Configuração Automática de Sistemas na Plataforma Enterprise JavaBeans

Herbert Yutaka Watanabe

Dissertação apresentada ao

Instituto de Matemática e Estatística da

Universidade de São Paulo

como parte dos requisitos

para a obtenção de grau de

Mestre em Ciência da Computação.

Orientador : **Prof. Dr. Fábio Kon**

- São Paulo, SP – Fevereiro de 2003 -

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Com a popularização do computador, a necessidade pelos mais diversos tipos de softwares se multiplicou fazendo com que o processo de desenvolvimento de sistemas e aplicações se tornasse necessariamente mais ágil. A tecnologia de componentes trouxe essa agilidade necessária ao possibilitar a reutilização de código, de uma forma organizada e simples, que favorece a construção de sistemas complexos simplesmente encaixando-se ‘componentes de prateleira’ uns aos outros.

Tecnologias de programação distribuídas e arquiteturas de servidores baseados em componentes surgiram para auxiliar ainda mais nesse desenvolvimento. Contudo não solucionaram todos os problemas, ainda continuaram existindo dificuldades de desenvolvimento de softwares confiáveis, eficientes e dinamicamente configuráveis baseados em componentes. Alguns trabalhos surgiram mostrando como a representação das dependências entre os componentes auxilia na construção de softwares mais confiáveis, eficientes e dinamicamente configuráveis.

Nessa dissertação, mostramos esses trabalhos e suas características. Também transportamos o arcabouço (*framework*) de um desses trabalhos para plataforma *Enterprise JavaBeans* (EJB) mostrando sua viabilidade também nessa plataforma.

Outra contribuição dessa dissertação é a extensão do arcabouço de forma a possibilitar a utilização de operadores booleanos na descrição das dependências e a utilização dos serviços de nomes e diretórios na resolução das dependências.

Abstract

When the computer became popular, the necessity for all kinds of softwares increased making the development process of systems and applications necessarily more agile. The component technology brought this agility making the code reuse possible in an organized and simple form. That allows the construction of complex systems by plugging to each others ‘off-the-shelf’ components.

Distributed programming technology and servers architectures based on components were created to assist and help even more in this development. However, the problems had not been solved at all, difficulties to develop trustworthy, efficient and dynamic configured systems based on components still continues to exist. Some research had appeared showing how components dependency representation helps on the development of trustworthy, efficient and dynamic configured systems.

In this dissertation we show these researches and its characteristics. We also exchange the platform of a framework, proposed at one of these research, to the Enterprise JavaBeans platform showing its viability also in this platform.

Another contribution of this dissertation is the framework extension that makes possible the use of boolean operators in the description of dependencies and the uses of naming and directory services in the resolution of dependencies.

Capítulo 1

Introdução

No início da era da informática, devido ao alto custo de hardware e relativo baixo custo da mão de obra, os softwares eram projetados visando otimizar a utilização do computador não se importando tanto na construção de interfaces amigáveis para os usuários dos sistemas e das aplicações. As empresas que desejavam adquirir esses computadores deveriam investir quantias vultosas e, portanto, possuir um bom motivo para essa aquisição. Uma vez adquiridos esses computadores, as pessoas que tinham acesso a eles tinham dois perfis: ou eram desenvolvedores e analistas, usuários altamente técnicos que desenvolviam programas para atender às necessidades da empresa ou eram usuários comuns, que tinham suas atividades moldadas de forma a aumentar a eficiência e utilização dos computadores a fim de atingir o objetivo para o qual estes foram adquiridos.

Com a redução do custo de hardware, a popularização do computador e o relativo crescimento do custo da mão-de-obra, todo esse quadro foi alterado. As atividades dos usuários já não são tão moldadas pelas limitações do computador, agora buscam por software dos mais diversos tipos que aumentem sua produtividade [1]. Isso gera uma alta demanda por novos softwares com constantes especializações; o desenvolvimento de uma boa interface com o usuário já não é mais deixado em segundo plano. Agora, o foco está na agilidade de desenvolvimento visando atender os mais diversos nichos de mercado no menor espaço de tempo e com o menor custo possível.

Ao mesmo tempo, existe uma grande variedade de arquiteturas de hardware com uma forte tendência de expansão com as novas tecnologias de Computação Móvel e Espaços Ativos (PDAs, telefones celulares, sensores, etc) que irão coexistir com o computador pessoal, estações de trabalho, servidores e supercomputadores.

Segundo Desmond F. D’Souza e Alan Cameron Wills, a chave para se fazer uma grande variedade de softwares em um curto espaço de tempo é fazer um pedaço de código ser utilizado em vários produtos [2]. Essa reutilização não significa copiar e colar código, se isso for feito, incontáveis locais de alteração serão gerados fazendo com que, rapidamente a manutenção desse código vire um pesadelo.

Uma estratégia mais eficiente é fazer pequenos projetos genéricos que são construídos para serem utilizados em vários produtos. Esses pedaços reutilizáveis podem ser código, modelos, padrões, especificações e até um planejamento de projeto.
Duas regras chaves para construção dessas partes reutilizáveis são:

1) Essas partes não devem ser modificadas por quem as usa, somente especializadas.
2) Essas partes têm de formar um conjunto coerente.

Essas partes reutilizáveis que podem ser adaptadas mas não modificadas são chamadas componentes [2].

A tecnologia de componentes é uma conseqüência natural da maturidade do processo de desenvolvimento e a sua não utilização significa, na maioria dos casos, ‘reinventar’ soluções, sendo assim muito difícil escapar da utilização de componentes [3].

Atualmente uma característica freqüentemente encontrada nos sistemas é a distribuição, tanto de dados como do processamento. Os sistemas necessitam consultar ou realizar operações remotas e, a fim de se facilitar a construção de aplicações distribuídas escaláveis, surgiram padrões de middleware como o *Commom Object Request Broker Architecture* (CORBA), *Microsoft Distributed Component Object Model* (DCOM) e *Java Remote Method Invocation* (RMI) que não serão objeto de estudo neste trabalho, mas resultam em implementações *ad-hoc* devido à falta de padronização na implementação dos componentes e na comunicação entre eles. Com isso, surgiram definições de arquiteturas de servidores baseados em componentes como CORBA *Component Model* (CCM) [29], *Enterprise JavaBeans* (EJB) [18] e *Microsoft Transaction Server* (MTS) [30]. Em cada uma dessas arquiteturas, o desenvolvedor não precisa se preocupar com o desenvolvimento de um servidor por completo, somente conecta os diversos componentes existentes, que podem ser por ele desenvolvidos ou não. Toda a infra-estrutura como gerenciamento de *threads*, transações e segurança são deixados a cargo do servidor de aplicação que é definido pela arquitetura [14].

Contudo, isso ainda não solucionou um problema de extrema importância e relevância: as interdependências entre os componentes. Essas dependências têm sido estudadas e tratadas de diversas formas a fim de se obter softwares de melhor administração, organização e distribuição [9]; capazes de se configurar automaticamente [6] e de se reconfigurar dinamicamente [6] [8] [13] e visando acima de tudo, uma melhor reutilização desses componentes de forma a obter softwares mais confiáveis em um menor tempo. Tudo isso é cada vez mais necessário à medida que os softwares se tornam mais complexos, de maior escala e possuindo maior interação com o ambiente como, por exemplo, os projetos de computação ubíqua (espaços ativos) que são espaços físicos - como escritórios, salas de aula, casas, hospitais – onde estão acoplados periféricos computacionais integrados com o ambiente. O objetivo desses periféricos é prover informações para os usuários do ambiente, os auxiliando a executar ações mais facilmente ou ações que antes não eram possíveis [4].

1.1. Contribuição da dissertação

A fim de resolver o problema descrito, nesta tese implementamos um arcabouço (*framework)* que:

- possibilita a representação das dependências de seus componentes e
- possibilita que componentes se configurem automaticamente através da manipulação dessas dependências.

Essa configuração automática, ao simplificar a configuração de aplicativos, vem de encontro com o aumento do custo de mão-de-obra. Uma vez que a complexidade de instalação e manutenção desses aplicativos é reduzida, será possível diminuir o tempo que uma pessoa especializada fica alocada para esse serviço.

Nesta tese mostramos também a viabilidade do arcabouço apresentado por Kon no Sistema de Configuração Automática do 2K (ver seção 2.4) sobre uma plataforma diferente da utilizada por ele, utilizamos a arquitetura de componentes EJB. Além de permitir a representação explícita das dependências entre os componentes EJB, essa tese trás extensões como a inclusão de operadores booleanos (OR e AND) na descrição dessas dependências e a inclusão dos serviços de nomes e de diretórios na resolução das dependências.

1.2. Organização da dissertação

Essa dissertação está organizada da seguinte forma: no Capítulo 2 apresentamos os trabalhos relacionados à configuração automática dos sistemas e aplicativos. No Capítulo 3 apresentamos uma visão geral da arquitetura do sistema de configuração automática proposta, mostrando cada parte do sistema e definindo as terminologias utilizadas nessa dissertação. No Capítulo 4 apresentamos as tecnologias utilizadas na implementação do sistema bem como a implementação propriamente dita. No Capítulo 5 temos uma avaliação de desempenho do sistema implementado e uma comparação com outro sistema. Apresentamos também a plataforma utilizada para esses experimentos e como os dados foram coletados. No Capítulo 6 apresentamos possíveis trabalhos futuros e no Capítulo 7 apresentamos nossas conclusões.

Capítulo 2

Trabalhos Relacionados

Atualmente, os usuários de computadores estão se deparando com diferentes tipos de dispositivos que vão desde PDAs e Desktops até poderosas estações de trabalho e servidores. Cada um desses dispositivos pode possuir diferentes sistemas operacionais cada um possuindo seus próprios softwares com diferentes interfaces de configurações. Assim, os usuários têm de lidar com uma grande quantidade de manuais para configurar corretamente cada um desses dispositivos.

Como vemos, a necessidade da configuração automática dos sistemas é uma necessidade cada vez maior nos dias atuais. A seguir apresentamos os principais trabalhos que encontramos na literatura sobre configuração automática.

2.1. Advanced Packaging Tool (APT)

O sistema operacional Linux Debian [15] possui o dpks, um utilitário de manutenção que gerencia as dependências entre os executáveis, bibliotecas, documentações e arquivos de dados associados aos pacotes de software da Debian. Ele mantém um banco de dados de dependências que possibilita ao usuário instalar, atualizar e apagar pacotes sem comprometer a consistência do sistema. Até mesmo bibliotecas compartilhadas por diversos pacotes são gerenciadas consistentemente.

Contudo, ainda existia a dificuldade para uma rápida e prática instalação dos pacotes, que gerenciasse automaticamente as dependências e cuidasse da configuração quando se fizesse uma atualização. Para isso foi criado o *Advanced Packaging Tool* (APT).

O APT possui o seu próprio banco de dados para se saber quais pacotes estão instalados, quais não estão e quais estão disponíveis para instalação. O comando apt-get utiliza o dpks para verificar as dependências de cada pacote a fim de instalar automaticamente todas as dependências de cada pacote requisitado.

2.2. Jini [16]

O Jini provê um mecanismo simples que facilita a configuração de serviços Java e possibilita que esses serviços sejam *plug-and-play*. Ele não gerencia as dependências das aplicações nem dos componentes.

A Tecnologia Jini especifica um modo de como os clientes e serviços podem se encontrar e trabalhar juntos para realizar uma tarefa. Os provedores de serviços fornecem para os clientes objetos baseados na tecnologia Java. Estes objetos disponibilizaram aos clientes o acesso ao serviço. A interação via rede entre o cliente e o provedor do serviço pode ser feito com qualquer tipo de tecnologia (CORBA, RMI, SOAP) uma vez que os clientes somente vêem o objeto Java fornecido.

Quando um serviço entra na rede de serviços e/ou dispositivos do Jini, ele publica o objeto Java que implementa a *Application Programming Interface* (API) do serviço. Quando um cliente pega o objeto publicado pelo serviço, ele baixa tudo o que precisa para se comunicar com o serviço, “aprendendo” assim como se comunicar com o serviço através da API.

2.3. Java Network Launcher Protocol (JNLP)

O *Java Network Launcher Protocol* (JNLP) é um protocolo de entrega baseado na Web e um ambiente para aplicações produzido em Java para a Web. Aplicações que implementam essa especificação são chamadas de clientes JNLP.

Esses clientes são capazes de:

- Fazer a transferência do código, guardar esse código localmente e abrir uma aplicação 100% Java que está disponível em um servidor Web.
- Prover um ambiente padrão para a execução dessa aplicação.
- Fazer a transferência do código, instalação e gerenciamento dos ambientes de execução Java (JRE) e outros pacotes.

No nosso trabalho, os códigos também podem ser transferidos da mesma forma que o JNLP, contudo, ao contrário do cliente JNLP que roda num *sandbox*, os componentes do nosso trabalho rodam num servidor EJB.

2.4. Serviço de Configuração Automática do 2K

O trabalho de Kon [6] é o mais extenso que encontramos sobre configuração automática e reconfiguração dinâmica de sistemas baseados em componentes. O seu trabalho se baseia no modelo *What You Need Is What You Get* (WYNIWYG), ou seja, o sistema se configura automaticamente e carrega um conjunto mínimo de componentes necessários para a execução de uma aplicação. Esse modelo é muito diferente dos modelos atuais dos sistemas operacionais, middleware e aplicações onde diversos módulos são disponibilizados na instalação padrão mesmo que nunca sejam utilizados.

O sistema de configuração automático do sistema operacional 2K [5] foi construído para testar o modelo WYNIWYG. O sistema baixa da rede, somente um pequeno conjunto de componentes necessários para dar inicio à aplicação. Quando um componente é carregado no sistema, as suas dependências são checadas e, todos os componentes e serviços necessários são disponibilizados. Isso é possível, pois na arquitetura apresentada por Kon cada componente tem especificadas todas as suas dependências de hardware e software.

A arquitetura apresentada por Kon é uma arquitetura de configuração automática de sistemas baseados em componentes distribuídos em C++ e CORBA [6]. Essa arquitetura se propõe a oferecer o suporte necessário para que sistemas e aplicações representem a sua organização estrutural interna de dependências entre os componentes e essa habilidade é chamada de "consciência arquitetural" (*architectural-awareness*). Essa representação de dependências entre os componentes é utilizada na implementação de componentes capazes de se configurar automaticamente e de se adaptar dinamicamente às mudanças do ambiente em que se encontram.

A arquitetura apresentada é formada por seis grandes blocos:

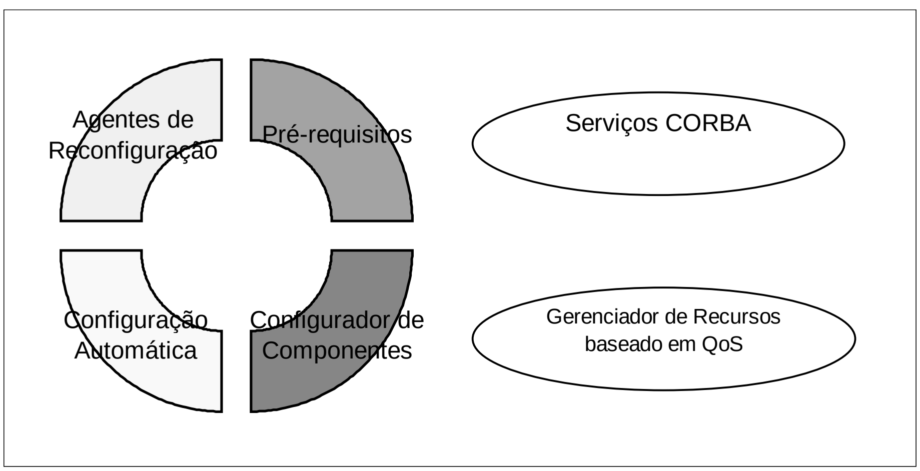


FIGURA 2.1 – ARQUITETURA DO SERVIÇO DE CONFIGURAÇÃO AUTOMÁTICO DO 2K

Os Pré-requisitos se referem às dependências estáticas dos componentes enquanto os Configuradores de Componentes se referem às dependências dinâmicas de tempo de execução do componente. A Configuração Automática é baseada nos pré-requisitos e constrói os Configuradores de Componentes. Os Agentes Móveis de Reconfiguração são utilizados para tornar o gerenciamento dos componentes mais escalável e mais eficiente.

Para se especificar os pré-requisitos é apresentado um formato de arquivo, o *Simple Prerequisite Description Format* (SPDF) [4], onde são especificadas as dependências estáticas de cada componente.

São considerados três diferentes tipos de pré-requisitos:

1) A natureza dos recursos de hardware necessários
2) A capacidade necessária de cada recurso de hardware
3) Outros softwares, ou seja, outros componentes

Para cada item é também possível se especificar se é um item necessário ou opcional para o correto funcionamento do componente.

Os requisitos de software são todos referentes a um repositório de componentes de onde são pegos os componentes necessários. Isso é feito durante o processamento do arquivo de dependências do componente. Se constatada a necessidade de um determinado componente, ele é procurado no repositório de componentes através da referência contida no arquivo de dependência, é realizada a transferência do seu código caso este já não esteja na máquina local e ele é instalado.

```
:hardware requirements
machine_type      SPARC
os_name           Solaris
os_version        2.7
min_ram           5MB
optimal_ram       40MB
cpu_speed         >300MHz
cpu_share         10%

:software requirements
FileSystem        CR:/sys/storage/DFS1.0 (optional)
TCPNetworking     CR:/sys/networking/BSD-sockets
WindowManager     CR:/sys/WinManagers/simpleWin
JVM               CR:/interp/Java/jvm1.2 (optional)
```

FIGURA 2.2 – EXEMPLO DE UM ARQUIVO SPDF.

Na figura 2.2 mostramos um exemplo de um arquivo de dependência de um *Web-Browser* no formato SPDF. Na primeira parte do arquivo temos as especificações dos pré-requisitos de hardware. Temos especificado que a aplicação foi compilada em uma máquina Sparc executando o Solaris 2.7, que ele precisa de no mínimo 5MB de memória RAM, mas que 40MB é o recomendado, e que ele necessita de 10% do tempo de processamento de uma CPU com no mínimo de 300MHz.

Na segunda parte, temos as especificações de software. Temos especificado que o Web-browser necessita de quatro componentes (ou serviços): um sistema de arquivos que é opcional, um serviço de rede TCP, um gerenciador de janelas, e uma máquina virtual Java que também é opcional.

A linha

```
FileSystem CR:/sys/storage/DFS1.0 (optional)
```

especifica que o componente que implementa o sistema de arquivos pode ser localizado no diretório `/sys/storage/DFS1.0` do repositório de componentes (CR). Esse pré-requisito é declarado opcional (*optional*), ou seja, se o serviço de configuração automática não puder carregar esse componente, ele simplesmente irá lançar um aviso e continuar a processar o arquivo.

Em tempo de execução, a representação das dependências é feita através de um objeto que é ligado a cada componente. Esse objeto é chamado de configurador de componentes. Ele é responsável por guardar informações em tempo de execução sobre dependências dinâmicas entre os componentes. Esse componente é utilizado para as reconfigurações dinâmicas. Ele permanece 'dormente' até que receba uma requisição para reconfiguração ou algum outro evento de notificação.

Sabendo que o gerenciamento da configuração de um software em grande escala é um dos grandes problemas nos sistemas atuais, Kon em sua arquitetura prevê ainda a reconfiguração dinâmica dos componentes de uma aplicação distribuída através de agentes de reconfiguração. Esses agentes são injetados numa rede de componentes distribuídos com as reconfigurações a serem processadas. Cada nó da rede replica o agente e o repassa para o próximo nó, processa os comandos de reconfiguração, coleta os resultados da reconfiguração e os devolve ao nó anterior do qual recebeu o agente. Essa rede de componentes é especificada pelo administrador através de uma ferramenta visual.

Em sua tese, Kon ainda apresenta um sistema de distribuição de multimídia mostrando como a arquitetura de configuradores de componentes auxilia na construção de sistemas com tolerância a falhas [7].

Esse é o trabalho base para a nossa tese de onde foram extraídos as nossas principais idéias com as seguintes modificações:

- A plataforma foi modificada, ao invés do C++ com CORBA estamos utilizando a plataforma EJB que é uma arquitetura padrão de servidor baseado em componentes que devido às suas padronizações evitam as implementações *ad-hoc*;
- Os pré-requisitos também foram alterados. Ao invés do arquivo de especificação ser no formato proprietário *Simple Prerequisite Description Format* (SPDF), foi utilizado um arquivo XML que pode ser escrito de acordo com uma gramática criada, *Document Type Definition* (DTD), que proporciona a utilização de operadores booleanos (OR e AND) na especificação das dependências que será posteriormente detalhada;
- As formas de se obter os componentes de software apresentados nos pré-requisitos também foram estendidos, além de se obter o componente do repositório de componentes, pode-se utilizar componentes em outras máquinas através de referências. Para isso, essas referencias podem ser localizadas utilizado-se um serviço de nomes ou de diretórios que serão posteriormente detalhados;

2.5. CoDelivery [9]

O *CoDelivery* é uma arquitetura baseada no protocolo HTTP e em Java em que se pode catalogar, publicar, baixar e executar componentes reutilizáveis. Ele é baseado no conceito de composição e pressupõe que as aplicações Java são compostas por um conjunto de componentes distribuídos pela Internet.

A composição significa que as aplicações e componentes podem ser desenvolvidas integrando e utilizando recursos providos por componentes já disponíveis. Essa composição nada mais é que as dependências em tempo de execução entre os componentes, que expressam a comunicação e os relacionamentos entre eles. No caso de aplicações Java, essas dependências são caracterizadas por referências a outras classes como, por exemplo, cláusulas *extends* e *implements*. O CoDelivery não trata as classes do pacote padrão do Java como componentes.

Os componentes são classificados em três tipos:

- Básico: implementam funcionalidades sem utilização de outros recursos
- Composto: utilizam composição, eles implementam funcionalidades utilizando outros componentes, que por sua vez podem ser do tipo básico ou compostos.
- Aplicações: é somente um caso especial de componentes de composição com a adição de mais alguns atributos e operações.

Essa classificação é utilizada pelo algoritmo do CoDelivery, pois de acordo com a classificação do componente, ele é tratado de uma forma diferenciada.

O CoDelivery possui o conceito de domínio e cada produtor de objetos tem o controle sobre um determinado domínio onde ele pode criar sub-domínios e publicar os seus componentes. Cada componente tem o seu identificador que é universal e único na Internet. Um identificador segue a seguinte sintaxe: N/V, onde N e V são, respectivamente, o nome e versão do componente. Um exemplo de nome é com.enginz.examples.Click/1, que representa a primeira versão de um componente/aplicação Click da empresa produtora Enginz cujo domínio é *com.enginz*. O CoDelivery distingue, descobre, localiza e baixa os componentes utilizando um mecanismo baseado nesses identificadores.

Um componente básico é descrito no CoDelivery pelo seu identificador e implementação (arquivo de *bytecode* do Java). Um componente composto tem atributos adicionais especificando sua composição: identificadores dos componentes referenciados. Já a componente do tipo aplicação além dos identificadores dos componentes referenciados possui três parâmetros especificando o intervalo de tempo para atualização automática, remoção da aplicação e transferência antecipada dos componentes.

A arquitetura do sistema CoDelivery é composta por três camadas:

- Repositório (Application Repository Service): banco de dados distribuído para guardar e buscar componentes;
- Diretório (Application Directory Service): serviço de nomes para registrar e localizar componentes;
- Gerente (Application Management Service): gerencia localmente o ciclo de vida dos componentes no ambiente do usuário.

Todos os componentes estão sujeitos às regras de execução dos sistemas, assim, um componente pode ser instalado ou desinstalado de acordo com a configuração do sistema e com a funcionalidade requisitada pelo usuário, ou seja, somente os componentes referenciados em tempo de execução serão localizados, pegos e instalados na máquina do usuário.

Devido à atual infra-estrutura da Internet poder introduzir uma quantidade significativa de atraso na transferência dos componentes, o CoDelivery possui um parâmetro de configuração para cada componente, o *prefetch mode*. Esse parâmetro indica o número de níveis de composição que devem ter os códigos dos componentes transferidos antecipadamente.

Para que um componente seja disponibilizado no ambiente CoDelivery, ele precisa ser registrado pela sua empresa produtora no Serviço de Diretório e ter o seu código incluído no Serviço de Repositório. Ao registrar um componente é necessário incluir um identificador (Nome/Versão); a sua implementação (Arquivo .class do java); especificar o tipo; sua composição (dependência); intervalo de atualização; tempo de vida do componente; e o *prefetch mode*.

Como vemos, o CoDelivery é um trabalho onde foram resolvidos grandes problemas tais como a transparência de localização, administração de versões e economia de recursos, mas a granularidade escolhida, o arquivo class do java, pode ser muito pequena para sistemas de grande porte tornando muito complexo o desenvolvimento desses sistemas. No nosso trabalho o menor componente é um arquivo *jar* do java que pode conter desde um arquivo de *bytecode* (extensão .class) até vários arquivos *bytecode* que poderiam formar um aplicativo. Outra diferença é a plataforma EJB utilizada em nosso trabalho que já fornece toda uma infra-estrutura para os aplicativos a serem desenvolvidos que, no CoDelivery, devem ser desenvolvidos e catalogados.

2.6. Composição Automática de sistemas através de componentes com dependências anônimas [8]

Esse trabalho de composição de software é baseado em um algoritmo de combinação de componentes onde cada componente possui uma lista de propriedades exigidas (dependências) e propriedades fornecidas. Essas propriedades exigidas são chamadas de dependências anônimas, pois não têm em vista um componente especifico e sim uma propriedade que um ou mais componentes podem possuir. As necessidades dos clientes são especificadas em termo de ‘propriedades desejadas’ no sistema.

Esse trabalho propõe um modelo para se especificar as propriedades dos componentes que são relevantes na composição bem como um método para se determinar as dependências entre os componentes dinamicamente durante a composição. Seus autores também propõem que o método de composição deve ser independente do domínio da aplicação e que toda configuração que for específica ao domínio deve ficar na aplicação que fará a interface com o usuário final, ou seja, somente será montado pelo algoritmo de composição o que for independente da plataforma.

O algoritmo de composição considera que a aplicação a ser montada possui a arquitetura de camadas. Cada componente é considerado uma camada e provê um conjunto de serviços (propriedades) que podem ser utilizados por outros componentes nas camadas superiores. Assim a estratégia do algoritmo de composição proposto é 'resolver as propriedades requisitadas através das propriedades fornecidas'.

As propriedades requisitadas são classificadas de três formas:

1) De acordo com sua direção elas podem ser classificadas como *downward* ou *upward requirements*, ou seja, esse requisito precisa ser suprido por componentes das camadas abaixo ou acima, respectivamente.
2) De acordo com sua necessidade, o requisito pode ser classificado como um requisito forte ou fraco. Os requisitos fortes devem necessariamente ser supridos enquanto os requisitos fracos somente não devem ser contraditos, ou seja, se ele é um requisito *upward*, essa propriedade não pode existir nos componentes das camadas abaixo, mas também não precisa existir nas camadas acima.
3) Pode ser classificado como imediato (*immediate*) ou não, ou seja, o requisito deve ser suprido necessariamente pela camada adjacente.

Além dessas classificações, as propriedades requisitadas podem ser classificadas por ordem de preferência indicando que um determinado requisito deve ser suprido antes ou depois de outro requisito.

Dado um conjunto de propriedades desejadas do cliente, uma lista ordenada de componentes é considerada uma boa solução quando:

- Todos os requisitos são supridos, ou seja, estão presentes nas propriedades providas dos componentes da lista

- Os requisitos downward são satisfeitos
- Os requisitos upward são satisfeitos
- As seqüências impostas de solução de requisitos são obedecidas
- Não há contradição dos requisitos fracos.

O ponto de partida do algoritmo é procurar por pelo menos um componente que satisfaça uma das propriedades requisitadas pelo cliente formando uma lista de potenciais componentes. Dessa lista é escolhido um componente de acordo com alguns critérios:

- Não inserir componentes com propriedades já supridas
- Observar a ordem de preferência de solução de requisitos
- Não contradição dos requisitos fracos

Quando esse componente é encontrado é bem provável que ele possua os seus próprios requisitos formando um novo problema de composição. Passo a passo o problema será solucionado.

A composição de software não é o foco em nosso trabalho, contudo é possível se implementar o algoritmo apresentado nesse trabalho através da especificação de pré-requisitos por atributos que são semelhantes às dependências anônimas apresentadas nesse trabalho. Veremos no próximo capítulo o que são exatamente esses pré-requisitos por atributos (ver seção 3.3).

2.7. O Modelo DAS utilizando LEAD++

Noriko Amano e Takuo Watanabe trabalham com o conceito de capacidade de adaptação dinâmica de sistemas [13]. Segundo eles, um sistema tem essa capacidade se puder alterar seu comportamento de acordo com as mudanças do ambiente de execução em que ele se encontra. Essa alteração de comportamento pode se dar de duas formas:

- alteração de comportamento de cada componente que forma o sistema e
- alteração da estrutura de composição dos componentes em tempo de execução.

Eles propõem um modelo de sistema com capacidade de adaptação dinâmica que denominaram DAS [10][11] e utilizam o LEAD++ [12] como sua linguagem de descrição. O LEAD++ é uma linguagem reflexiva orientada a objetos baseada no Java.

Para realizar os dois tipos de alteração de comportamento, o LEAD++ utiliza três coisas:

1) procedimentos adaptáveis,
2) objeto de eventos e
3) um mecanismo de adaptação reflexiva.

O procedimento adaptável é uma variação dos procedimentos comuns (*functions*) cujos métodos são utilizados de acordo com o ambiente de execução. O objeto de eventos monitora e detecta mudanças de estado do ambiente de execução e chama os métodos de notificação (*call-back*) das mudanças. O mecanismo de adaptação reflexiva realiza as adaptações nos procedimentos adaptáveis.

Note que esse procedimento adaptável pode ser entendido mais ou menos como uma dependência, pois, dependendo do ambiente de execução, ele pode se comunicar (utilizar) com um componente distinto.

Considerando que um componente é um objeto baseado numa classe, e que os componentes se conectam entre si através de mensagens, ilustraremos, na figura a seguir, um exemplo de sistema baseado em componentes.

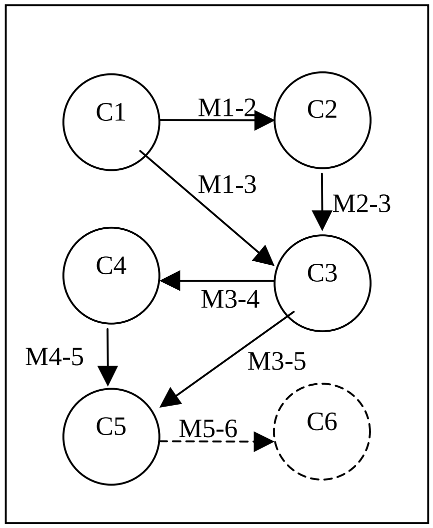


FIGURA 2.3 – EXEMPLO DO LEAD++

Na figura acima, o sistema apresentado é semelhante a um grafo, onde, Cn representa um componente e Mm-n representa uma mensagem de Cm até Cn. Essa mensagem corresponde a uma chamada de método. As linhas tracejadas são componentes em potencial que não estão instanciados.

Para se alterar o comportamento do sistema, o DAS se utiliza de dois recursos conforme já mencionado:

- Modificação da estrutura de composição dos componentes, ou seja, inclusão ou exclusão de algum componente no grafo.
- Modificação do comportamento de cada componente, ou seja, modificação do fluxo (mensagens) entre os componentes.

Em ambos os casos, a modificação implica numa mudança no grafo. Essa modificação se daria conforme a mudança de estado no ambiente de execução, assim, se supõe que os programadores desenvolvam os componentes já prevendo quais estados o ambiente de execução poderia assumir.

Na figura a seguir é mostrado um sistema em dois momentos distintos, cada um num ambiente de execução com estado diferente.

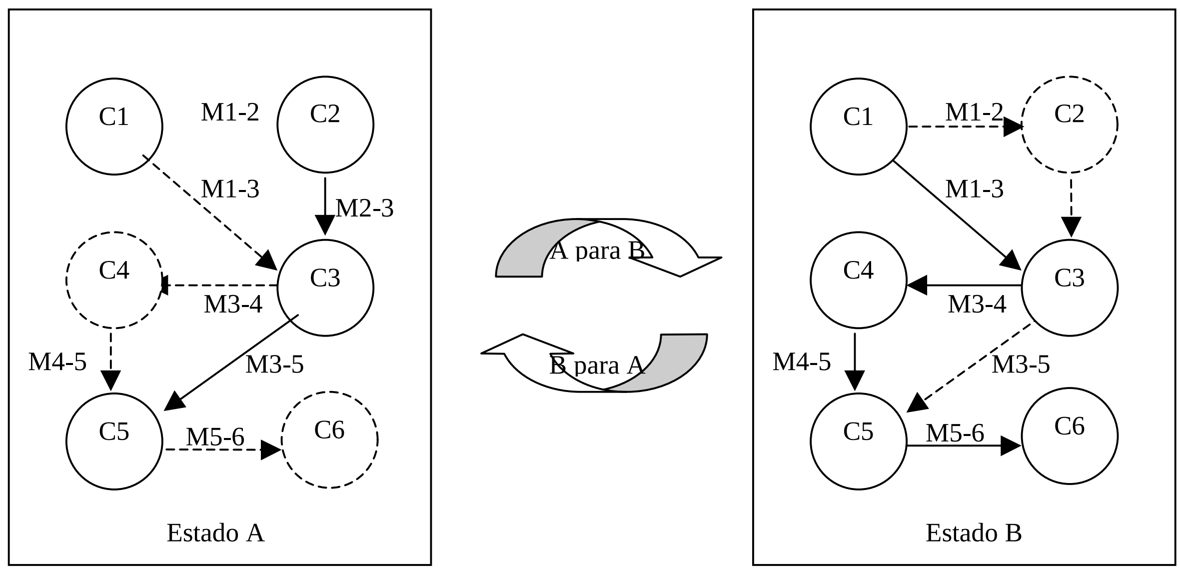


FIGURA 2.4 – EXEMPLO DE TROCA DE ESTADO NO LEAD++

Quando o estado muda do estado A para o B, o sistema realiza as seguintes adaptações:

1) C1 manda uma mensagem diretamente para C3 sem passar por C2;
2) C4 é instanciado e passa enviar mensagens para C5;
3) C3 deixa de enviar mensagens para C5 e passa enviar mensagens para C4;
4) C6 é instanciado e passa a receber mensagens de C5.

Note que além de instanciar novos componentes e excluir outros, houve uma mudança de comportamento de cada componente.

No LEAD++, para a implementação de sistemas com adaptação dinâmica, é necessário a construção de componentes com métodos adaptáveis de acordo com o variáveis do ambiente de execução. Para se juntar esses componentes, é necessário se compilar um objeto base denominado objeto *template* onde são predeterminados todos os *slots* (variáveis) para os componentes. Esse componente base será o responsável por juntar todos os componentes de maneira a formar o grafo da figura 2.3. Em nosso trabalho, para se utilizar uma estrutura nova, basta se modificar o arquivo XML que vai com cada componente, não necessitando recompilar esse componente base.

2.8. Tabela comparativa

Abaixo temos uma tabela comparativa entre os trabalhos relacionados.

	Configuração automática no 2K	CoDelivery	Dependências anônimas	LEAD++	Esta dissertação
Especificação de Dependência	Ok	Ok	Ok	Ok	Ok
Dependência de software	Ok	Ok	Ok	Ok	Ok
Especificação da dependência por atributos	-	-	Ok	-	Ok
Operadores booleanos na especificação	-	-	-	-	Ok
Controle de versão	-	Ok	-	-	-
Dependência de hardware	Ok	-	-	-	-

Tabela 2.1 – Comparação entre os trabalhos relacionados

Capítulo 3

Arquitetura do Sistema Proposto

3.1. Visão Geral

Na figura a seguir, apresentamos um esboço da arquitetura deste trabalho com quatro grandes blocos.

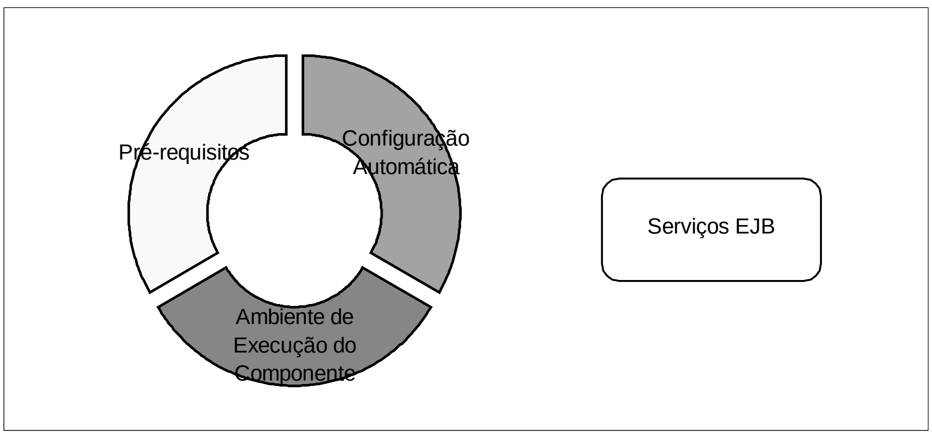


FIGURA 3.1 – VISÃO GERAL DA ARQUITETURA

Seguindo os conceitos apresentados no Sistema de Configuração Automática do 2K (ver seção 2.4), nosso trabalho possui os pré-requisitos que representam as dependências estáticas dos componentes enquanto o ambiente de execução do componente possui as dependências dinâmicas relativas ao tempo de execução. A configuração automática é responsável pelo gerenciamento dos componentes. Ele processa os pré-requisitos de cada componente inicializando o ambiente de execução do componente com os pré-requisitos selecionados. O processo de seleção será detalhado na seção 3.3.

Na figura a seguir mostramos como os outros blocos se relacionam.

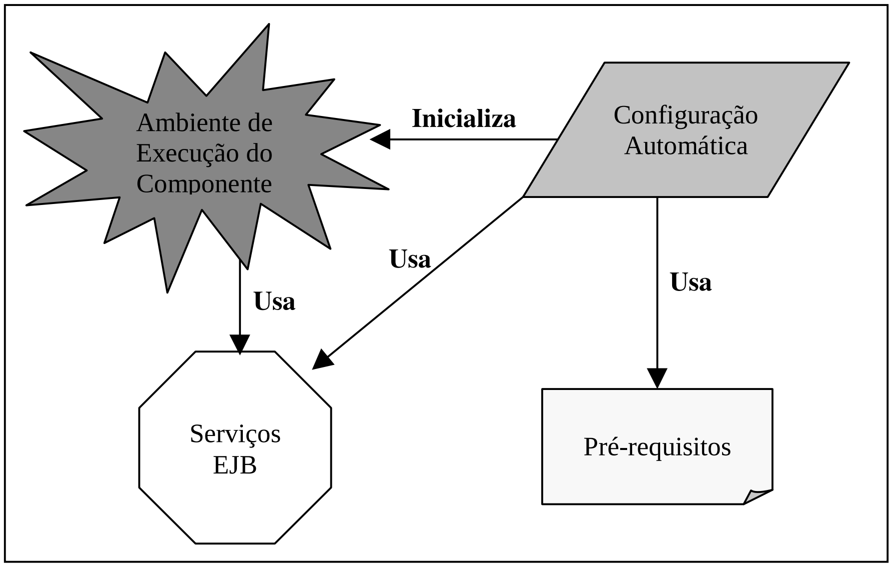


FIGURA 3.2 – RELACIONAMENTO DOS BLOCOS DA ARQUITETURA

A seguir descrevemos os principais blocos do nosso sistema.

3.2. Configuração Automática

Nosso configurador automático é semelhante ao Sistema de Configuração Automática do 2K e também visa automatizar o processo de instalação e configuração dos softwares utilizando o modelo "*What You Need Is What You Get*" (WYNIWYG). Em outras palavras, para fugirmos dos modelos atuais dos sistemas operacionais, middleware e aplicações que disponibilizam em sua instalação padrão diversos pacotes que não serão utilizados, o configurador automático é responsável por configurar automaticamente o sistema, a aplicação ou componente utilizando o mínimo de recursos necessários.

Nesse modelo de configuração automática, os sistemas (aplicações) são formados por uma rede de componentes. Esses componentes podem ser carregados e executados localmente a partir de diversos repositórios de componentes via protocolo HTTP ou já podem estar em execução, sendo localizados através de um nome ou de características.

Os repositórios de componentes nada mais são que servidores Web. Carregar códigos executáveis pela Internet trás riscos bem conhecidos no meio acadêmico, contudo, não faz parte do escopo desse trabalho utilizar uma infra-estrutura de segurança. Isso pode ser desenvolvido futuramente através da autenticação de cada servidor e da utilização de HTTPS.

Cada aplicação, sistema ou componente tem especificado todos os componentes necessários para o seu correto funcionamento. O conjunto desses requisitos necessários é chamado de pré-requisitos.

3.3. Pré-requisitos

A especificação dos pré-requisitos de um dado componente C1 contém todos componentes de que este depende para o seu correto funcionamento. A nossa especificação de pré-requisitos se diferencia do Sistema de Configuração Automática do 2K por proporcionar que o desenvolvedor especifique não somente dependências fixas dizendo somente se elas são opcionais ou não. No nosso trabalho, o desenvolvedor pode especificar um conjunto de componentes que podem servir para configurar corretamente o componente C1, ou seja, se um determinado componente não estiver disponível pode ser especificado um outro componente alternativo, ou um outro conjunto de componentes, para substituir esse componente que não foi encontrado. Um componente é considerado não disponível em dois casos:

1) Se ele não for encontrado ou
2) Se o componente não possuir suas dependências corretamente supridas.

Para um melhor entendimento, vamos ver um exemplo. Na especificação dos pré-requisitos de um componente C1 pode constar os componentes C2 e C3, e mais, se por acaso C3 não estiver disponível, pode ser utilizado o componente C4 em seu lugar. Assim, temos dois conjuntos de componentes como solução para os pré-requisitos de C1: C2 e C3 ou ainda, C2 e C4.

Isso pode ser descrito da seguinte forma:

```
C1 => C2 AND (C3  OR C4)
```

FIGURA 3.3A – EXEMPLO DE ESPECIFICAÇÃO (A)

ou

```
C1 => (C2 AND C3) OR (C2 AND C4)
```

FIGURA 3.3B – EXEMPLO DE ESPECIFICAÇÃO (B)

Os componentes C2, C3 e C4 também podem ter seus pré-requisitos.

Outra característica da especificação dos pré-requisitos em nosso trabalho é que este também deve informar a forma de resolução dessa dependência, ou seja, deve ser especificado onde procurar e como fazer a procura por essas dependências.

Existem três formas de resolução de dependências:

1) É fornecida a localização do código executável do componente desejado. Através dessa localização, é possível identificar o repositório de componentes e local nesse repositório onde o código executável do componente desejado se encontra. Esse código é baixado, carregado e executado no ambiente de execução pelo configurador automático;
2) É fornecido um nome que identifica e localiza o componente desejado;
3) São fornecidos atributos necessários do componente desejado. É procurado por componentes que possuam esses atributos. É possível se encontrar vários componentes com esses atributos, mas o configurador automático deve escolher somente um. Para simplificar, estamos escolhendo o primeiro componente encontrado, mas, futuramente, podemos fazer escolhas mais elaboradas.

Assim, no nosso exemplo anterior temos de especificar a forma de resolução de cada componente. Poderíamos dizer então que, C2 deve ser pego de um repositório de componentes cuja localização é http://www.ime.usp.br/~yutaka/repositorio/c2.jar, C3 deve ser encontrado pelo nome 'Com3' e que C4 deve ter os atributos 'Classificação', 'Tipo' e 'Nome' com os valores 'Algoritmo', 'Ordenação' e 'BubbleSort', respectivamente. O exemplo anterior pode ser então escrito da seguinte forma:

```
C1=> C2 [Repositório: http://www.ime.usp.br/~yutaka/repositorio/c2.jar] AND
(C3 [Nome: Com3] OR C4 [Atributos: Classificação=Algoritmo, Tipo=Ordenação,
Nome=BubbleSort])
```

FIGURA 3.4 – EXEMPLO COM ESPECIFICAÇÃO MAIS COMPLETA

Como exemplo prático podemos considerar um editor de texto que poderia especificar na sua lista de pré-requisitos um componente gerenciador de janelas, outro para fazer o acesso ao serviço de arquivos, outro componente gerenciador de fontes, outro para impressão, entre outras coisas. O configurador automático seria responsável por localizar, baixar, instalar e executar o editor formado por todos estes componentes.

3.4. Ambiente de execução do componente

Uma vez obtida a especificação dos pré-requisitos de um componente C1 que se deseja instalar, o configurador automático irá processá-los e procurar por um conjunto de componentes disponíveis de acordo com a forma de resolução de cada pré-requisito. Esse conjunto de componentes encontrado deve ser conectado ao componente C1 para que possam ser utilizados. Essa conexão é realizada através de *ganchos* que todo componente pode possuir.

Os ganchos nada mais são que referências para outros componentes. Cada componente tem um número predeterminado de *ganchos* onde componentes de que se depende devem ser conectados. Cada gancho deve conter um componente com uma predeterminada funcionalidade. Toda vez que o componente necessitar dessa funcionalidade, ele irá utilizar o componente conectado ao gancho relativo à funcionalidade desejada. Assim, na especificação dos pré-requisitos de um componente, deve-se também informar a qual gancho cada componente irá se conectar. O nosso exemplo de especificação pode ser então escrito da seguinte forma:

```
C1=> C2 [Repositório: http://www.ime.usp.br/~yutaka/repositorio/c2.jar,
Gancho:G1] AND (C3 [Nome: Com3, Gancho:G2] OR C4 [Atributos:
Classificação=Algoritmo, Tipo=Ordenação], Gancho:G2)
```

FIGURA 3.5 – EXEMPLO DE ESPECIFICAÇÃO FINAL

A conexão dos componentes encontrados pelo configurador automático ao componente C1 é realizada em duas etapas:

1) Referências para os componentes disponíveis do pré-requisito são disponibilizadas no ambiente de execução;
2) A primeira vez que o gancho for acessado pelo componente, ele irá procurar no ambiente de execução pela referência disponibilizada.

Na figura abaixo ilustramos nosso exemplo anterior. As setas são as referências do ambiente de execução que apontam para os componentes que satisfazem os pré-requisitos de C1. As linhas pontilhadas indicam que C4 não está sendo utilizado, somente é uma opção a ser utilizado no gancho G2.

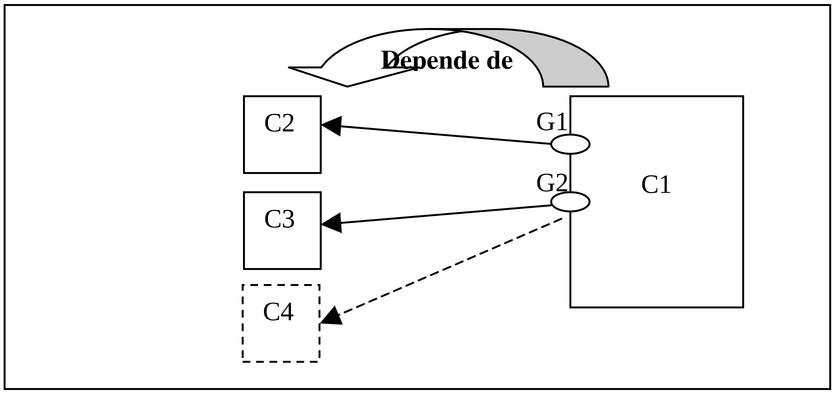


FIGURA 3.6 – EXEMPLO DE DEPENDÊNCIA

Caso C4 tenha também pré-requisitos, nosso exemplo poderia ser escrito da seguinte forma:

```
C1=> C2 [Repositório: http://www.ime.usp.br/~yutaka/repositorio/c2.jar, Gancho:G1] AND (C3 [Nome: Com3, Gancho:G2] OR C4 [Atributos: Classificação=Algoritmo, Tipo=Ordenação, Nome=BubbleSort], Gancho:G2)

C4=> C5 [Nome: Com5, Gancho:G4]
```

FIGURA 3.7 – EXEMPLO DE ESPECIFICAÇÃO DE DOIS COMPONENTES

Teríamos então a seguinte figura:

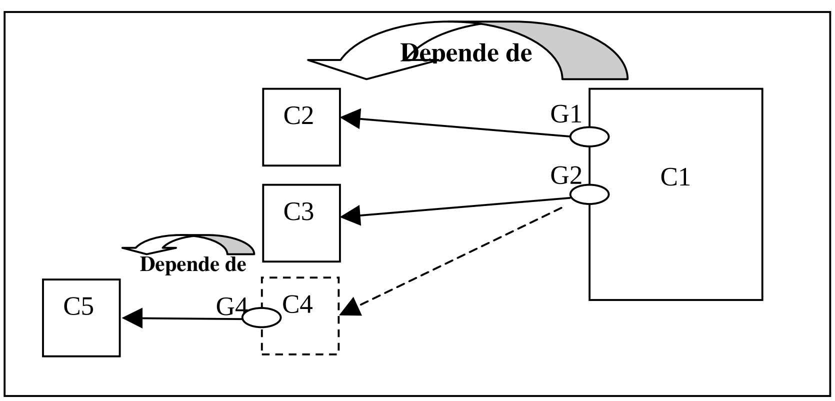


FIGURA 3.8 – EXEMPLO DE DEPENDÊNCIA MAIS COMPLETA

É importante lembrar que o componente C4 e suas dependências só serão processados caso C3 não esteja disponível.

Capítulo 4

Implementação

Nesta seção apresentaremos as tecnologias que foram utilizadas para a implementação do nosso trabalho. Também mostraremos como cada uma delas foi utilizada.

4.1. Tecnologias Utilizadas

4.1.1. Enterprise JavaBeans – EJB

O nosso trabalho foi baseado na arquitetura *Enterprise JavaBeans* (EJB) [14][18] que é uma arquitetura de servidores baseado em componentes para o desenvolvimento e implantação de aplicações distribuídas orientadas a objetos. A arquitetura EJB é baseada no *Java Remote Method Invocation* (RMI) fornecendo ao desenvolvedor um arcabouço para a construção de sistemas distribuídos facilitando em muito a construção de softwares no modelo *three-tier* (três camadas) ou *n-tier*.

A arquitetura EJB define um servidor de aplicação que disponibiliza um contêiner no qual são implantados componentes.

A figura 4.1 a seguir nos ilustra a arquitetura EJB:

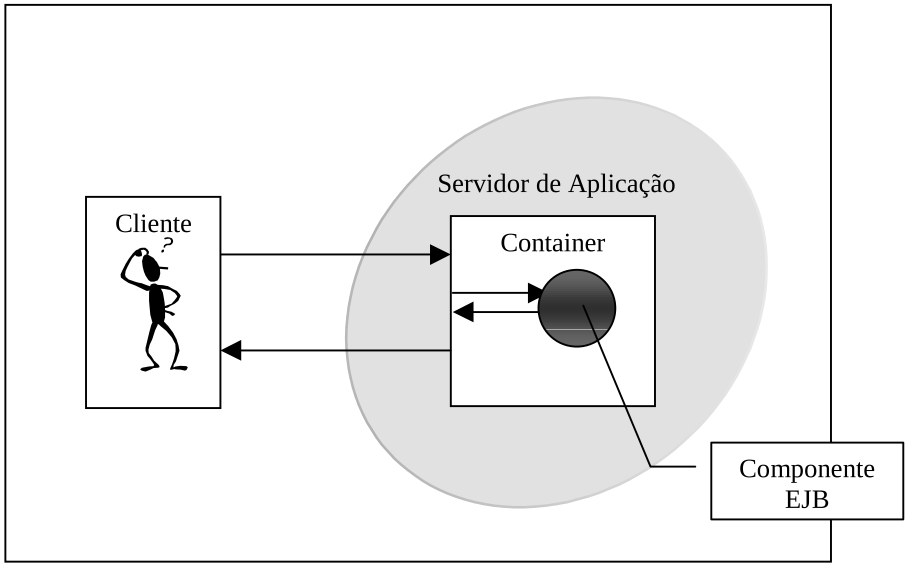


FIGURA 4.1 – ARQUITETURA EJB

O servidor de aplicação é responsável:

- Por fornecer contêiners onde componentes EJB possam ser implantados;
- Pela alocação de recursos como *threads*, processos, memória e conexões a banco de dados;
- Prover acesso a serviços como serviço de nomes, diretório, de segurança, de transação e de persistência.

O contêiner é quem faz a interface entre o componente e o ‘mundo exterior’. Os clientes nunca acessam um componente diretamente. O contêiner é responsável por:

- Fornecer o contexto onde o componente é executado;
- Registrar o componente no serviço de nomes;
- Fornecer interface remota do componente, criar e destruir instâncias;
- Gerenciar transações, estado e persistência.

Existem dois tipos de interface:

- Interface *home*, que define os métodos do ciclo de vida do componente: criação, remoção e busca.
- Interface *remote*, que define os métodos de negócio.

O componente EJB é onde ficam as implementações do desenvolvedor. Um componente EJB é um arquivo do tipo JAR contendo um ou mais arquivos de *bytecode* do Java (arquivo de extensão .class) mais um descritor de implantação. Esse descritor de implantação contém várias informações sobre o componente. Seguem alguns exemplos de informações que podem ser incluídas no descritor de implantação:

- Nome do componente
- Se o componente é persistente ou não;
- Quem é responsável pela persistência do componente, o contêiner ou o próprio componente;
- Se o componente possui estado ou não, ou seja, se ao responder uma requisição, o componente deve ou não levar em consideração as requisições anteriores;
- Nome completo das interfaces *home* e *remote* do componente;
- Referências EJB que referenciam interfaces *home* de outros objetos que serão utilizados pelo componente.
- Entre outras coisas.

O descritor de implementação é um arquivo XML [17] escrito de acordo com uma *Document Type Definition* (DTD) definida pela especificação EJB [18].

4.1.1.1. Referências EJB e o serviço de nomes

Uma característica muito importante da arquitetura EJB para o nosso trabalho são as referências EJB (ejb-ref) de um componente. Elas foram utilizadas em nosso trabalho para implementar o conceito de ganchos descrito na seção 3.4.

Essas referências foram criadas para o desenvolvimento aplicações, ou de outros componentes, através da composição de componentes EJB já existente. Para se realizar esta composição, basta se agrupar os componente EJB necessários para a criação do componente, ou aplicação, em um arquivo do tipo Jar e descrever esta composição em termos de que componentes se conectam em quais ejb-ref. Não é necessária nenhuma recompilação.

Cada componente EJB possui um registro no serviço de nomes identificado por um nome que está associado a um objeto do tipo *proxy*, definido na API de Java, pelo qual ele será acessado. Esse registro é configurado via JNDI (veja subseção 4.1.4). Um nome pode ser criado dentro de um contexto C1 que, por sua vez, pode estar localizado dentro de outro contexto C2, com isso, o contexto C1 é chamado de sub-contexto de C2. Pode-se fazer uma analogia do serviço de nomes ao serviço de arquivos de um sistema operacional onde um arquivo é criado dentro de um diretório que pode estar dentro de outro diretório.

Na arquitetura EJB, cada componente EJB tem seu nome registrado em um contexto específico onde se pode encontrar referências para todos os componentes disponíveis neste servidor EJB. Também cada componente tem o seu próprio contexto que é utilizado para sua configuração.

Sabendo que cada componente EJB tem seu nome cadastrado no serviço de nomes, os ejb-ref são configurados simplesmente associando-se um nome ao ejb-ref. Essa associação é cadastrada no serviço de nomes como um registro no contexto do componente onde temos os nomes das referências EJB (ejb-ref) relacionados a um nome de um componente EJB que preenche essa referência. Assim, para se fazer a composição dos componentes, basta se configurar o serviço de nomes corretamente. Essa configuração se dá através do arquivo ejb-jar.xml.

4.1.2. JBOSS/Java Management Extensions (JMX)

Escolhemos o JBoss [28] como implementação da arquitetura EJB sobre a qual iríamos desenvolver esse trabalho. Essa escolha foi devido ao fato de seu código fonte estar disponível no sítio http://www.jboss.org, ser gratuito, inclusive para uso comercial e, devido à facilidade de se acrescentar novos módulos.

O JBoss é uma implementação da especificação EJB 2.0 e fornece, juntamente com o servidor EJB, o contêiner EJB. Uma grande facilidade oferecida pelo JBoss é o *hot deployment*. Basta se copiar um arquivo do tipo JAR contendo um componente EJB no seu diretório de implantação que ele será automaticamente implantado, ou se já estiver implantado, desinstalado e implantado novamente.

O JBoss foi desenvolvido em cima do *Java Management Extension* (JMX) [27] da Sun que define uma arquitetura, padrões de desenho, APIs e serviços para gerenciamento de aplicações e de redes baseado na linguagem Java.

O JMX é composto de três partes:

1) Ferramentas (*Instrumentation Level*), são os objetos gerenciáveis. Eles são chamados de *Managed Beans* (MBeans);
2) Agentes (*Agent Level*), são objetos que realizam os serviços de gerenciamento das aplicações. Existe um objeto denominado *servidor MBean*, onde os objetos gerenciáveis são registrados;
3) Serviços Distribuídos (*Distributed Services Level*), são objetos que fornecem uma interface para se implementar o gerenciamento de agentes JMX de forma remota.

O JBoss utiliza essa arquitetura registrando MBeans no servidor MBean com todas as funcionalidades necessárias para a implementação da arquitetura EJB. Esses MBeans são utilizados através da interface provida pelo JMX.

4.1.2.1. Dependências no JBoss

A partir da versão 3.0, lançado no meio do ano de 2002, o JBoss possui suporte para se descrever as dependências de um componente EJB. Contudo, ele se limitar somente a não prosseguir com a implantação de um componente C, aguardando que um componente C1 do qual se depende seja instalado. Assim que C1 for instalado, a implantação do componente C tem prosseguimento.

Essa abordagem não foi utilizada em nosso trabalho, pois o que nos propomos a fazer vai além do realizado pelo JBoss. No nosso trabalho, ao se instalar um componente C que tem como dependência um componente C1, se C1 não foi instalado com sucesso por causa da falta de um determinado componente C2, podemos ter outros componentes como alternativa para o componente C1 não interessando que, quando C2 for instalado, a instalação de C1 tenha prosseguimento.

4.1.3. *Lightweight Directory Access Protocol* (LDAP) [21]

Nesse trabalho imaginávamos implementar algo semelhante ao Serviço de Negociação do CORBA (*Object Trading Service*) [20] pois desejávamos procurar por componentes não somente através de um nome, mas, também, através de atributos desejáveis em um componente.

A arquitetura EJB não possui uma interface para esse serviço do CORBA por isso foi utilizado a API JNDI do Java, que será abordada na próxima subseção, que fornece acesso ao serviço de diretórios que tem funcionalidade semelhante ao do *trading service*.

O serviço de diretórios se diferencia do serviço de nomes pela possibilidade de se fazer buscas e de se recuperar informações nomeadas. Os acessos de leitura são mais comuns do que os de atualização.

O serviço de diretórios escolhido foi o *Lightweight Directory Access Protocol* (LDAP) que é derivado do *Directory Access Protocol* (DAP) do X.500 [22]. O LDAP foi desenvolvido como uma opção ao DAP possuindo um menor custo computacional. Com o LDAP é possível nomear, gerenciar e acessar coleções de pares de atributo-valor.

O modelo de nomes do LDAP é semelhante a uma árvore. Cada nó tem um nome que é denominado *relative distinguished name* (RDN), que é relativo ao nó pai, e um *distinguished name* (DN), que é a especificação do caminho da raiz até o nó.

A implementação LDAP utilizada foi o 'Iplanet Server' da Netscape [24] devido a possibilidade de execução em ambiente Windows. Atualmente essa solução foi incorporada ao produto *Sun Open Net Enviroment* (Sun ONE). Contudo, uma opção gratuita seria o OpenLDAP [23] para o sistema operacional Linux.

Apesar de utilizarmos o LDAP para se procurar componentes de acordo com seu atributos, como descreveremos na seção 4.2.2.2, nesse trabalho não iremos nos preocupar com a ontologia [25] dos atributos.

4.1.4. *Java Naming Directory Interface* (JNDI) [19]

A arquitetura EJB não define uma forma de acesso a um serviço LDAP, para isso, então, utilizamos a API JNDI.

O JNDI é uma API que possibilita que as aplicações baseadas em JAVA acessem servidores de nomes e de diretórios como:

- Sistema de arquivos;
- Serviço de diretórios com o *Lightweight Directory Access Protocol* (LDAP);
- *Network Information System* (NIS);
- Serviço de Nomes do *Common Object Request Broker Architecture* (CORBA);
- Registry do *Java Remote Method Invocation* (RMI);

A arquitetura JNDI é composta de uma API e de uma interface para provedores de serviço - *Service Provider Interface* (SPI). A API JNDI é utilizada pelas aplicações Java para acessar os diversos tipos de serviços de nomes e de diretórios. A SPI JNDI possibilita que os serviços de nomes e diretórios sejam conectados e utilizados de forma transparente.

A figura a seguir fornece uma visão dessa arquitetura:

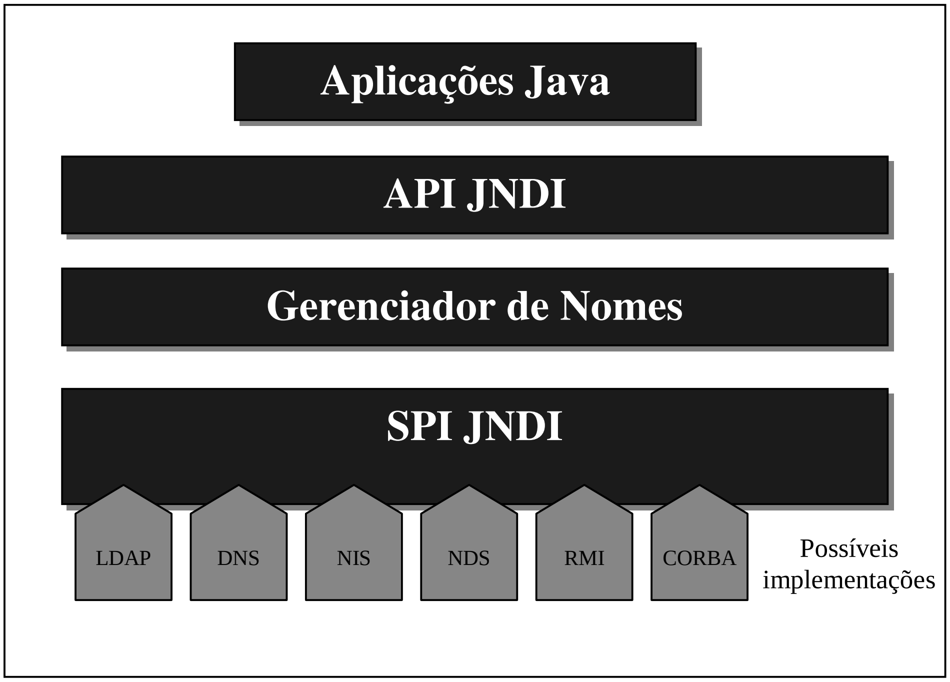


FIGURA 4.2 – ARQUITETURA JNDI

4.1.5. Extensible Markup Language (XML)

Nesse trabalho precisávamos descrever as dependências dos componentes num formato aberto e, como o descritor de implantação dos componentes da arquitetura EJB é em XML, nada mais natural que o nosso descritor de dependência também o fosse.

Para isso definimos uma gramática (ver seção 4.2.1) que especifica como deve ser escrito esse descritor de dependências. Essa gramática, *Document Type Definition* (DTD), nada mais é que definições léxicas do descritor. Definições semânticas devem aparecer nos comentários de cada *tag* do DTD.

4.2. Utilização das Tecnologias

Para que fosse implementada a configuração automática no JBoss, foi necessária a inclusão de um MBean que chamamos de AutoConfMBean. Esse MBean será o responsável por toda a configuração automática sendo chamado pelo JBoss através de funções providas pelo JMX.

Para se especificar os pré-requisitos de um componente EJB utilizamos um arquivo XML (autoconf.xml) que deverá ser empacotado junto com o componente no arquivo do tipo JAR. O DTD criado será apresentado na seção seguinte.

O JBoss também foi alterado de forma que, ao se processar os arquivos ejb-jar.xml e jboss.xml, arquivos de implantação da especificação EJB e de extensão do JBoss respectivamente, também se processe o nosso arquivo autoconf.xml. Isso é feito também através de uma chamada de função JMX, uma vez que quem irá processa-lo é o nosso módulo AutoConfMBean.

```
server.invoke(new ObjectName("user:service=AutoConf"),
        "configureDependency",
        new Object[] { getAutoConfMetaData() },
        new String[] { br.usp.ime.autoconf.metadata.AutoConfMetaData.class.getName()
});
```

FIGURA 4.3 – EXEMPLO DE CHAMADA DE MÉTODO VIA JMX.

Na figura acima temos um exemplo de como é executada essa chamada ao AutoConfMBean onde:

- **Server** é uma variável que contém um apontador para a instância MBeanServer que contém todos os MBeans cadastrados;
- **Invoke** é um método, pelo qual se faz chamadas a métodos dos MBeans cadastrados, que possui os seguintes parâmetros:
    1) O nome com o qual o Mbean de que se deseja executar uma função foi cadastrado,
    2) O nome da função que se deseja executar;

3) Um vetor com os parâmetros para essa função que se está querendo executar;
4) Um vetor com os tipos de cada parâmetro enviado.

4.2.1. Definição do Document Type Definition (DTD)

Segue o DTD criado:

```
<!--
   Esse é o DTD do módulo de configuração automática do JBoss que define seus
   elementos e atributos
   -->

<!-- ==========================================================================
  O elemento AUTOCONF é o elemento raiz do arquivo auto-conf.xml. Ele possui
  Configurações que não constam nem no arquivo ejb-jar.jar nem no jboss.xml.

  1) Nele está contida a lista dos pares (atributo, valor) relativo ao
  componente.
  2) Também possui todas as informações de dependência do componente.
  -->
<!ELEMENT autoconf (software-dependency)>

<!-- ==========================================================================
  O elemento ENTERPRISE-BEANS contém a configuração do componente no serviço de
  Diretórios.

  Utilizado em: AUTOCONF
  -->
<!ELEMENT enterprise-beans (session | entity)+>

<!-- ==========================================================================
  O elemento SESSION contém as informações relativas ao bean session que não
  constam no arquivo ejb-jar.xml nem no arquivo jboss.xml. Esse bean tem de
```

```
  estar declarado no ejb-jar.xml. (Veja as tags para maiores detalhes)

  Utilizado em: ENTERPRISE-BEANS
  -->
<!ELEMENT session (ejb-name, directory-service?, software-dependency?>

<!-- ==============================================================================
  O elemento ENTITY contém as informações relativas ao bean entity que não
  constam no arquivo ejb-jar.xml nem no arquivo jboss.xml. Esse bean tem que
  estar declarado no ejb-jar.xml. (Veja as tags para maiores detalhes)

  Utilizado em: ENTERPRISE-BEANS
  -->
<!ELEMENT entity (ejb-name, directory-service?, software-dependency?>

!-- ==============================================================================
  O elemento SOFTWARE-DEPENDENCY contém as dependências do componente.
  Essas dependências podem ser expressas utilizando expressões booleanas OR ou
  AND. Para cada ejb-ref declarado no arquivo ejb-jar.xml, devemos ter um
  componente declarado no autoconf.xml que possa preencher a referência.
  cada um dos elementos (OR, AND, COMPONENT) que irão compor o
  SOFTWARE-DEPENDENCY devem conter elementos suficientes para preencher todos os
  pré-requisitos do componente, ou seja, se ele possuir somente um elemento,
  esse elemento deve conter pelo menos um componente para cada pré-requisito do
  componente. Essa característica é necessária, pois o SOFTWARE-DEPENDENCY irá
  agir como um elemento OR para que se possa especificar mais de um conjunto
  de componentes como solução para os pré-requisitos do componente.

      Ex: O componente c1 tem dois ejb-ref, ref1 e ref2. Então uma possível
      especificação dos pré-requisitos é:
         O componente c2 para ref1 e c3 para ref2
         ou o componente c4 para ref1 e c5 para ref2.

  Utilizado em: SESSION, ENTITY
  -->
<!ELEMENT software-dependency (or | and | component)+>

<!-- ==============================================================================
```

```
  O elemento OR implementa a função booleana 'ou'. Assim, o módulo de
  configuração automática irá considerar que a dependência está resolvida quando
  UM dos elementos do OR estiver resolvido.

  Utilizado em: SOFTWARE-DEPENDENCY, AND, OR
  -->
<!ELEMENT or (or | and | component)+>

<!-- ==========================================================================
  O elemento AND implementa a função booleana 'e'. Assim, o módulo de
  configuração automática irá considerar que a dependência está resolvida
  quando TODOS os elementos do AND estiverem resolvidos.

  Utilizado em: SOFTWARE-DEPENDENCY, AND, OR
  -->
<!ELEMENT and (or | and | component)+>

<!-- ==========================================================================
  O elemento COMPONENT contém o 'componente' que irá resolver as dependências e
  preencher as referências EJB. Esse 'componente' pode ser resolvido de três
  formas:

  1) Via serviço de nomes
  2) Via serviço de diretórios
  3) Via repositório de componentes.

  Utilizado em: SOFTWARE-DEPENDENCY, AND, OR
  -->
<!ELEMENT component (name-service | directory-service | component-repository)>
<!--
  O atributo HOOK é utilizado para indicar qual referência EJB esse elemento
  irá preencher.
  -->
<!ATTLIST component hook (#PCDATA) #REQUIRED>

<!-- ==========================================================================
  O elemento NAME-SERVICE representa uma dependência do tipo serviço de nomes.
  Indica o nome pelo qual se deve procurar no serviço de nomes.
```

```
  Utilizado em: COMPONENT
  -->
<!ELEMENT name-service (name)>

<!-- ===============================================================================
  Indica o nome pelo qual se deve procurar no serviço de nomes.
  Também é utilizado como nome de um atributo no serviço de diretórios.

  Utilizado em: NAME-SERVICE, ATTRIBUTE
  -->
<!ELEMENT name (#PCDATA)>

<!-- ===============================================================================
  O elemento DIRECTORY-SERVICE representa uma dependência do tipo serviço de
  diretório. Indica os pares (atributo, valor) pelo qual se deve procurar no
  serviço de diretórios. Também é utilizado para se atribuir os pares
  (atributo, valor) para um componente EJB.

  Utilizado em: COMPONENT, SESSION, ENTITY
  -->
<!ELEMENT diretory-service (context | attribute+)>

<!-- ===============================================================================
  O elemento CONTEXT contém um par (atributo, valor) que indica o contexto em
que
  deve ser criado ou procurado um componente EJB.

  Utilizado em: DIRECTORY-SERVICE
  -->
<!ELEMENT context (context | attribute+)>

<!-- ===============================================================================
  O elemento ATTRIBUTE contém um par atributo=valor que deve ser procurado ou
  atribuído no serviço de diretórios ou que deve ser atribuído a um componente
  EJB.

  Utilizado em: DIRECTORY-SERVICE, CONTEXT
```

```
  -->
<!ELEMENT attribute (name, value)>

<!-- ==========================================================================
  O elemento VALUE contém o valor de um atributo.

  Utilizado em: ATTRIBUTE
  -->
<!ELEMENT value (#PCDATA)>

<!-- ==========================================================================
  O elemento COMPONENT-REPOSITORY representa uma dependência do tipo repositório
  de componente, ou seja, deve ser procurado por um componente no repositório a
  fim de resolver uma dependência do componente EJB.

  Utilizado em: COMPONENT
  -->
<!ELEMENT component-repository (link)>

<!-- ==========================================================================
  O elemento LINK especifica o repositório de componente em que se encontra o
  Componente procurado e sua localização. Nada mais é que um web link.

  Utilizado em: COMPONENT
  -->
<!ELEMENT link (#PCDATA)>
```

FIGURA 4.4 – ESPECIFICAÇÃO DO DTD DO AUTOCONF.XML

Um fato importante de se notar é o caráter recursivo do DTD definido. Os elementos OR e AND podem possuir quantos elementos forem necessários, inclusive outros elementos OR e AND.

Como trabalhamos com lógica booleana, falaremos em termos de se um marcador (elemento) está resolvido ou não. Assim, se um determinado componente, que é indicado pelo marcador COMPONENT, não estiver disponível no momento ou, se por algum motivo não for possível implantá-lo com sucesso, diremos então que o marcador COMPONENT não foi resolvido. Com isso, o elemento que contém esse componente, que pode ser um OR ou AND, deve tratá-lo como não resolvido.

Para que um marcador OR seja dado como resolvido, basta que um dos elementos contidos nele seja resolvido. Já o marcador AND, para ser dado como resolvido, deve possuir todos os elementos nele contidos resolvidos.

A sintaxe escolhida para o marcador SOFTWARE-DEPENDENCY, por convenção, é igual ao marcador OR.

4.2.1.1. Exemplo de um autoconf.xml

```
<autoconf>
  <enterprise-beans>
    <session>
      <ejb-name>Exemplo1</ejb-name>
      <software-dependency>
        <and>
          <component hook='G1'>
            <component-repository>
              <link>http://www.ime.usp.br/~yutaka/repositorio/c2.jar</link>
            </component-repository>
          </component>
          <or>
            <component hook='G2'>
              <name-service>
                <name>com3</name>
              </name-service>
            </component>
            <component hook='G2'>
              <directory-service>
```

```
20                <atribute>
21                  <name>Classificação</name>
22                  <value>Algoritmo</value>
23                </atribute>
24                <atribute>
25                  <name>Tipo</name>
26                  <value>Ordenação</value>
27                </atribute>
28              </directory-service>
29            </component>
30          </or>
31        </and>
32      </software-dependency>
33    </session>
34  </enterprise-beans>
35 </autoconf>
```

FIGURA 4.5 – AUTOCONF.XML DO COMPONENTE C1.

Na figura 4.5 descrevemos os pré-requisitos do componente C1 do exemplo da figura 3.3(a).

Na linha 4 temos declarado o nome do componente EJB. Esse nome tem de ser o mesmo utilizado para nomear o componente EJB no arquivo ejb-jar.xml, que é o arquivo descritor de implantação do componente EJB especificado na arquitetura EJB. Logo na linha seguinte iniciamos a declaração dos pré-requisitos do componente e sua localização.

O marcador AND da linha 6 contém dois elementos: o componente C2 (linha 7) e um elemento OR (linha 12). O elemento OR possui também dois elementos: o componente C3 e o componente C4. Para que o marcador AND seja considerado resolvido, tanto o C2 como o OR devem ser resolvidos. Por sua vez, para que o marcador OR seja considerado resolvido, ou C3 ou C4 devem ser resolvidos com sucesso.

4.2.2. Resolução dos pré-requisitos

Os pré-requisitos podem ser de dois tipos:

- Componentes a serem baixados do repositório de componentes ou
- Componentes, remotos ou não, a serem utilizados por referência

Essas referências podem ser adquiridas de duas formas:

- Pelo serviço de nomes ou
- Pelo serviço de diretórios.

Assim, temos três formas de resolução de pré-requisitos:

1) Via instalação de componentes do repositório de componentes;
2) Via aquisição de referência a um componente via serviço de nomes;
3) Via aquisição de referência a um componente via serviço de diretórios.

4.2.2.1. Repositório de componentes

Uma vez encontrado um pré-requisito que deve ser resolvido via repositório de componentes, o configurador automático tem de realizar as seguintes ações:

- Baixar o código do componente;
- Instalá-lo na máquina local.

Nos serviços disponibilizados pelo servidor EJB JBoss, um deles é a instalação de um componente EJB via HTTP. Basta se fornecer a URL do arquivo que contém o pacote JAR contendo o código executável do componente EJB que ele é baixado e instalado. Esse serviço nada mais é que uma chamada, via JMX, de uma função disponível no MBean EJBDeployer que o MBean é responsável no JBoss por implantar os componentes EJB. Esse MBean irá realizar todo o processo de transferência do arquivo e implantação.

4.2.2.2. Serviço de diretório

Ao se encontrar um pré-requisito que deve ser resolvido via serviço de diretório, o configurador automático utiliza o JNDI para se ter acesso ao serviço de diretório.

Alteramos o JBoss de forma que, caso esteja especificado no arquivo autoconf.xml, ao se implantar um componente, este é registrado no serviço de diretórios conforme dados e atributos fornecidos no mesmo arquivo. Esse registro é realizado através da inclusão de uma entrada no diretório LDAP contendo como atributo o nome do componente EJB no serviço de nomes, mais outros atributos especificados no autoconf.xml.

No nosso trabalho, podemos especificar em que nó (ou contexto) do LDAP deve ser procurado por componentes com os atributos fornecidos, então, a primeira coisa que o configurador automático tem de fazer é procurar pelo contexto em que se vai fazer as buscas para depois procurá-lo pelos elementos desse nó que possuem os atributos requisitados.

Uma vez executada a busca, pode-se encontrar um ou mais componentes que possuem os atributos requisitados. Caso se encontre mais de um componente, o configurador automático escolhe um qualquer.

4.2.2.3. Serviço de nomes

No caso do pré-requisito ser do tipo serviço de nomes, o configurador automático não necessita executar nenhuma tarefa especifica. O nome fornecido como pré-requisito será associado diretamente ao seu gancho, configurando o ambiente de execução do componente (ver seção 3.4).

4.2.3. Algoritmo de resolução dos pré-requisitos (dependências)

Para que não seja preciso acessar o arquivo XML toda vez que for necessário consultar alguma configuração de pré-requisitos e, seguindo o padrão de programação do JBoss, foi criada uma estrutura de dados que irá representar o XML em tempo de execução. Cada elemento (tag) do XML possui a sua respectiva estrutura de dados que é chamado de MetaDado.

O algoritmo de configuração automática recebe essa estrutura como parâmetro e executa a configuração automática percorrendo essa estrutura de forma linear na mesma seqüência que se encontra no arquivo autoconf.xml.

Cada MetaDado possui um estado que pode ser 'resolvido' ou 'não resolvido'. Também possuem uma tabela de *hash* que conterá os resultados da resolução de elementos contidos no MetaDado. Esse resultado é o nome do gancho (nome do ejb-ref) e um nome de um componente no serviço de nomes.

A seguir, temos o tratamento dos principais MetaDados.

4.2.3.1. MetaDado do elemento SOFTWARE-DEPENDENCY

Ao encontrar o MetaDado SOFTWARE-DEPENDENCY, o configurador automático irá tratar os seus elementos com a mesma lógica do OR, conforme mencionado anteriormente (veja seção 4.2.1).

4.2.3.2. MetaDado do elemento OR

Ao encontrar o MetaDado OR, o configurador automático irá tentar resolver pelo menos um elemento por ele contido. Com isso, ele vai percorrendo todos os seus elementos e tentando resolvê-los e, assim que um elemento for resolvido ele irá terminar o processamento desse MetaDado atribuindo a ele o estado de resolvido.

4.2.3.3. MetaDado do elemento AND

Ao encontrar o MetaDado AND, o configurador automático irá tentar resolver todos os elementos por ele contidos. Com isso, ele irá percorrer cada um de seus elementos tentando resolvê-los. Se ao final, todos estiverem resolvidos, o MetaDado AND terá o estado de resolvido, mas, se pelo contrário, algum elemento não puder ser resolvido com sucesso, o processamento irá parar e o estado desse MetaDado será de não resolvido.

4.2.3.4. MetaDado do elemento COMPONENT

Ao encontrar um MetaDado do elemento COMPONENT, ele verifica que tipo de pré-requisito ele contém. Se for do tipo a ser resolvido via serviço de nomes, o configurador automático irá verificar se o nome dado realmente se encontra cadastrado no serviço de nomes, caso positivo, então é atribuído o estado de resolvido ao MetaDado.

Se for do tipo a ser resolvido via serviço de diretórios, o configurador automático irá verificar se existe algum objeto cadastrado no serviço de diretórios que apresente todos os atributos relacionados no MetaDado. Caso encontre algum, o nome do serviço de nomes associado a esse objeto é pego e verifica-se sua existência ou não no serviço de nomes. Caso exista, então é atribuído o estado de resolvido a esse MetaDado. Caso não se encontre nenhum objeto do serviço de diretório ou, o nome no serviço de nomes associado ao objeto encontrado no serviço de diretório não exista, então é atribuído o estado de não resolvido ao MetaDado.

Se for do tipo repositório de componentes, o configurador automático irá realizar uma chamada, via JMX, a uma função do EJBDeployer (veja seção 4.2.2.1).

Esse componente baixado como pré-requisito também pode ter os seus próprios pré-requisitos configurados no seu arquivo autoconf.xml. Caso ele o possua, na sua implantação esse arquivo será processado pelo módulo de configuração automática de forma que a implantação e configuração do primeiro componente fiquem em modo de espera até que o do segundo termine.

Se esse componente especificado no pré-requisito não estiver disponível na URL especificada ou se sua implantação não for finalizada com sucesso, então esse MetaDado será marcado como não resolvido.

Para cada um desses casos, sempre é devolvido um nome de um componente EJB no serviço de nomes. Esse nome é associado com o nome do gancho que cada elemento COMPONENT deve possuir especificado. A associação é registrada no *hash* do MetaDado COMPONENT e no MetaDado pai que o contém. Veja o exemplo da figura a seguir.

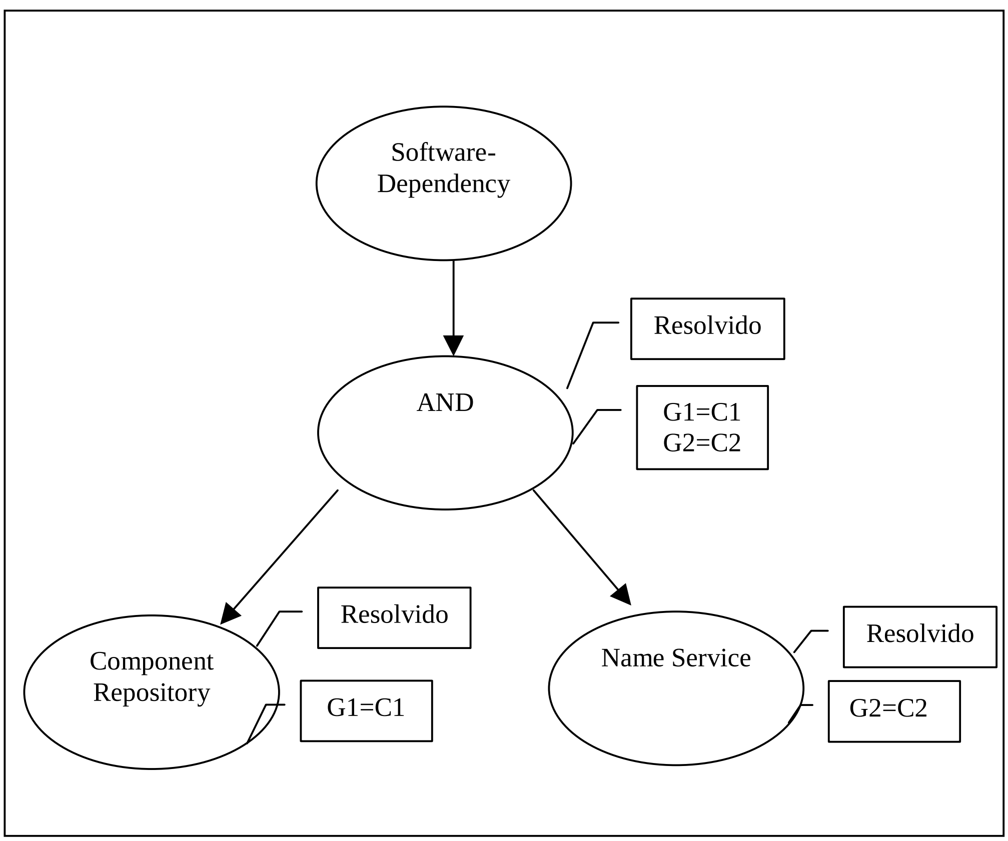


FIGURA 4.5 – EXEMPLO DE RESOLUÇÃO DE DEPENDÊNCIAS E GANCHOS

O elemento AND possui dois elementos COMPONENT, um do tipo a ser resolvido via repositório de componentes e outro via serviço de nomes. Os dois elementos possuem o estado de resolvido, assim o AND também está resolvido. O elemento COMPONENT-REPOSITORY resolve a dependência do gancho G1 com o componente EJB indicado pelo nome C1, já o elemento NAME-SERVICE resolve a dependência G2 com o componente indicado pelo nome C2. Assim, o elemento AND possui essas duas resoluções em sua tabela da *hash*.

Ao final do processo, o elemento SOFTWARE-DEPENDENCY, deve possuir todos os pré-requisitos (ganchos) do componente resolvidos; isso, considerando que o desenvolvedor não esqueceu ou omitiu nenhuma configuração de dependência. Se isso ocorrer, o nosso configurador automático não irá rejeitar a configuração obtida, contudo, na configuração dos ganchos (ejb-ref), se não houver a configuração para o gancho esquecido, ou omitido pelo desenvolvedor, nos outros arquivos (ejb-jar.xml ou jboss.xml), o JBoss não irá prosseguir com a implantação do componente.

O JBoss também foi alterado para que, no momento em que se estiver configurando os ganchos de um componente, seja consultada a estrutura MetaData que o configurador automático montou e configurou. Essa consulta é realizada através de uma chamada a uma função, via JMX , do módulo AutoConfMBean.

Capítulo 5

Avaliação de Desempenho

Neste capítulo apresentaremos um estudo do comportamento do JBoss com nossa implementação de configuração automática. A configuração automática foi implementada na forma de um MBean conectado ao JBoss. Esse MBean tem aproximadamente 26KB de um total de 27500KB que o JBoss possui em sua instalação padrão.

Fizemos dois tipos de experimentos, o primeiro focado na quantidade de componentes e o segundo focado no tamanho dos componentes. A fim de criar componentes de diferentes tamanhos para avaliar o desempenho de nossa implementação, criamos um programa que gera componentes EJB. De acordo com o parâmetro da linha de comando, o programa gera o código para um componente EJB contendo um dado número de métodos e variáveis. Devido a uma limitação do Java que impede que sejam dados saltos (*jumps*) no bytecode maiores que 64KB, ao tentarmos implantar um componente EJB com mais de 3440 métodos, o *proxy* gerado pelo JBoss lança o erro “*Code of a method longer than 65535 bytes*”, por isso os componentes EJB gerados possuem somente o primeiro método exportado. Na seção 5.3 analisaremos os impactos dessa decisão. Se todos os métodos criados fossem exportados, o tempo total de implantação do componente iria aumentar devido ao mapeamento dos métodos do componente EJB no contêiner do JBoss. Em média, o cadastro de 14 métodos leva 10 milésimos de segundo.

Esses experimentos foram realizados utilizando um AMD Duron 750MHz com 512MB de memória RAM rodando Linux RedHat 7.3 como repositório de componentes e um notebook COMPAQ Armada PII 400MHz com 196MB de memória RAM rodando Windows XP como cliente onde foi executado o JBoss e o Serviço de Diretórios. O Serviço de Nomes está incluso no JBoss. As duas máquinas estavam conectadas por uma rede Fast Ethernet de 100Mbps. A máquina virtual Java utilizada é a do JDK 1.3 da Sun.

Para que fossem realizados os experimentos, acrescentamos no nosso módulo de configuração automática (AutoConf) a ferramenta de log utilizada pelo JBoss para arquivarmos nossas medições de desempenho.

Um fato importante de se notar é que o arquivo de log indica uma resolução de milésimos de segundos, contudo fizemos um teste que indicou uma precisão de somente 10 milésimos de segundos. Esse teste consistiu em se fazer um laço onde eram impressos a hora do sistema com precisão de milésimos de segundo. Entretanto, o que percebemos é que esses valores da hora cresciam de 10 em 10 milésimos de segundo. Com isso temos uma tolerância em nossas medições de +/- 10 milésimos de segundos.

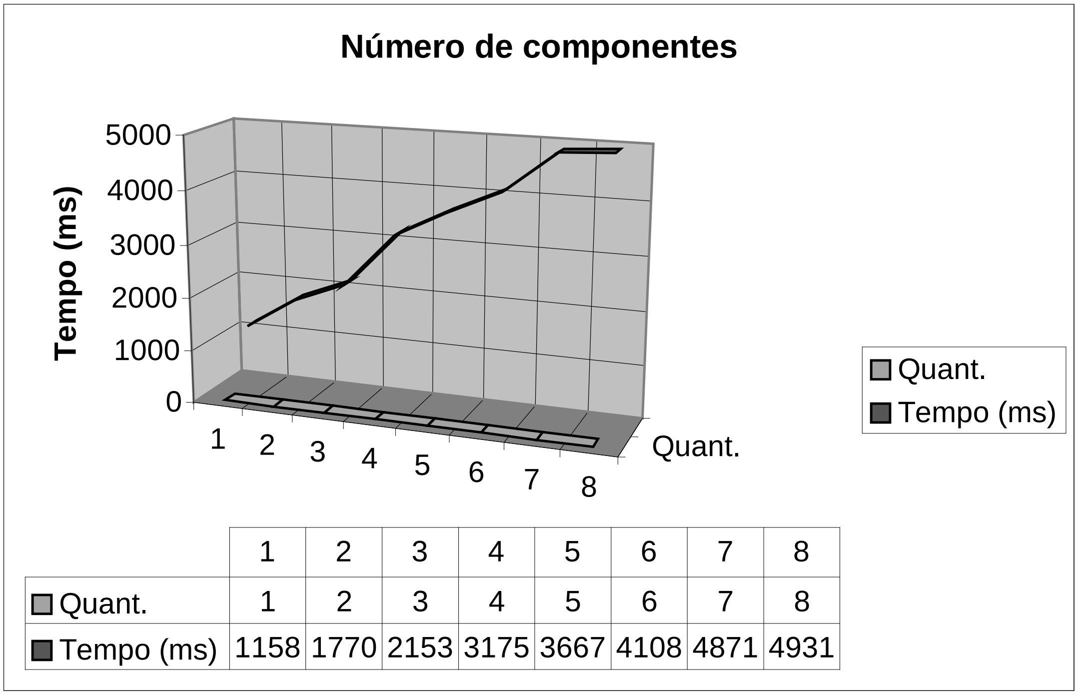
Para cada valor inserido nos gráficos utilizamos a média de cinco execuções do experimento. Para cada execução realizada, o JBoss foi reinicializado evitando que os dados fossem guardados no cache.

5.1. Implantando múltiplos componentes

Nesse experimento implantamos no JBoss um componente com o número de dependências variando de um a oito componentes, cada um possuindo 19.2Kbytes. Executamos esse teste para cada um dos três tipos de resolução de dependência que o nosso trabalho possui: resolução via Repositório de Componentes, via Serviço de Nomes e via Serviço de Diretórios.

5.1.1. Repositório de Componentes

A figura 5.1 mostra o tempo utilizado para se instalar um componente com dependências resolvendo-as via repositório de componentes.



	1	2	3	4	5	6	7	8
Quant.	1	2	3	4	5	6	7	8
Tempo (ms)	1158	1770	2153	3175	3667	4108	4871	4931

FIGURA 5.1 – IMPLANTAÇÃO COM RESOLUÇÃO DE DEPENDÊNCIAS VIA REPOSITÓRIO DE COMPONENTES

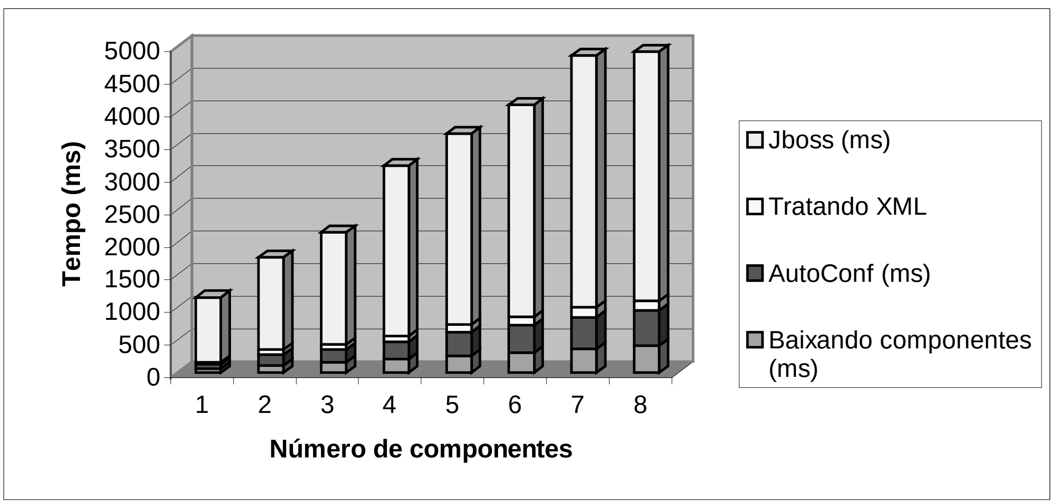


FIGURA 5.2 – TEMPO ABSOLUTO DE IMPLANTAÇÃO COM RESOLUÇÃO DE DEPENDÊNCIA VIA REPOSITÓRIO DE COMPONENTES POR NÚMERO DE COMPONENTES

Já na figura 5.2, temos os tempos absolutos em cada passo do processo. Podemos observar que à medida que o número de componentes aumenta, os tempos de tratamento do arquivo autoconf.xml, de processamentos do configurador automático e o tempo de baixar componentes também aumentam. Contudo, em termos percentuais esse aumento não é percebido como vemos na figura a seguir.

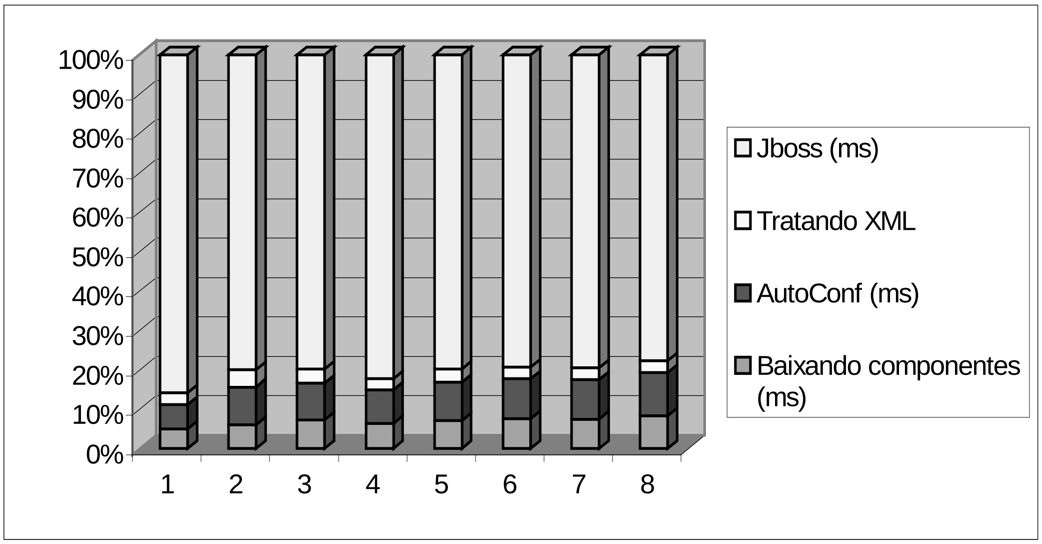


FIGURA 5.3 – TEMPO PERCENTUAL DE IMPLANTAÇÃO COM RESOLUÇÃO DE DEPENDÊNCIA VIA REPOSITÓRIO DE COMPONENTES

A tabela a seguir mostra mais detalhadamente como o tempo é gasto na implantação de um componente de 19,2Kbytes.

Ação	Tempo (ms)	% do total	Desvio padrão
Baixar o componente	51	9	5
Tratamento do XML	2	0	4
Tratamento das dependências	3	1	5
Implantação do componente pelo JBoss	484	90	227
Total	540	100	

Tabela 5.1 – Tempo de implantação de um componente de 19,2KBytes

A tabela foi montada a partir da média de todas as implantações do componente de 19,2Kbytes, ou seja, a partir de média de 180 implantações.

Os valores de Tratamento do XML e Tratamento das dependências deveriam ser zero (0), pois esses componentes não possuíam o arquivo autoconf.xml, ou seja, esses componentes não possuíam dependências. Contudo, isso é aceitável devido à tolerância de +/- 10 milésimos de segundo. Já o desvio padrão observado na implantação do componente é devido ao comportamento do JBoss não ser totalmente controlável, como por exemplo, o coletor de lixo do Java. Trechos de código que levavam 20 milésimos de segundos às vezes eram executados em 80 até 100 milésimos de segundo.

Podemos também comparar esses resultados como os resultados obtidos no *Serviço de Configuração Automática do 2K* (ver seção 2.4). No sistema 2K, devido à sua linguagem de programação ser o C++ e, portanto bem mais rápida que a nossa, vemos que a implantação dos seus componentes representava somente 5% do total na instalação, como vemos na tabela 5.2, ao contrário da nossa em que a implantação representa 90%.

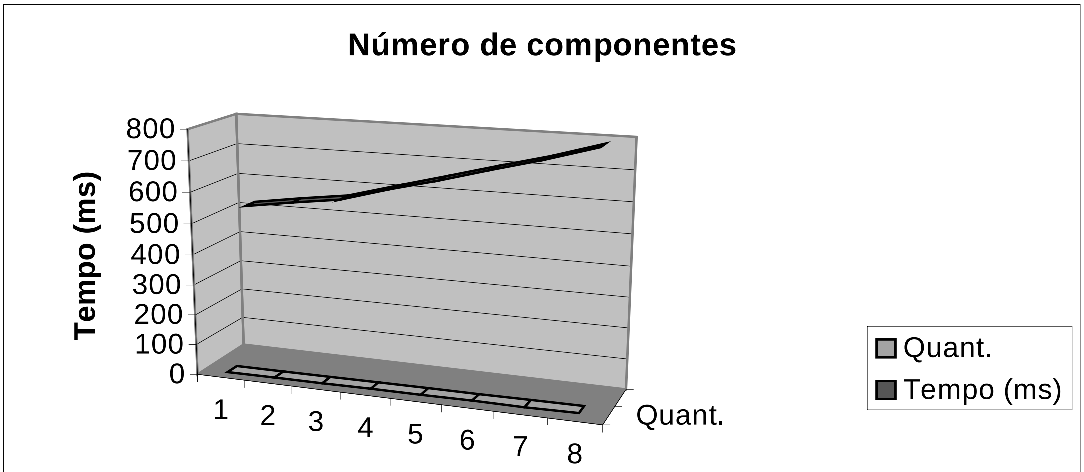
Ação	Tempo (ms)	% do total	Desvio padrão
Baixar o componente	8	33	0
Operações do Configurador automático	11	46	0,7
Implantação do componente	5	21	0
Total	24	100	0,7

Tabela 5.2 – Tempo de implantação de um componente de 19,2Kbytes no sistema 2K

Nesses testes do sistema 2K foram utilizadas duas máquinas Sparc Ultra-60 rodando Solaris 7, conectados por uma rede Fast Ethernet de 100Mbps. Os componentes utilizados também possuíam 19.2Kbytes de tamanho.

5.1.2. Serviço de Nomes

A figura a seguir mostra o tempo utilizado para se instalar um componente com dependências resolvendo-as via serviço de nomes.



	1	2	3	4	5	6	7	8
Quant.	1	2	3	4	5	6	7	8
Tempo (ms)	519	543	565	609	649	693	735	784

FIGURA 5.4 – IMPLANTAÇÃO COM RESOLUÇÃO DE DEPENDÊNCIAS VIA SERVIÇO DE NOMES POR NÚMERO DE COMPONENTES

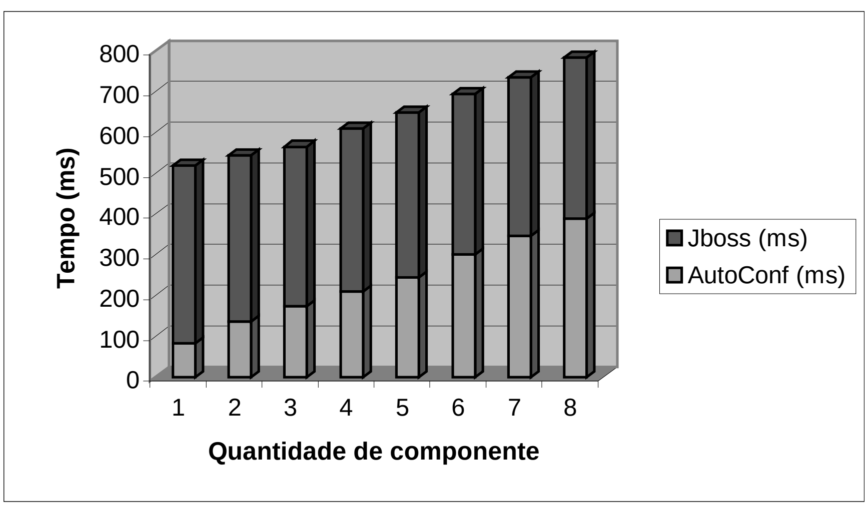


FIGURA 5.5 – TEMPO ABSOLUTO DE IMPLANTAÇÃO COM RESOLUÇÃO DE DEPENDÊNCIAS VIA SERVIÇO DE NOMES

Na figura 5.5 vemos, em tempo absoluto, os tempos gastos pelo JBoss e pelo configurador automático.

Temos na figura a seguir em termos percentuais em relação ao tempo total.

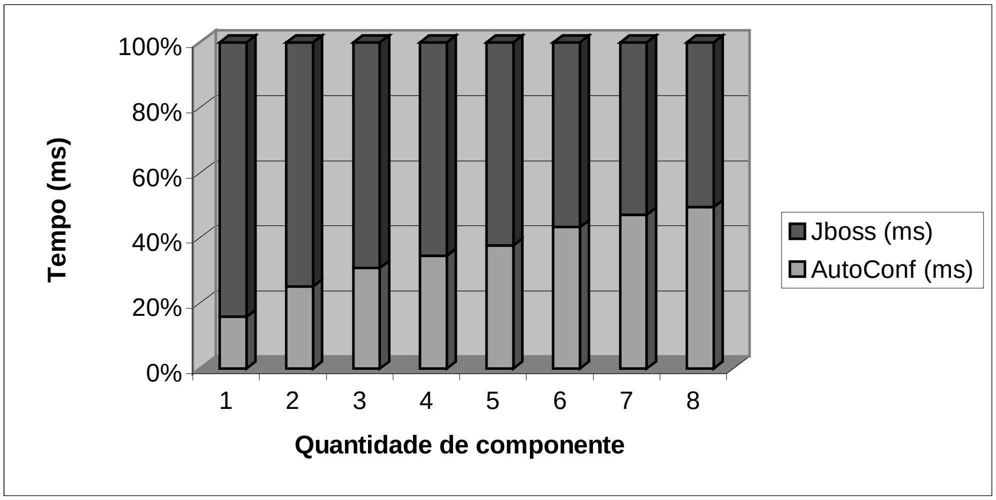
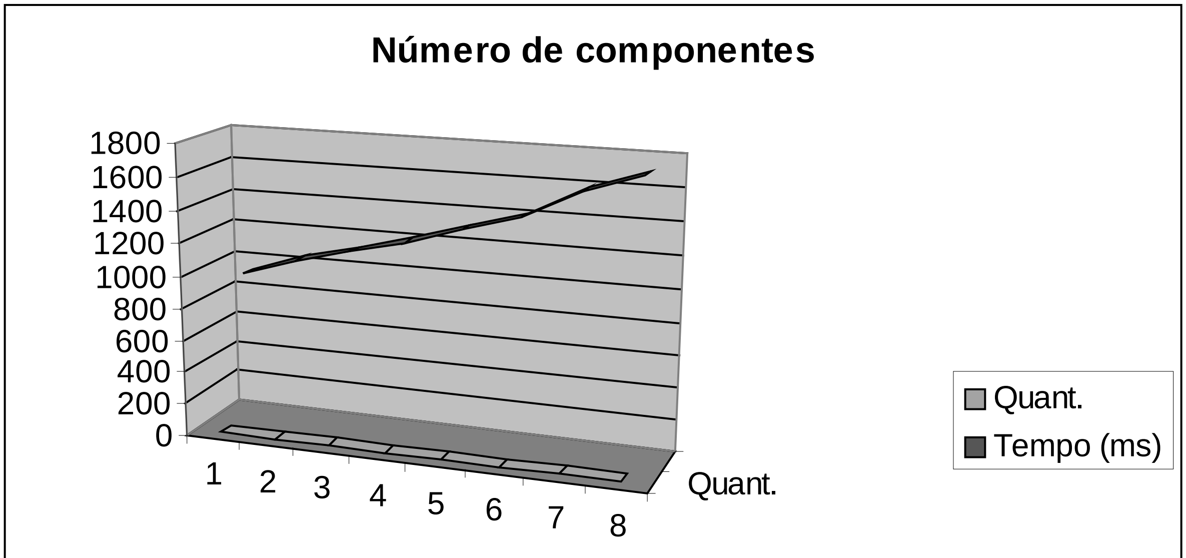


FIGURA 5.6 – TEMPO PERCENTUAL NA IMPLANTAÇÃO COM RESOLUÇÃO DE DEPENDÊNCIAS VIA SERVIÇO DE NOMES

É interessante notar que não existe o tempo de transferência do arquivo devido à resolução da dependência ser via serviço de nomes. Assim, como podemos ver, o tempo de execução do configurador automático tende a aumentar enquanto o tempo de implantação do componente no JBoss é mais ou menos constante.

5.1.3. Serviço de Diretórios

A figura a seguir mostra o tempo utilizado para se instalar um componente com dependências resolvendo-as via serviço de diretórios.



	1	2	3	4	5	6	7	8
Quant.	1	2	3	4	5	6	7	8
Tempo (ms)	922	1034	1118	1206	1314	1414	1586	1703

FIGURA 5.7 – IMPLANTAÇÃO COM RESOLUÇÃO DE DEPENDÊNCIAS VIA SERVIÇO DE DIRETÓRIO POR NÚMERO DE COMPONENTES

Na figura a seguir vemos, em tempo absoluto, os tempos gastos pelo JBoss e pelo configurador automático.

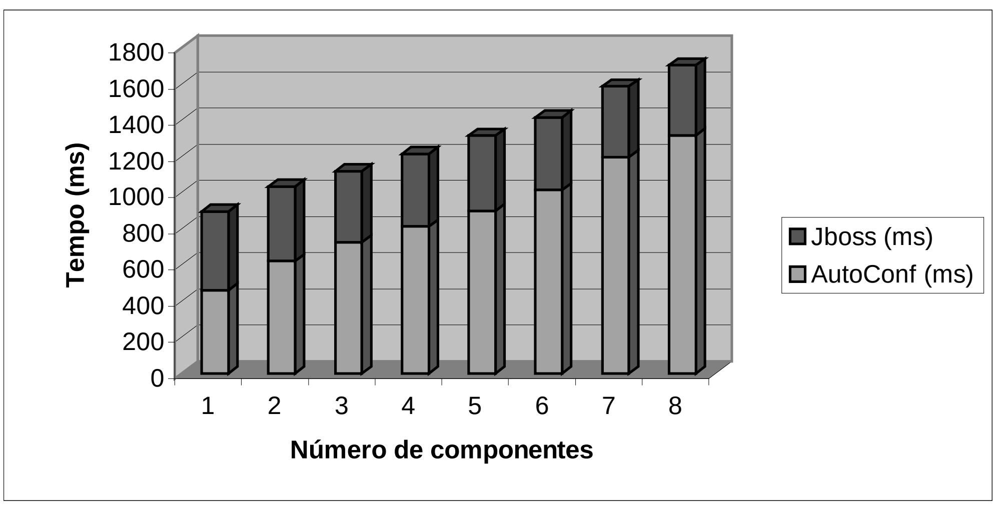


FIGURA 5.8 – TEMPO ABSOLUTO NA IMPLANTAÇÃO COM RESOLUÇÃO DE DEPENDÊNCIAS VIA SERVIÇO DE DIRETÓRIOS

Temos na figura a seguir em termos percentuais em relação ao tempo total.

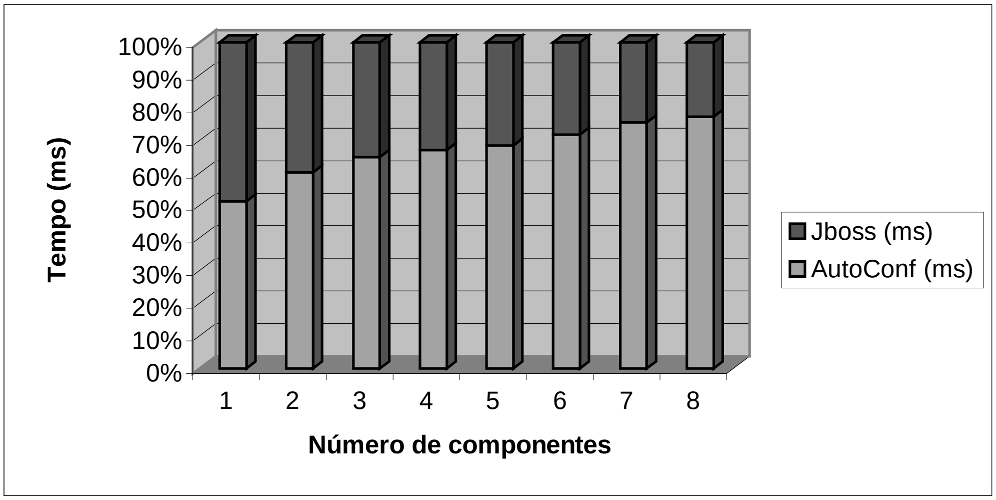


FIGURA 5.9 – TEMPO PERCENTUAL DE IMPLANTAÇÃO COM RESOLUÇÃO DE DEPENDÊNCIA VIA SERVIÇO DE DIRETÓRIOS

Da mesma forma que na resolução via serviço de nomes, na resolução via serviço de diretórios o componente já está implantado e, portanto, não é necessário realizar a transferência das dependências. Vemos que o tempo de processamento do JBoss é praticamente constante e que a participação do tempo de processamento do configurador é cada vez maior. Note que com somente uma dependência, o tempo de execução do configurador automático é quase 50% do total, mas ao aumentar o número dependências, esse tempo não é dobrado. Isso se deve ao fato de que o primeiro acesso ao serviço de diretório é muito mais demorado, pois é quando o JNDI inicializa a conexão com o servidor LDAP.

5.1.4. Comparação entre as formas de resolução

Nas figuras 5.10 e 5.11 a seguir temos uma comparação entre os tempos de processamento que o AutoConf utiliza na implantação dos componentes e o tempo total de implantação dos componentes para cada um dos tipos de resolução de dependência, respectivamente.

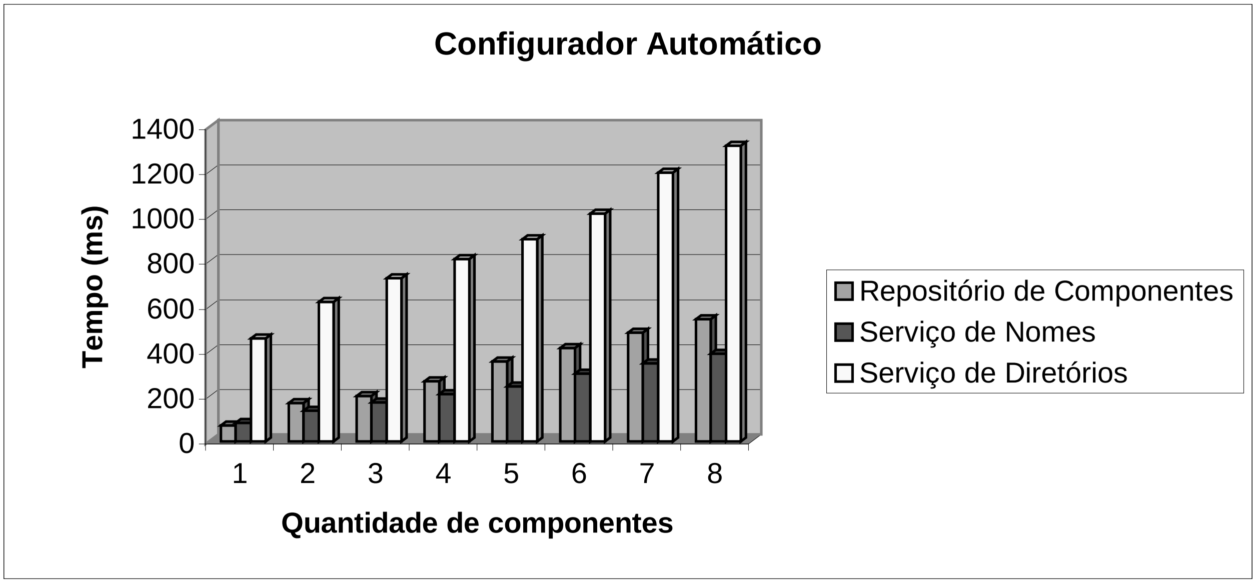


FIGURA 5.10 – COMPARAÇÃO DO TEMPO DO CONFIGURADOR AUTOMÁTICO POR NÚMERO DE COMPONENTES

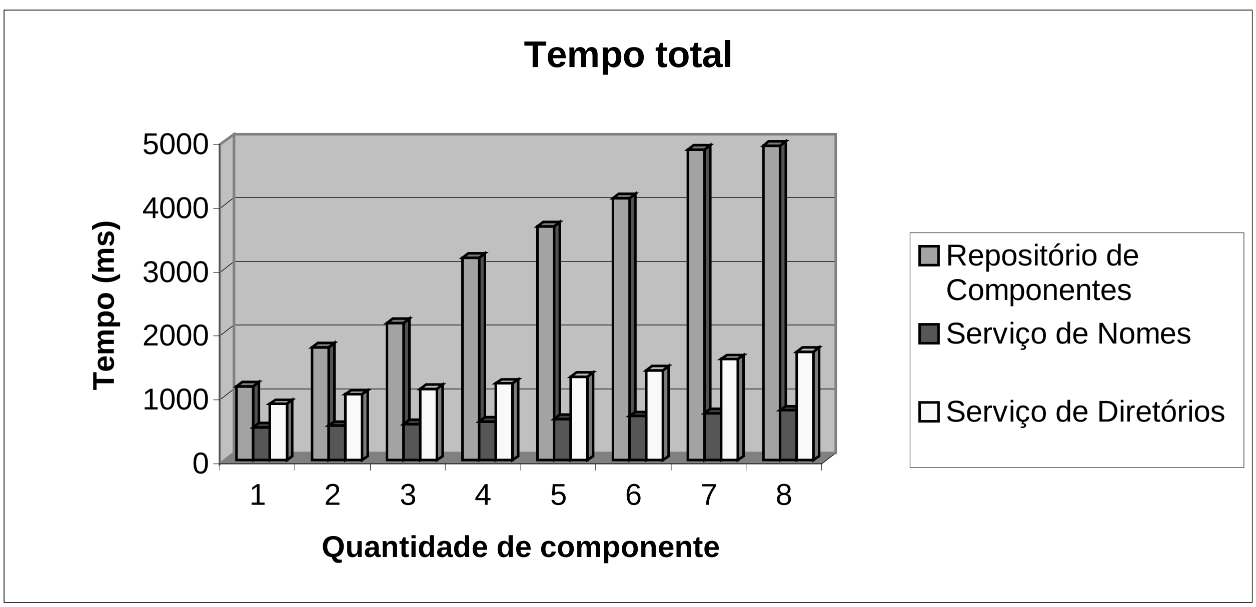


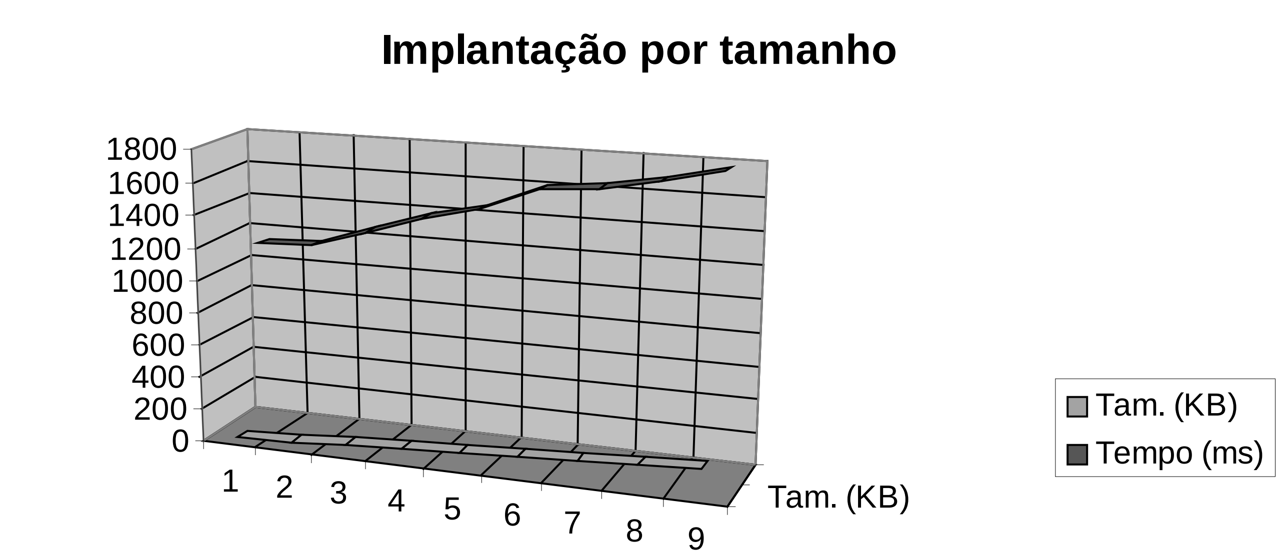
FIGURA 5.11 – COMPARAÇÃO DO TEMPO TOTAL DE IMPLANTAÇÃO POR NÚMERO DE COMPONENTES

5.2. Implantando componentes de diversos tamanhos

Nesse experimento implantamos no JBoss um componente com uma dependência variando, agora, o seu tamanho de 4,5 até 161 Kbytes. Também executamos esse experimento para cada um dos três tipos de resolução de dependência.

5.2.1. Repositório de Componentes

Temos na figura 5.12 o tempo utilizado para se instalar o componente via repositório de componentes.



	1	2	3	4	5	6	7	8	9
Tam. (KB)	4,5	19,2	44,2	64,6	84	104	123	141	161
Tempo (ms)	1146	1158	1266	1380	1464	1606	1635	1698	1782

FIGURA 5.12 – IMPLANTAÇÃO COM RESOLUÇÃO VIA REPOSITÓRIO DE COMPONENTES POR TAMANHO DE COMPONENTE

Temos a seguir um gráfico que mostra os tempos absolutos para cada passo do processo.

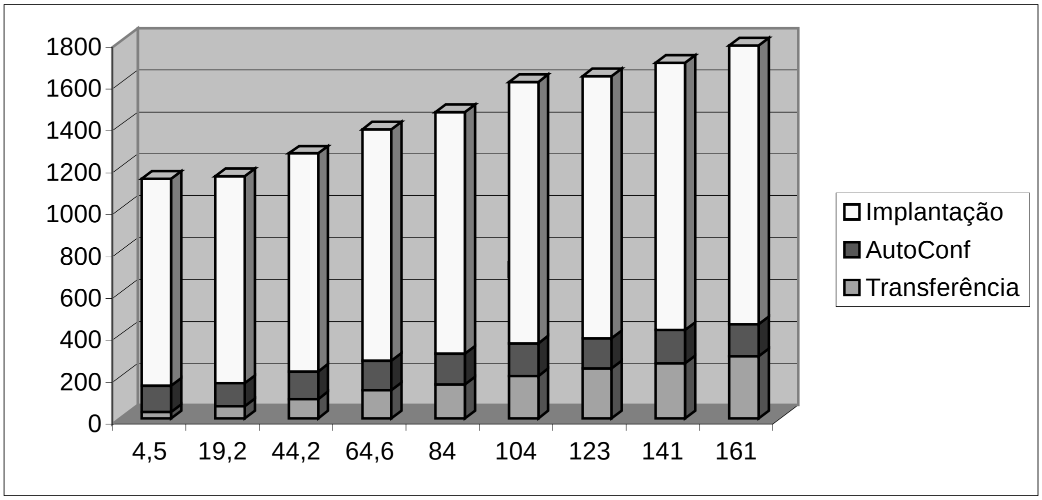


FIGURA 5.13 – TEMPO ABSOLUTO DE IMPLANTAÇÃO COM RESOLUÇÃO VIA REPOSITÓRIO DE COMPONENTE POR TAMANHO DE COMPONENTE

Note que o tempo de processamento do configurador automático não varia e que com o aumento do tamanho do componente, a participação percentual do configurador automático tende a decrescer conforme a figura a seguir.

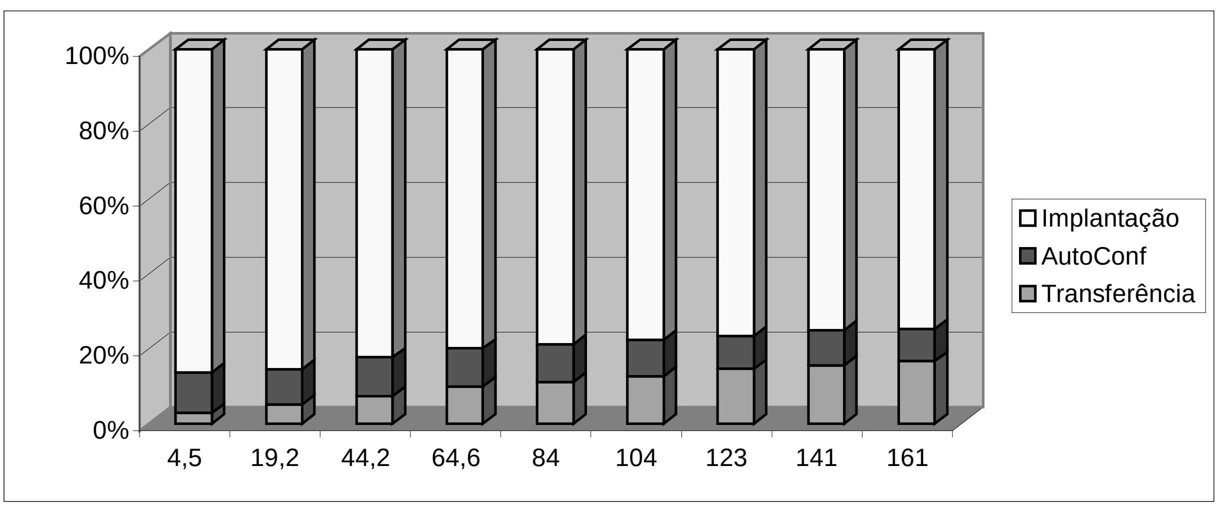


FIGURA 5.14 – TEMPO PERCENTUAL DE IMPLANTAÇÃO COM RESOLUÇÃO DE DEPENDÊNCIA VIA REPOSITÓRIO DE COMPONENTES.

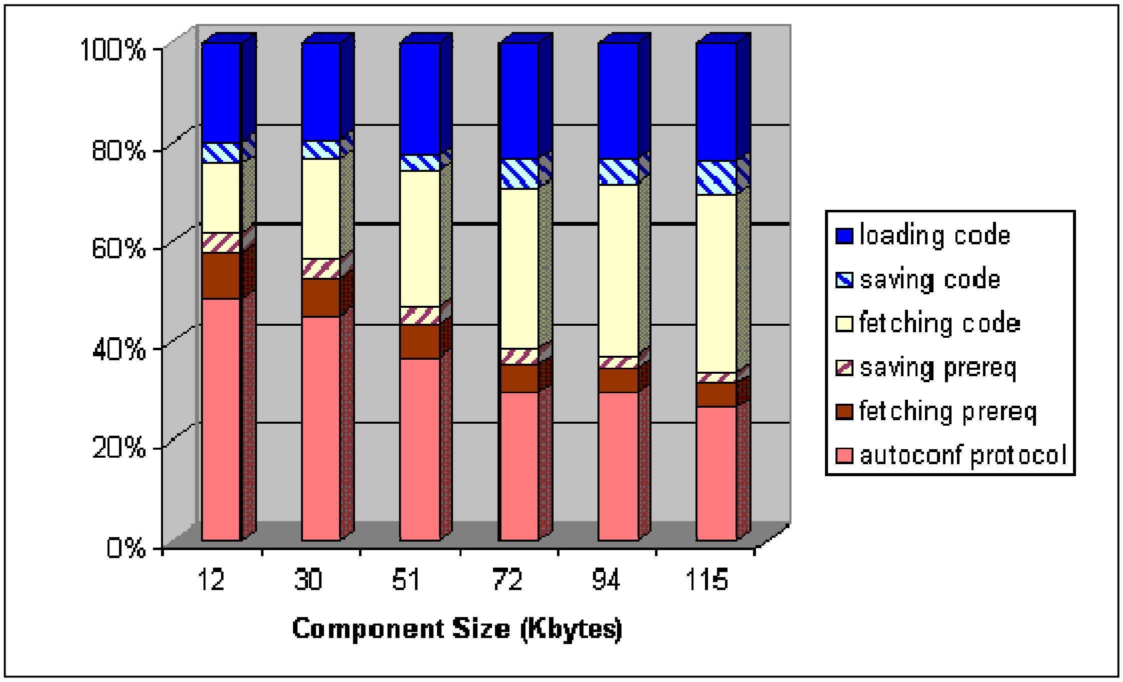


FIGURA 5.15 – TEMPO PERCENTUAL DE CARGA DE COMPONENTE DE DIFERENTES TAMANHOS NO SISTEMA 2K.

Podemos perceber que no nosso trabalho, a participação percentual do configurador automático no tempo total de implantação tende a diminuir ao aumentarmos o tamanho do componente implantado. Essa característica é semelhante ao encontrado no trabalho sistema 2K, contudo essa participação tende a diminuir com menor velocidade do que no configurador automático do 2K, pois a participação percentual do nosso configurador automático no tempo total de implantação de um componente já é pequena desde o início, ao contrário do que ocorre nos experimentos realizados com o 2K.

5.2.2. Serviço de Nomes

Temos na figura abaixo o tempo utilizado para instalar o componente com dependências a serem resolvidas via serviço de nomes.

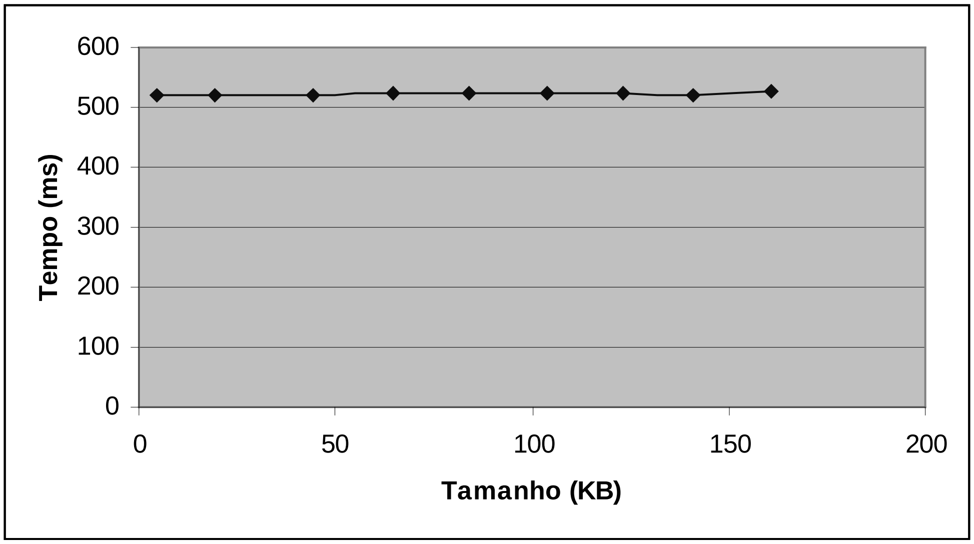


FIGURA 5.15 –IMPLANTAÇÃO COM RESOLUÇÃO VIA SERVIÇO DE NOMES POR TAMANHO DE COMPONENTES

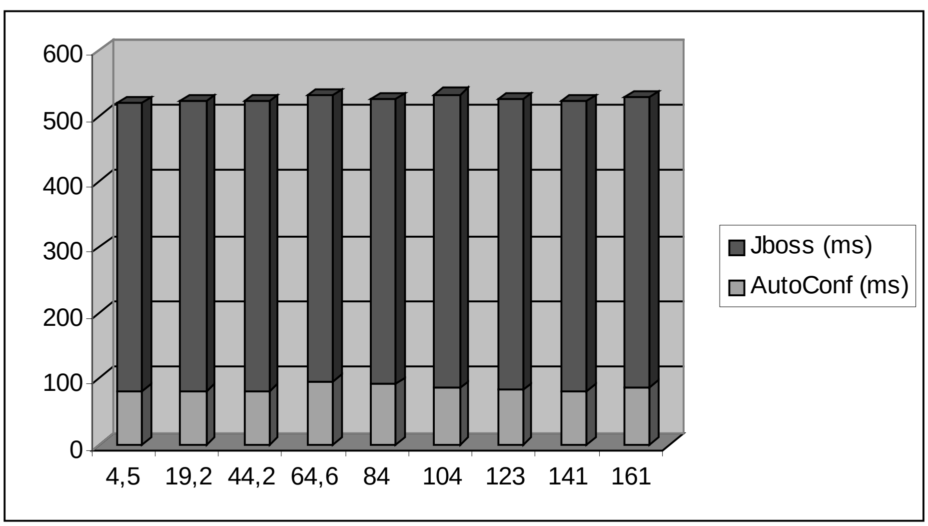


FIGURA 5.16 – TEMPO ABSOLUTO DE IMPLANTAÇÃO COM RESOLUÇÃO DE DEPENDÊNCIA VIA SERVIÇO DE NOMES

Podemos observar que o tempo de implantação é praticamente constante, pois os componentes dos quais se depende já estão implantados não influenciando no tempo de configuração do componente que está sendo instalado. Temos uma visão mais detalhada da participação do AutoConf na figura 5.16.

5.2.3. Serviço de Diretórios

Temos na figura abaixo o tempo utilizado para instalar o componente com dependências a serem resolvidas via serviço de diretórios.

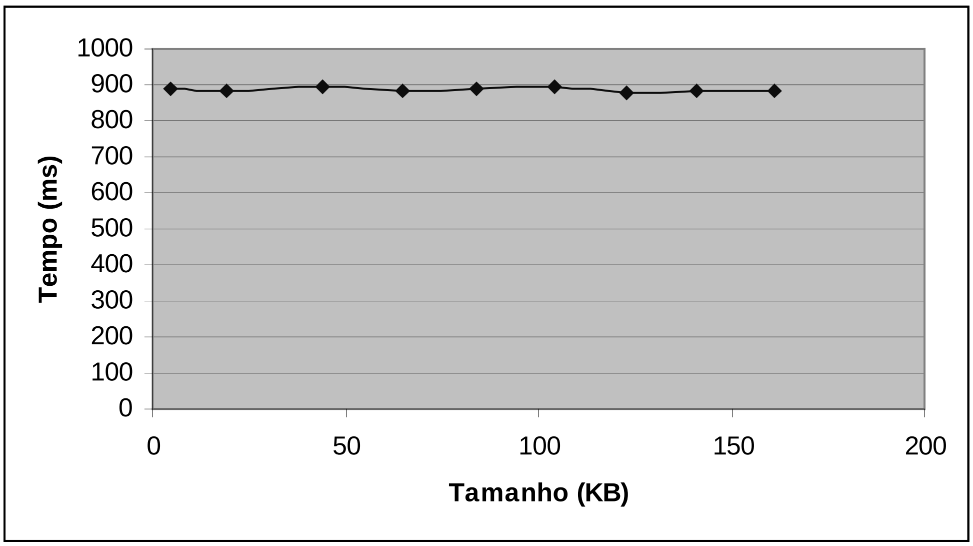


FIGURA 5.17 – IMPLANTAÇÃO COM RESOLUÇÃO VIA SERVIÇO DE DIRETÓRIO POR TAMANHO DE COMPONENTES

Aqui também podemos observar que o tempo de implantação é praticamente constante, pois os componentes dos quais se depende já estão implantados não influenciando no tempo de configuração do componente que está sendo instalado. Temos uma visão mais detalhada da participação do configurador automático na figura a seguir.

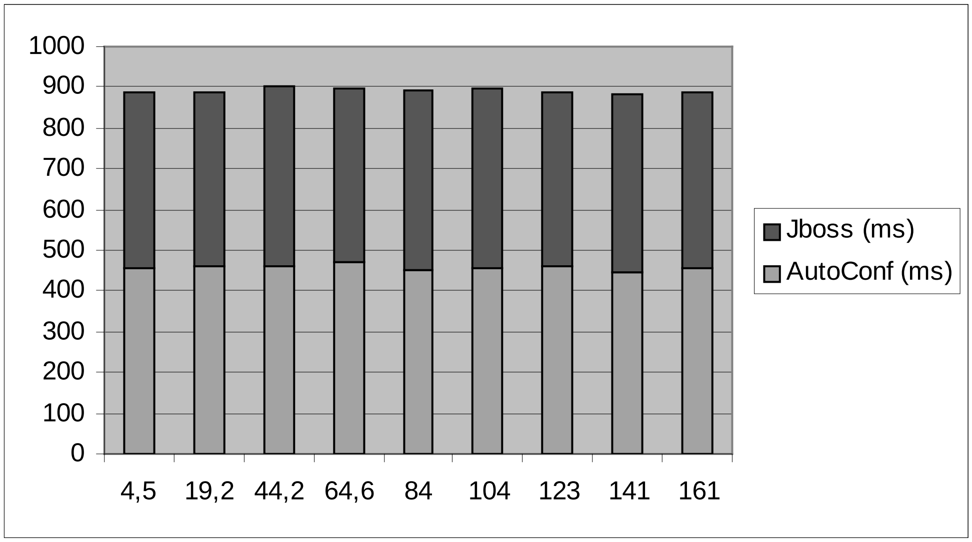


FIGURA 5.18 – TEMPO ABSOLUTO DE IMPLANTAÇÃO COM RESOLUÇÃO VIA SERVIÇO DE DIRETÓRIO

Nesse caso podemos notar que o tempo de processamento do AutoConf no processamento total é praticamente 50% do total.

5.2.4. Comparação entre as formas de resolução

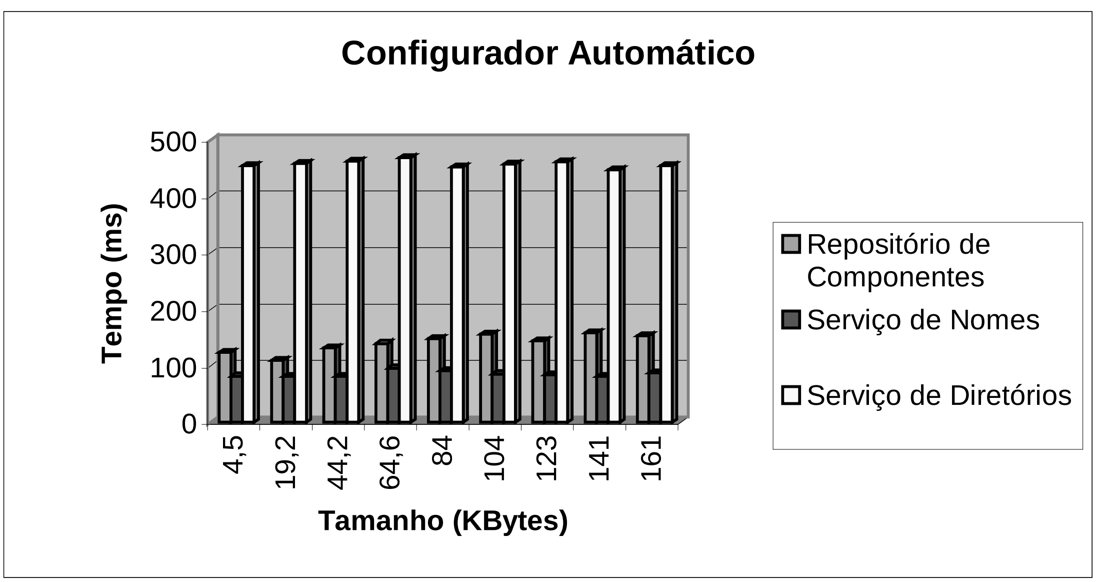


FIGURA 5.19 – COMPARAÇÃO DOS TEMPOS DO CONFIGURADOR AUTOMÁTICO POR TAMANHO DE COMPONENTES

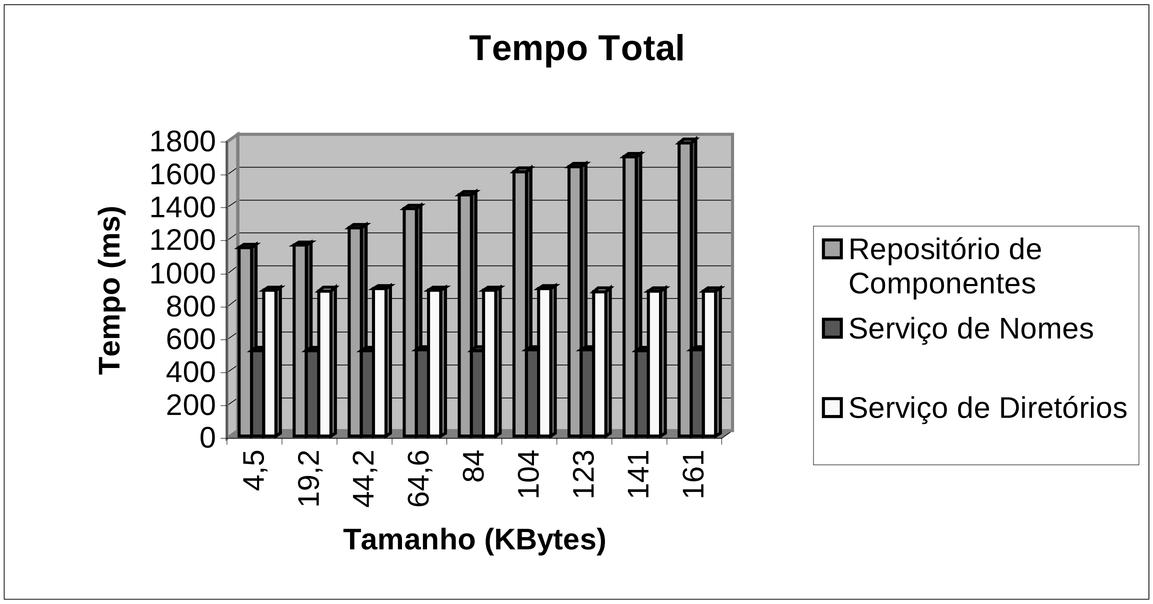


FIGURA 5.20 – COMPARAÇÃO DOS TEMPOS DE IMPLANTAÇÃO POR TAMANHO DOS COMPONENTES

Nas figuras 5.19 e 5.20 temos uma comparação entre os tempos de processamento que o AutoConf utiliza na implantação dos componentes e o tempo total de implantação dos componentes para cada um dos tipos de resolução de dependências, respectivamente. Podemos ver que somente a resolução via repositório de componentes sofre alteração devido ao tamanho do componente implantado, as outras duas formas, como era de se esperar, não sofrem alteração alguma. Entretanto, observando a figura 5.19, podemos perceber que o configurador automático não sofre nenhuma alteração significativa, o que também era de se esperar uma vez que ele não sofre nenhuma ação com relação ao tamanho do componente. As únicas atividades que sofrem alteração conforme o tamanho do componente são o ato de baixar o arquivo e sua instalação no JBoss.

5.3. Análise dos dados

Como era de se esperar, os tempos de instalação de um componente com dependências a serem resolvidos via repositório de componentes são os maiores obtidos, uma vez que na resolução de dependências via serviço de nomes ou de diretórios, os componentes dos quais se depende já devem estar implantados em uma máquina sendo necessário somente se obter a referência para esses componentes e ligá-los aos respectivos ganchos. Notamos também que na resolução via serviço de nomes e de diretórios, o percentual de participação do configurador automático no tempo total de processamento é maior, pois todo o processamento de resolução é feito pelo configurador automático. Já na resolução via repositório de componentes, o configurador automático simplesmente obtém a localização do componente e pede para o JBoss implantá-lo.

Caso nosso gerador de componentes EJB criasse componentes com todos os métodos exportados, teríamos os tempos de implantação do componente EJB pelo JBoss bem maiores, o que significaria uma menor participação percentual do configurador automático em todos os dados medidos.

No geral, podemos concluir que o nosso módulo de configuração automática não traz grandes prejuízos na implantação de um componente, somente o benefício de se poder especificar as dependências de um componente.

Capítulo 6

Trabalhos Futuros

Nesse trabalho não tratamos as dependências de hardware que são de grande importância na configuração de um sistema computacional. Não é desejável que um software simplesmente não funcione caso não possua algum recurso disponível ou em quantidade suficiente. O tratamento das dependências de hardware se faz mais necessário ainda sabendo que temos uma enorme variedade de hardware disponível no mercado, que continuam a surgir novos hardwares e que estamos numa época em que a computação móvel e ubíqua está sendo cada vez mais estudada e requisitada.

Outro item importante é a reconfiguração dinâmica, que possibilita que o sistema se reconfigure automaticamente em tempo de execução caso sofra algum dano em alguma de suas dependências. A reconfiguração dinâmica aumenta a confiabilidade do sistema uma vez que, caso ocorra algum problema, ele se reconfigura ao invés de lançar erros ao usuário.

Podemos também futuramente melhorar a utilização do LDAP. Não foram esgotadas todas as possibilidades de utilização do LDAP uma vez que podemos criar nele objetos que guardem qualquer tipo de informação. Uma primeira coisa que poderia ser desenvolvida seria a sua integração com o serviço de nomes como no trabalho desenvolvido por Gustavo Scalco Isquierdo [26]. Isso possibilitaria utilizar somente o LDAP e não mais o LDAP em conjunto com o serviço de nomes como fazemos atualmente.

O controle de versão dos componentes e a verificação dos tipos nos ganchos também auxiliariam na implementação de softwares mais confiáveis e, com certeza, o quesito segurança na transferência do código deve ser tratado nos futuros trabalhos para que se possa aplicar esse tipo de configuração automática via Internet.

Capítulo 7

Conclusão

Com o aumento da complexidade, tamanho e distribuição das aplicações atuais, sua correta configuração e administração é cada vez mais difícil e crucial. Nesse trabalho não somente mostramos a viabilidade do arcabouço apresentado por Kon numa arquitetura padrão de servidores baseado em componentes, o *Enterprise JavaBeans*, mas também apresentamos extensões que aumentaram a expressividade das especificações de dependência e o poder do Configurador Automático.

Referências

[1] Wendy E. Mackay. Users and Customizable software: a co-adaptive phenomenon. Doctor's thesis, Alfred P. Sloan School of Management, Massachusetts Institute of Technology, May 1990.

[2] Desmond f. D'Souza, Alan Cameron Wills. *Objects, Components, And Frameworks with UML. The Catalysis Approach*. Addison Wesley. 1999.

[3] Clemens Szyperski. *Component Software Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[4] Fabio Kon, Roy H. Camplbell, M. Dennis Michunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2K: A Distributed Operating System for Heterogeneous Environments. Technical Report UIUCDCS-R-99-2132, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1999.

[5] Fabio Kon, D. Carvalho, and R. Campbell. Automatic Configuration in the 2K Operation System. Proc. ECOOP'99 Workshop Object Orientation and Operation Systems. Chemnitzer Informatik-Berichte, Germany, June 1999, pp.10-14.

[6] Fabio Kon. Automatic Configuration of Component-Based Distributed Systems. PhD Thesis, University of Illinois at Urbana-Champaign, 2000

[7] Fabio Kon, R.H. Campbel, K. Nahrstedt. Using dynamic configuration to manage a scalable multimedia distribution system. Computer Communication 2001

[8] Ioana Sora, Franl Matthijs. Automatic Composition of Software Systems from components with anonymous dependencies. DistriNet labs, Department of Computer Science, K.U. Leuven, Belgium.

[9] Glêdson Elias da Silveira, Silvio Lemos Meira. CoDelivery: An Environment for Distribution of Reusable Components. TOOLS Europe. St. Malo, France. June, 2000.

[10] Noriko Amano and Takuo Watanabe. A Procedural Model of Dynamic Adaptability and its Description Language. *Proc. IWPSE'98*, pp. 103-107, April 1998.

[11] Noriko Amano and Takuo Watanabe. LEAD: Linguistic Approach to Dynamic Adaptability for Practical Applications. *Proc. IFIP Conf. System Implementation 2000,* Chapman & Hall, pp. 277-290, February 1998.

[12] Noriki Amano and Takuo Watanabe. LEAD++: Na Object-Oriented Reflective Language for Dynamically Adaptable Software Model. *IEICE Trans,* E82-A(6):1009-1016, June 1999.

[13] Noriki Amano and Takuo Watanabe. An Approach for Constructing Dynamically Adaptable Component-based Software Systems using LEAD++. *OOPSLA'99 Workshop on Reflection and Software Engineering.*

[14] Myrthes Cavalcante de Aguiar. Arquiteturas de Componentização de Servidores como Integradoras de Bancos de Dados Distribuídos. Instituto de Matemática e Estatística da Universidade de São Paulo, 2002.

[15] The Debian Project. Debian linux operating system home page. http://www.debian.org, 2003. Acessado em fevereiro de 2003.

[16] Sun Microsystems. Jini [tm] Network Technology. http://wwws.sun.com/software/jini/, 2002. Acessado em fevereiro de 2003.

[17] World Wide Web Consortium. Extensible Markup Language (XML) 1.0. http://www.w3.org/tr/1998/REC-xml-19980210, 1998. Acessado em fevereiro de 2003.

[18] Sun Microsystems. Enterprise JavaBeans Specification. Sun Microsystem, Version 2.0 Final Release, August 14, 2001.

[19] Sun Microsystems. Java Naming and Directory Interface (JNDI). http://java.sun.com/products/jndi/, 2002. Acessado em fevereiro de 2003.

[20] Michi Henning, Steve Vinoshi. *Advanced CORBA Programming with C++*. Addison Wesley Professional Computing Series. February, 2000.

[21] Tim Hower, Mark Smith. *LDAP Programming Directory – Enabled Applications with Lightweight Directory Acess Protocol*. Macmillan Technical Publishing, Indianopolis, IN, 1997.

[22] ITU - T. Information Technology – Open Systems Interconnection – The Directory – Recomendations X.500-X.521 ITU – International Telecommunication Union, 1993.

[23] OpenLDAP Foundation. OpenLDAP. http://www.openldap.org, 2003. Acessado em fevereiro de 2003.

[24] Sun Microsystems. Iplanet Server Products. http://wwws.sun.com/software, 2003. Acessado em fevereiro de 2003.

[25] Roberto Carlos dos Santos Pacheco, Vinícius Medina Kern. Uma ontologia comum para a integração de bases de informação e conhecimento sobre ciência e tecnologia. *Ci. Inf.*, Brasília, v.30, n.3, p. 56-63, set./dez. 2001.

[26] Gustavo Scalco Isquierdo. Integração do Serviço de Diretório LDAP com o serviço de Nomes CORBA. Tese de mestrado. Instituto de Matemática e Estatística da Universidade de São Paulo. Junho 2001.

[27] Marc Fleury, Juha Lindfors and The JBoss Group. *JMX: Managing J2EE with Java Management Extensions.* Que/SAMS, 2002.

[28] JBoss Group. Jboss Home Page. http://www.jboss.org, 2002. Acessado em fevereiro de 2003.

[29] Object Management Group (OMG). CORBA Component Model. Object Management Group, Version 3.0, June 2002.

[30] Microsoft Microsystems. Microsoft Transaction Server. At http://www.microsoft.com/com/tech/MTS.asp, 1999. Acessado em fevereiro de 2003.