Genova, Italy, 2009. ACM.

[47] Andrew Rollings and Dave Morris. *Game Architecture and Design: A New Edition.* New Riders Games, 2003.

[48] Ounsa Roudiès and Mounia Fredj. A reuse based approach for requirements engineering. In *AICCSA '01: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, page 448, Washington, DC, USA, 2001. IEEE Computer Society.

[49] Johannes Sametinger. *Software Engineering with Reusable Components.* Springer-Verlag, New York, NY, 1997.

[50] Humphrey Sheil. Distributed scientific computing in java: observations and recommendations. In James F. Power and John Waldron, editors, *Proceedings of the 2nd International Symposium on Principles and Practice of Programming in Java PPPJ*, volume 42 of *ACM International Conference Proceeding Series*, pages 219–222. ACM, June 16-18 2003.

[51] W. Stallings. *Computer Organization and Architecture.* Prentice Hall, London, 1996.

[52] 2004 Sun Microsystems Java Remote Method Invocation Specification 1.5.0. http://java.sun.com/j2se/1.5.0/docs/guide/rmi/, MARCH 2009.

[53] Clemens A. Szyperski. *Component Software — Beyond Object-Oriented Programming.* Addison Wesley, second edition edition, 2002.

[54] Andrew S. Tanenbaum. *Sistemas Operacionais Modernos.* Prentice-Hall, 2003.

[55] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice.* Wiley, 2009.

[56] Y. M. Teo, Y. K. Ng, and B. S. S. Onggo. Conservative simulation using distributed-shared memory. In *Proc. of the 16th Workshop on Parallel and Distributed Simulation*, pages 3–10, 2002.

[57] Yong Meng Teo, Seng Chuan Tay, and Siew Theng Kong. Structured parallel simulation modeling and programming. In *Annual Simulation Symposium*, pages 135–142. IEEE Computer Society, 1998.

[58] Rob V. van Nieuwpoort, Jason Maassen, Rutger Hofman, Thilo Kielmann, and Henri E. Bal. Ibis: an efficient java-based grid programming environment. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande (JGI-02)*, pages 18–27, New York, November 3–5 2002. ACM Press.

[59] The OMG's CORBA Website. http://www.corba.org, setembro 2009.

[60] J. E. White. RFC 707: High-level framework for network-based resource sharing, December 1975. Status: UNKNOWN. Not online.

[61] Jeremy B. Williams and Michael Goldberg. The evolution of e-learning. *ASCILITE 2005*, December 2005.

[62] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[63] Zhonghua Yang and Keith Duddy. CORBA: A platform for distributed object computing. *ACM Operating Systems Review*, 30(2):4–31, April 1996.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo