MOISÉS LIMA DUTRA

COMPOSIÇÃO DINÂMICA DE APLICAÇÕES EM AMBIENTES DE EXECUÇÃO SOB DEMANDA

FLORIANÓPOLIS
2005

UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO
EM ENGENHARIA ELÉTRICA

COMPOSIÇÃO DINÂMICA DE APLICAÇÕES EM AMBIENTES DE EXECUÇÃO SOB DEMANDA

Dissertação submetida à
Universidade Federal de Santa Catarina
como partes dos requisitos para a
obtenção do grau de Mestre em Engenharia Elétrica.

Moisés Lima Dutra

Florianópolis, Fevereiro de 2005.

COMPOSIÇÃO DINÂMICA DE APLICAÇÕES EM AMBIENTES DE EXECUÇÃO SOB DEMANDA

Moisés Lima Dutra

‘Esta Dissertação foi julgada adequada para obtenção do Título de Mestre em Engenharia Elétrica, Área de Concentração em *Sistemas Computacionais*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.’

_____

Prof. Ricardo José Rabelo, Dr.
Orientador

_____

Prof. Denizar Cruz Martins, Dr.
Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

_____

Prof. Ricardo José Rabelo, Dr.
Presidente

_____

Prof. Leandro Buss Becker, Dr.

_____

Prof. Joni da Silva Fraga, Dr.

_____

Prof. Frank Augusto Siqueira, Dr.

À minha familia.

Agradecimentos

- Ao Ricardo “Kadu” Rabelo, meu orientador, por me mostrar a importância de se ter “um pé na realidade e outro na viagem”;
- Ao Celson Lima, meu tio, por sua contribuição inicial a este trabalho;
- Aos amigos que fiz no Mestrado, entre 2002 e 2005;
- Ao pessoal do G-Sigma (Leandro, Baldo, Rui, Gesser, Fábio e Tiago), por toda a ajuda que me ofereceram durante a confecção deste.
- Ao Zé Eduardo, pela ajuda com a impressão e entrega das correções;

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários
Para a obtenção do grau de Mestre em Engenharia Elétrica.

COMPOSIÇÃO DINÂMICA DE APLICAÇÕES EM AMBIENTES DE EXECUÇÃO SOB DEMANDA

Moisés Lima Dutra

Fevereiro / 2005

Orientador: Ricardo José Rabelo, Dr.
Área de Concentração: Sistemas Computacionais.
Palavras-chave: Componentes, Provedores de Serviços de Aplicação, Arquitetura Peer-to-Peer, Flexibilidade Funcional, Execução Adaptável.
Número de Páginas: 155.

Resumo: Esta dissertação propõe uma abordagem para minimizar o problema da reduzida flexibilidade funcional verificada nos complexos sistemas industriais, que são comprados como pacotes ou módulos inteiros, geralmente caros, mesmo que não se adaptem completamente às necessidades das empresas. Esta abordagem baseia-se na composição dinâmica e inteligente de componentes de *software*. Esta composição irá ocorrer somente quando determinada funcionalidade for efetivamente necessitada e adaptada para o corrente ambiente de utilização em tempo de execução.

A composição é feita sob demanda, através de uma nova visão dos Provedores de Serviços de Aplicação – PSAs (*Application Service Providers - ASPs*) – e aproveitando-se algumas idéias da abordagem de Aplicações Compostas (*Composite Applications*). Um dos principais conceitos a serem introduzidos no modelo proposto é o de Federação de Provedores de Aplicação (FPA), uma entidade distribuída que reúne logicamente uma série de provedores de aplicação (componentes).

Um protótipo foi elaborado e testado. Além disso, procurou-se esboçar um pequeno modelo de negócios que englobasse alguns aspectos que poderão vir a ser úteis para quem desejar levar o conceito adiante em termos comerciais, na ótica de *software* como serviço.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the Requirements for the degree of Master in Electrical Engineering.

DYNAMIC COMPOSITION OF APPLICATIONS INSIDE ON DEMAND ENVIRONMENTS

Moisés Lima Dutra

February / 2005

Advisor: Ricardo José Rabelo, Dr.
Area of Concentration: Computing Systems.
Keywords: Components, Application Service Providers, Peer-to-Peer Architecture, Functional Flexibility, Adaptable Execution.
Number of Pages: 155.

Abstract: This work proposes an approach to minimize the reduced functional flexibility in the complex industrial systems, where they are bought as a whole package or module, quite expensive, even though they are not used at all or do not fit the enterprise's need completely. The approach is based on the idea of a dynamic and intelligent plugging of *software* components. This plugging will occur only when the components functionalities are effectively needed, adapted to the current computing environment in use.

The plugging is made on demand, applying a new perspective to the Application Service Providers (ASPs) and to the Composite Applications. One of the main concepts to be introduced in the proposed model is the Federation of Application Providers (FAP), a distributed cluster of application providers.

A prototype also has been built up in order to valuate the proposed model. Moreover, a draft of a business model was proposed, in order to help someone that intends to go further in the commercial context of software as a service.

Sumário

Lista de Figuras

Lista de Tabelas

Glossário

AOS – Arquitetura Orientada a Serviços

API – Application Programming Interface

B2B – Business-to-Business

BBS – Bulletin Board System

CCM – CORBA Component Model

CICS – Customer Information Control System

CIDL – Component Implementation Definition Language

COM – Component Object Model

CORBA – Common Object Request Broker Architecture

COTS – Commercial Off The Shelf

DBC – Desenvolvimento Baseado em Componentes

DCOM – Distributed Component Object Model

DLL – Dynamic Linked Library

EJB – Enterprise JavaBeans

ERP – Enterprise Resource Planning

EV – Empresa Virtual

FP – Function Points

FPA – Federação de Provedores de Aplicação

FS – Federação de Serviços

GUI – Graphical User Interface

GUID – Global Unique Identifier

GAC – Global Assembly Cache

HTML – Hypertext Markup Language

HTTP – Hypertext Transfer Protocol

IDE – Integrated Development Environment

IDL – Interface Definition Language

IIOP – Internet Inter-ORB Protocol

ISP – Internet Service Provider

IP – Internet Protocol

J2EE – Java 2, Enterprise Edition

JDBC – Java Database Connectivity

JMS – Java Message Service

JSP – Java Server Pages

LAN – Local Area Network

LOC – Lines of Code

MIDL – Microsoft Interface Definition Language

MIT – Massachusetts Institute of Technology

MOM – Message-Oriented Middleware

MSMQ – Microsoft Message Queue Server

MTS – Microsoft Transaction Server

NAT – Network Address Translation

OCL – Object Constraint Language

OMG – Object Management Group

ORB – Object Request Broker

OSD – Open Source Descriptor

P2P – Peer-to-Peer

PA – Provedor de Aplicação

PAC – Provedor de Aplicação Coordenador

PC – Personal Computer

POA – Portable Object Adapter

POO – Programação Orientada a Objetos

PSA – Provedor de Serviços de Aplicação

PSP – Plataforma de Suporte à Plugagem

RLC – Repositório Local de Componentes

RMI – Remote Method Invocation

RPC – Remote Procedure Call

SC^2 – Supply Chain Smart Co-ordination

SDK – Software Development Kit

Sebrae – Serviço Brasileiro de Apoio às Micro e Pequenas Empresas

SOAP – Simple Object Access Protocol

SQL – Structured Query Language

SVA – Serviço de Valor Agregado

TCP – Transmission Control Protocol

TI – Tecnologia da Informação

UDDI – Universal Description, Discovery and Integration

UML – Unified Modelling Language

URL – Uniform Resource Locator

USENET – Users Network

VPN – Virtual Private Network

WAN – Wide Area Network

Web – World Wide Web (WWW)

WfMC – Workflow Management Coalition

WSDL – Web Services Description Language

XML – Extensible Markup Language

XSL – Extensible Stylesheet Language

Capítulo 1

1. Introdução

Os modernos sistemas computacionais têm se notabilizado por sua crescente complexidade. Apesar de agregarem cada vez mais funcionalidade, estes acabam por requerer das empresas um verdadeiro "arsenal" técnico, financeiro e tecnológico para serem adquiridos e implantados. Segundo o Sebrae (2004), 98% das empresas formais brasileiras são pequenas e médias, logo, muitas das soluções que representariam a manutenção das suas competitividades se tornam inviáveis de ser adquiridas.

Uma das saídas encontradas para lidar com esta situação é o desenvolvimento de sistemas cada vez mais especializados. Essa customização se dá no âmbito de tarefas bem definidas, de maneira que adaptações deverão ser feitas para cada uma das tarefas empreendidas. Entretanto, apesar das muitas iniciativas internacionais em andamento e de alguns resultados já obtidos, os principais sistemas continuam aplicando o conceito de customização em um estágio que se pode considerar primário. Em termos gerais, o máximo que permitem é a criação de uma instância de um sistema maior – os chamados *My...* – usando um número mais reduzido de módulos, portanto menos custosa, mais rápida de ser implantada e mais adaptada funcionalmente a uma dada empresa. A questão de base é que, na prática, um módulo de um sistema ainda é desenvolvido com uma *granularidade*[1] elevada, ou seja, geralmente compra-se um módulo inteiro e não apenas instâncias ou partes dele. Isso significa que as empresas continuam arcando com uma série

[1] Granularidade diz respeito ao tamanho (como grãos) das unidades de código em determinado contexto e também ao nível de detalhe empregado nestas unidades (HyperDictionary, 2003). Quanto menor (ou mais fina) for a granularidade, mais simples e específica será a unidade de código.

de custos associados aos módulos quando, na prática, sabe-se que a maior parte delas (seus usuários) exploram somente uma pequena parte do conjunto das funcionalidades de cada módulo. Não necessariamente porque as desconhecem; mas porque simplesmente não as necessitam.

A pesquisa científica nessa área tem obtido bons avanços através de iniciativas como as Aplicações Compostas (*Composite Applications*) (Golden ORB, 2003), onde se destacam os serviços via Web e o desenvolvimento baseado em componentes. Utilizar uma aplicação composta significa utilizar diversas aplicações distintas – inclusive geograficamente distribuídas – que irão se comportar como se fossem módulos de uma aplicação maior, situada em uma camada lógica superior. A aplicação composta aproveita as funcionalidades destes sistemas agregados e coordena o processo de interação entre eles com o intuito de executar determinada tarefa ou transação.

Apesar do bom desempenho, as aplicações compostas ainda deixam a desejar em relação à obtenção de uma "flexibilidade funcional" mais ampla, onde a empresa só utilize aquilo de que precisa, somente quando for necessário e em postos de trabalho (*software* e *hardware*) não necessariamente fixos e homogêneos, e não necessariamente dentro da empresa. Em outras palavras, o que se busca aqui é atingir um paradigma de utilização de *software* do tipo "usar apenas o necessário, no momento necessário, no local necessário".

1.1 Motivação e Justificativa

Obter completa flexibilidade funcional significa construir e disponibilizar um *software*, não mais de forma monolítica, mas como um aglomerado de entidades menores independentes, logicamente integradas, e que no todo dotam o sistema de sua completa funcionalidade. Sistemas com essas características obtêm enormes ganhos de escalabilidade (capacidade do mesmo de adicionar ou retirar módulos – agregando, excluindo ou redefinindo funcionalidades – sem inteferir no seu funcionamento global). Essa é uma característica desejável quando se pensa, por exemplo, em atualizações de versão em sistemas de larga escala. Ou da ampliação destes de maneira a torná-los capazes de exercer novas tarefas. Uma melhor escalabilidade acabará por gerar uma considerável redução no tempo de adaptação dos sistemas, e possibilitará às empresas direcionar seus esforços na manutenção e gerenciamento de módulos menores, de menor complexidade,

tão eficazes quanto seus equivalentes monolíticos, e que lhe sejam efetivamente interessantes.

Com base na utilização de sistemas funcionalmente flexíveis, outros cenários potencialmente interessantes também podem ser visualizados, aqueles onde existam aplicações que executem de maneira adaptável ao ambiente de *hardware* e *software* disponíveis. Segundo esta idéia, diversas versões de cada módulo seriam construídas, levando-se em conta diferentes características de ambiente, tais como, sistema operacional utilizado, tipo de processador existente, quantidade de memória disponível, estilo da aplicação alvo (centralizada, distribuída, embutida, etc.) e dispositivo de *hardware* utilizado (*PC*, *Palm*, telefone celular, entre outros). Assim sendo, uma mesma aplicação poderia estar disponível para ser utilizada em diferentes configurações de ambiente. Por exemplo, numa situação em que uma empresa possua filiais independentes umas das outras, um novo módulo de *software* não encontraria problemas para ser instalado, pois possuiria diferentes versões que se adaptariam perfeitamente às idiossincrasias de cada uma das filiais.

Um outro aspecto a ser levado em conta é o financeiro, relacionado ao modelo de negócios envolvido. Num cenário de utilização de *software* completamente modularizado pode-se imaginar um modelo de negócios que gere contratos de utilização de *software* do tipo “pague somente o que for usar”, ou ainda, “pague somente pelo tempo em que usar”. Tome-se, como exemplo, os editores de texto. As últimas versões deste tipo de aplicação, disponíveis no mercado, agregam tantas funcionalidades distintas que a grande maioria dos usuários sequer toma conhecimento. Entretanto, acabam pagando por todas elas, pois seus valores estão embutidos no preço final do produto. Desta forma, paga-se caro por algo que nunca ou raramente será utilizado.

Em um modelo de negócios onde se pagasse apenas pelo que se utilizasse, este usuário tenderia a economizar uma quantidade significativa de recursos financeiros e de *hardware*, pois poderia adquirir apenas o módulo principal da aplicação, onde estariam presentes somente as funcionalidades essenciais do programa. Este módulo principal – um *kernel* – seria construído de maneira a ser integrável com módulos menores, agregadores de funcionalidades secundárias. No caso dos editores de texto, por exemplo, poder-se-ia ter disponibilizados módulos “secundários” (ferramentas de desenho, corretor ortográfico e tradutor, entre outros) separadamente do módulo principal, onde restariam apenas

funcionalidades básicas como abertura e salvamento de arquivos. Assim sendo, estes módulos adicionais seriam comprados e integrados *a posteriori*, conforme a vontade / necessidade do usuário.

Além disso, se o ambiente de utilização de *software* em questão puder disponibilizar esses módulos adicionais de maneira dinâmica (*on-line*), um modelo de negócios muito mais interessante poderá ser visualizado, aquele em que se pagará apenas pelo tempo em que se utilizar cada módulo. Neste caso, estabelecer-se-ia um contrato de uso onde estes módulos não seriam mais comprados, e sim "alugados". Ou seja, o *software* visto como um serviço e sendo acessado sob demanda. O usuário compraria apenas o pacote básico da aplicação, e sempre que fosse de seu interesse, requisitaria a integração (e posteriormente, a retirada) de algum pacote adicional, pagando somente pelo tempo de utilização do mesmo.

1.2 Objetivos

1.2.1 Gerais

Esta dissertação propõe um modelo que objetive diminuir o problema da baixa flexibilidade funcional e customização de *software*, ao vê-lo sob a perspectiva de uma composição dinâmica e inteligente, de acordo com as necessidades vigentes de um usuário / uma empresa. Essa composição terá como base a utilização de componentes funcionais, adaptados ao usuário / ambiente de utilização.

Cada um desses componentes funcionais poderá ser transparentemente inserido ou retirado do sistema, otimizado sem gerar interferência nos outros, integrado com outros módulos heterogêneos e / ou com sistemas legados, de maneira que passem a fazer parte do sistema como um todo. Do ponto de vista da fonte desses componentes, o que se pretende é dar uma nova dimensão aos chamados Provedores de Serviços de Aplicação (*Application Service Providers – ASPs*), tornando-os **Provedores de Aplicação (PAs)**. Este trabalho propõe um ambiente de utilização de *software* baseado em componentes, onde uma aplicação tenha suas funcionalidades dinâmica e inteligentemente instanciadas por um

processo de “plugagem” de componentes vindos dos mais adequados PAs, pertencentes a uma **Federação de Provedores de Aplicação (FPA)**.

1.2.2 Específicos

- Fazer uma análise de alguns modelos de aplicações compostas existentes, no intuito de se comparar o processo de composição dinâmica utilizado por elas com a arquitetura FPA.
- Avaliar as tecnologias-base escolhidas como suporte para este trabalho: Componentes, Provedores de Serviços de Aplicação e arquitetura *Peer-to-Peer*.
- Construir um protótipo para validar as idéias propostas. Este protótipo será construído para um cenário especificamente projetado, dotado de todas as características do modelo, e seus resultados serão tomados como referência para a validação da abordagem proposta.

1.3 Estrutura do Texto

Esta dissertação está dividida em cinco capítulos. Neste primeiro capítulo, foi descrito o contexto onde este trabalho está inserido, incluindo-se aí os principais problemas existentes e a descrição de possíveis cenários e metas a serem alcançados.

O capítulo 2 apresenta um levantamento bibliográfico sobre aplicações compostas. São descritas suas estruturas, suas possíveis configurações e são mostrados exemplos de utilização. Este capítulo também faz uma avaliação das tecnologias que servem de base para este trabalho, mostrando seus atuais estágios de utilização e possibilidades futuras.

O capítulo 3 apresenta a proposta conceitual da dissertação, centrada na *Federação de Provedores de Aplicação* e no processo de plugagem dinâmica. Serão discutidos os seus objetivos e a sua estrutura, detalhando-se todos os seus componentes.

O quarto capítulo se divide em três partes. Na primeira são detalhados o cenário-base, os requisitos e a estrutura do modelo de implementação do protótipo. Na segunda

parte estão descritos os testes efetuados, detalhando-se os critérios utilizados e apresentando-se uma avaliação dos resultados obtidos.

No capítulo 5 são apresentadas as conclusões deste trabalho e as perspectivas para futuros melhoramentos.

Capítulo 2

2. Aplicações Compostas

Um dos maiores desafios da Engenharia de Software nos últimos anos é a crescente necessidade de integração e colaboração de sistemas computacionais em escala cada vez mais global. Esta integração leva em conta aspectos como customização das aplicações, portabilidade entre arquiteturas e tecnologias de comunicação, e agilidade no desenvolvimento e na montagem de novas aplicações de maneira que estas possam estar disponíveis o mais rápido possível para a execução de suas tarefas e, em consequência, mais rapidamente suprir a demanda existente por produtos e serviços. Estima-se que aproximadamente 40% do custo de novas aplicações provem da integração de diferentes processos de negócios e diferentes aplicações, em vez de financiar o desenvolvimento de novas funcionalidades (E-Business Strategies, 2004). Para Golden ORB (2003), num mundo ideal, as empresas fariam seus negócios como se a sua gama de aplicações fosse uma única aplicação acessada através de uma interface baseada na Web. Em vez disso, o mundo real é constituído de grandes repositórios de dados que não interagem entre si, e por conjuntos distribuídos de aplicações que praticamente impossibilitam a unificação de processos entre empresas. Uma unificação que unisse aplicações dispersas, departamentos e outras empresas, faria com que a integração de negócios se comportasse como uma unidade coesa. É neste contexto que surgem as *Aplicações Compostas*.

Uma aplicação composta (*composite application*) é um *software* que combina elementos de um processo de negócios numa única interface de usuário. É uma camada inter-sistemas / inter-empresas que coordena todas as aplicações deste processo de maneira que elas se comportem como se fossem uma só. Pode também prover acesso *on-line* a toda alimentação de dados necessária e aos sistemas de controle desses dados, com várias

interfaces se comunicando umas com as outras no objetivo de realizar uma atividade de negócios (Golden ORB, 2003) (Woods, 2003) (Krass, 2004).

Os usuários de aplicações estão frequentemente restritos aos conceitos definidos pelo desenvolvedor de como e por que o programa deveria ser usado. Os sistemas ERP (*Enterprise Resource Plannning*) completos, aqueles onde todas as funcionalidades estão localizadas em uma única aplicação, são citados como exemplo de aplicação empresarial extremamente rígida, pois são construídos para executar tarefas bem específicas. Esta rigidez é interessante por um lado porque permite aos usuários executar seus trabalhos de maneira rápida e fácil. Por outro lado, isto requer que os procedimentos e práticas de negócios em geral permaneçam inalterados ao longo do tempo. E isto, é óbvio, não é verdade.

Entretanto, o cenário mais comum é aquele onde as empresas utilizam diferentes sistemas que, juntos, somadas as suas funcionalidades, funcionam como um grande sistema ERP. Neste caso, a maior dificuldade está na padronização dos processos e dos modelos de referência. Em um cenário ideal, onde esta padronização seja realidade e todos os processos e *software* de suporte processem os mesmos formatos de dados, acredita-se que o que irá variar, de fato, serão apenas as interfaces gráficas, os meios de exibição e a qualidade dos processos. Segundo Waloszek (2003), no presente momento, muitas empresas estão enfrentando o desafio de se adaptarem mais rapidamente às necessidades de seus clientes, não apenas localmente, mas ao redor do mundo. Esta situação acaba por forçá-las a melhorar suas infra-estruturas de comunicação e colaboração, de maneira a garantir melhorias nas tomadas de decisão e na produtividade.

As aplicações compostas são, em sua esmagadora maioria, baseadas em serviços Web (*Web Services*) (WebServices, 2004). Todavia, pode-se considerar também que alguns sistemas baseados em componentes se incluem nesta classificação. Como será visto na Seção 2.1, componentes de larga granularidade podem encapsular aplicações completas. No caso dos serviços Web, a idéia é a de que, através de um navegador, qualquer pessoa conectada à Internet possa utilizá-las (Krass, 2004). Para tal, utilizam em sua composição padrões já estabelecidos da indústria de *software* com o intuito de tornar a adaptação dos sistemas a melhor possível. Um dos padrões mais utilizados é a linguagem de definição de dados XML (*Extensible Markup Language*) (Xml.Com, 2004). De maneira geral, pode-se dizer que as aplicações compostas são baseadas em serviços, utilizam o padrão XML na

formatação dos seus dados, e são construídas para serem executadas através dos diversos sistemas e redes de comunicação já existentes. Krass (2004) afirma que, essencialmente, aplicações compostas criam uma aplicação maior, composta de várias outras menores (incluindo-se aí sistemas legados e componentes de *software* internos ou desenvolvidos separadamente), na tentativa de chegar o mais perto possível de um pacote de *software* especialmente desenvolvido para determinada empresa.

Havenstein (2003) acha que aplicações compostas serão projetadas para permitir que se obtenham aplicações através de sua *composição* em oposição à *programação* das mesmas. Se se tomar um contexto baseado em componentes, esta estratégia é potencialmente interessante. Desta forma, pode-se imaginar a especialização das empresas ocorrendo entre aquelas que desenvolvam aplicações-núcleo (que possuem funcionalidades a serem expandidas e / ou personalizadas) e as que desenvolvam componentes de *software*. A idéia consiste no fato de que, as que desenvolverem núcleos venderão sistemas enxutos a seus clientes e que poderão ser, dependendo da vontade e necessidade destes, integrados com componentes provindos de diversas origens.

A proposta desta dissertação possui algumas semelhanças com a opinião de Havenstein (2003), que acha que, do ponto de vista do desenvolvedor, todos os dados estarão agrupados de maneira que o mesmo não terá que lidar com todas as idiossincrasias inerentes a eles, nem às suas interfaces de utilização em rede. A idéia é possibilitar a montagem de blocos (como “legos”) de dados que poderão ser acessados em qualquer formato. Entretanto, pode-se dizer que na arquitetura proposta aqui os blocos não serão apenas compostos de dados, mas também de aplicações. Para Krass (2004), uma das vantagens deste tipo de abordagem é o tempo de desenvolvimento. Um projeto típico de aplicação composta é atualmente medido em semanas, num enorme contraste com outros tipos de aplicação que levam de 12 a 18 meses em média para serem desenvolvidos.

Para Havenstein (2003), falar em composição é falar a respeito de visualização através da Web de enormes quantidades de informação, de maneira diferente da que os portais da Internet vinham usando no passado quando a integração era muito custosa. É também falar sobre a criação do que se chama de aplicação (visualização da informação), só que neste caso com a informação provinda de várias fontes distintas. Além disso, o conceito de aplicações compostas representa um passo significativo em direção à disponibilização de *software* como um serviço em uma arquitetura orientada a serviços,

onde desenvolvedores e analistas de negócios possam disponibilizar aplicações oriundas de uma camada de serviços que mapeie diferentes domínios de negócios.

A composição de aplicações é vista como a próxima etapa evolutiva do desenvolvimento de sistemas. Segundo E-Business Strategies (2004), por exemplo, primeiro apareceu a automação de tarefas (basicamente armazenamento de dados), depois veio a automação dos *workflows* departamentais (por exemplo, os sistemas de folha de pagamento), e a seguir surgiu a automação das transações (representadas pelas aplicações de comércio eletrônico). A ênfase atual tem sido na automação dos processos (aplicações de gerenciamento da cadeia produtiva), e dentro deste contexto, vê-se que a etapa seguinte desta evolução se dá com o surgimento das aplicações compostas.

Waloszek (2003) acha que aplicações compostas representam um novo tipo de aplicação e são vistas por muitos especialistas como sendo uma resposta viável aos desafios de um mundo de negócios em eterna transformação. Um fator a ser ressaltado nesta abordagem é a sua nomenclatura, pois ainda não existe um consenso na indústria de *software* a este respeito. Por enquanto, Aplicações Compostas é o termo mais usado, mas outras propostas existem, tais como, *Solução Através de Aplicações Inteligentes*, *Fusão do Processo de Negócios* e *Arquitetura Orientada a Serviços* (Krass, 2004).

Dentre as tecnologias utilizadas como suporte à composição dinâmica pode-se destacar: os Componentes, os Serviços de Software (com destaque para os Provedores de Serviços de Aplicação) e a arquitetura *Peer-to-Peer* (destacando-se a plataforma JXTA).

2.1 Componentes

Segundo Szyperski (1997), um componente de *software* é uma unidade de composição, que possui interfaces e dependências de contexto específicas e bem definidas, que pode ser utilizado independentemente ou ser composto por terceiros. Esta integração poderá ser parcial ou total, e não acarretará "danos" para a aplicação principal. O desenvolvimento baseado em componentes disponibiliza uma abordagem mais flexível do que os métodos tradicionais, nos quais os sistemas são projetados de maneira monolítica, como uma única aplicação.

Componentware (Beneken *et al.,* 2003) – componentes e tecnologias baseadas em componentes – é uma expressão que vem se tornando bastante conhecida na engenharia de *software*. Os recentes esforços no sentido de se desenvolver melhores e mais eficientes ambientes computacionais, principalmente aqueles compostos por sistemas que cooperam em rede, começaram a fazer uso significativo desta tecnologia. O uso de componentes está diretamente relacionado com o desenvolvimento de aplicações mais escaláveis, flexíveis, e sobretudo, mais baratas.

Entretanto, a criação de ferramentas e metodologias que possibilitem o desenvolvimento destes módulos de *software* autônomos, integráveis entre si e reutilizáveis não é recente; já vem sendo pesquisada há mais de trinta anos na engenharia de *software*. Segundo Beneken *et al.* (2003), a idéia do desenvolvimento destes módulos remonta ao artigo publicado por M. D. McIlroy em uma conferência da OTAN, em Garmisch (Alemanha) no ano de 1968 (McIlroy, 1968), onde é proposta a idéia de produção em massa de componentes de *software*. Desde então, o desenvolvimento de componentes vem evoluindo em várias direções, algumas destas, inclusive, contraditórias (Beneken *et al.,* 2003). De maneira geral, no entanto, os sistemas baseados em componentes são cada vez mais utilizados. Existem atualmente diversos ambientes de programação que possibilitam esta utilização. Nos últimos anos, o crescimento deste tipo de desenvolvimento foi tal que transformou as expressões "componente" e "tecnologia baseada em componentes" em jargões da moda.

Componentes são projetados para executar desde simples tarefas até as mais complexas. Podem possuir granularidade variada, o que significa dizer que podem ser

implementados desde uma pequena função até um módulo bem maior, como um subsistema. Por exemplo, um componente pode ser construído para executar uma pesquisa específica em uma base dados, assim como também pode ser projetado para interagir em um ambiente de cooperação entre sistemas interligados em rede, possibilitando a conclusão de um processo de negócios. Beneken *et al.* (2003) vê várias vantagens na utilização de componentes: eles podem ser adaptados para o tipo de ambiente que irão executar; podem ser substituídos total ou parcialmente sem causar prejuízo ao resto do sistema; podem ser escritos em diferentes linguagens de programação, usando diferentes tecnologias e voltados para diferentes sistemas operacionais; suas interfaces explícitas permitem a conexão e a desconexão de partes cooperantes dentro do mesmo sistema.

Cada componente é um módulo de *software* que encapsula uma funcionalidade e que se comunica através de uma interface. Se uma analogia com a programação orientada a objetos pode ser feita aqui, pode-se dizer que os componentes, diferentemente dos objetos, fazem uma distinção clara entre suas "declarações" (especificações) e seus "métodos" (implementações) (D'Souza *et al.*, 1999). Os componentes também possuem grande encapsulamento e grande potencial de reúso. De fato, eles diferem dos objetos comuns em detalhes como linguagem de programação, ambiente de execução e nível de integração com a aplicação principal.

2.1.1 Componentes de Prateleira

Os Componentes de Prateleira ou *COTS (Commercial Off The Shelf)* são uma especialização do tipo clássico. Segundo Estublier *et al.* (2001), o desenvolvimento de sistemas que utilizam esta tecnologia não visa substituir o modelo clássico, mas sim complementá-lo. E cita como principal diferença a comunicação, que no modelo clássico é feita de maneira explícita – pois os componentes possuem conhecimento mútuo – e que aqui é feita implicitamente – pois os componentes são ignorados uns pelos outros. Outras diferenças marcantes entre um componente *COTS* e um componente clássico (Estublier *et al.*, 2001) (COTS, 2004) são as seguintes:

- Os *COTS* são como caixas pretas. Como seus códigos-fonte não são disponíveis, alterações não são possíveis;

- São geralmente projetados para atuarem de maneira *stand-alone*, ou seja, não requerem a presença de outras ferramentas (com exceção do sistema operacional e, eventualmente, serviços de rede);
- Interagem com o ambiente (ou mundo externo) de diferentes maneiras: via linhas de comando dos sistemas operacionais (*shell*), via interfaces programáveis de aplicação (*APIs – Application Programming Interfaces*), via interfaces gráficas com o usuário (*GUIs – Graphical User Interfaces*), via dispositivos de entrada e saída, entre outros;
- Podem possuir armazenamento local, em formato não publicado;
- Possuem comportamento (implementação) não-determinístico, ou seja, devido à interação entre diversos tipos de componentes pode-se considerar que exista este tipo de comportamento;
- São atualizados pelos seus desenvolvedores; o usuário tem pouco ou nenhum controle sobre as datas e o conteúdo técnico de eventuais mudanças.

Em contraste, um componente clássico é projetado para ser usado como um módulo de aplicação, e, como tal, vem com alguma documentação, definindo as interfaces disponíveis e requisitos bem detalhados de seu comportamento. A composição clássica de componentes consiste na conexão entre serviços requisitados e serviços disponíveis, o que significa que as suas assinaturas (especificações) devem ser compatíveis, de tal forma que possam compartilhar o mesmo tipo de dados. Em contraste, os componentes *COTS* não necessitam de serviços e não publicam seus tipos de uso. Portanto, possuem uma interoperação mais complicada.

2.1.2 Enterprise JavaBeans

A plataforma *Java 2, Enterprise Edition* (J2EE) define um padrão para o desenvolvimento de aplicações empresariais *multicamada*. A plataforma J2EE procura simplificar este tipo de aplicação, baseando-as no uso de componentes padronizados e na disponibilização de um conjunto de serviços de gerenciamento desses componentes, no propósito de diminuir a complexidade das aplicações (J2EE, 2004).

O *Enterprise JavaBeans* (EJB) é um modelo de componentes *server-side* baseado na plataforma J2EE (EJB, 2004). É dos modelos tradicionais de desenvolvimento baseado em componentes um dos mais utilizados ultimamente. O aparecimento de ferramentas e ambientes próprios para o desenvolvimento, aliado à evolução das infra-estruturas de comunicação (larguras de banda e *middlewares*), proporcionou um enorme crescimento da utilização deste tipo de ambiente nos últimos anos. Além das vantagens já vistas (Seções 2.1 e 2.1.1) do uso de componentes de *software*, estes ambientes também disponibilizam um gerenciamento do ciclo de vida dos componentes e um controle de transações entre eles. Este suporte integrado às ferramentas é de grande relevância para quem desenvolve este tipo de *software*. O desenvolvedor fica liberado para trabalhar apenas a lógica interna do módulo, devendo apenas observar as interfaces do mesmo com o mundo exterior. Todo o resto da funcionalidade de suporte será agregada ao componente pela ferramenta que será utilizada.

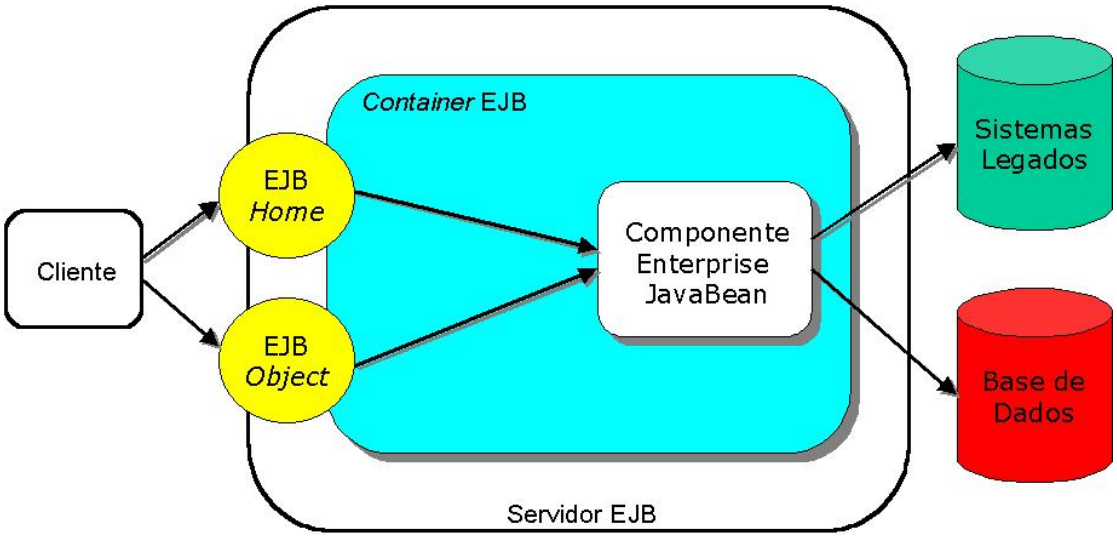

Figura 2.1 – Arquitetura EJB

A Figura 2.1 mostra a estrutura da arquitetura EJB. Ela é composta por um servidor, que irá abrigar o *container* EJB. Um *container* é uma entidade responsável pelo gerenciamento dos componentes e pelo controle de transações. O *container* pode ser acessado através de duas interfaces externas: EJB *Home* e EJB *Object*. A primeira opera no ciclo de vida dos componentes, serve para criá-los, localizá-los e removê-los. Através do EJB *Object* os clientes acessam remotamente as funcionalidades dos componentes. Cada componente presente no *container* EJB pode acessar aplicações externas a ele, tais como sistemas legados e bancos de dados. Apesar deste modelo ter sido projetado

preferencialmente para operar com a plataforma Java, segundo EJBFactory (2003) e Juric (2002), as aplicações-cliente que acessam os componentes EJB não necessitam necessariamente serem escritas em Java, bastando para isso que suportem o mesmo protocolo de comunicação – RMI / IIOP (JavaRMI , 2004), CORBA (OMG, 2004a), etc. Mas apesar desta flexibilidade em relação aos clientes, os componentes EJB só podem ser implementados na linguagem Java, o que se configura uma enorme desvantagem deste modelo. Os tipos de componentes EJB são o *Entity Bean*, o *Session Bean* e o *Message-Driven Bean*.

O primeiro pode ser considerado como um objeto persistente, que apesar de ser criado através de uma aplicação, precisa ser explicitamente destruído ou continuará existindo mesmo depois do final desta aplicação: é potencialmente interessante para quando existir a necessidade de se armazenar dados. Cada *Entity Bean* deve também possuir uma chave primária, que seja única no sistema. Os *Entity Beans* podem ser acessados remotamente e por mais de um cliente, inclusive por aqueles que não os criaram. Para isso, basta estarem conectados à mesma rede e procurarem pelos componentes desejados (EJBTutorial, 2004).

O *Session Bean* é um pouco diferente. Ele não é mais um objeto persistente, pelo contrário, é "volátil" e serve unicamente para a execução de uma tarefa. Este tipo de componente é voltado para a execução de serviços. Um *Session Bean* utiliza os dados que recebe para processar seus resultados e os devolve a quem os pediu: pode-se fazer aqui uma analogia com os RPCs (*Remote Procedure Calls*). Ao contrário dos *Entity Beans*, os *Session Beans* não podem ser compartilhados por mais de um cliente, a não ser em casos excepcionais em que duas aplicações distintas consigam compartilhar o mesmo manipulador (*handle*). Os *Session Beans* são potencialmente interessantes para a utilização em aplicações baseadas na *Web*. Neste caso um único *web-server* (servidor) pode ser acessado por infinitos navegadores (clientes), que poderiam utilizar um *Session Bean* para executar determinada tarefa em comum. Entretanto, de maneira geral, os clientes podem ser qualquer tipo de aplicação.

Os *Message-Driven Beans* são um outro tipo de componentes EJB. Este tipo é especial pois permite a integração com o *Java Message Service* (JMS), que é um serviço de suporte ao EJB através de mensagens assíncronas. Desta forma, as funcionalidades dos *Message-Driven Beans* poderão ser acessadas através de mensagens que estejam

enfileiradas no JMS. De maneira geral, o EJB possui várias semelhanças com outro modelo tradicional, que será visto a seguir.

2.1.3 CORBA Component Model

O *CORBA Component Model – CCM* é o modelo de componentes do Object Management Group (OMG), e é parte da especificação do CORBA 3. O CMM herda todas as características do CORBA como independência de linguagem e de plataforma, interoperabilidade e escalabilidade. Além disso, o CMM adiciona a elas um modelo de componentes distribuídos e um *container* para o gerenciamento do ciclo de vida destes componentes (OMG, 2002).

Um típica arquitetura CCM consiste dos seguintes itens (Raj, 1999):

- De *containers* CCM;
- De componentes CORBA, que são executados nestes *containers*;
- Do "Adaptador de Objetos Portáveis" (*Portable Object Adapter – POA*);
- Do "Negociador de Acesso a Objetos" (*Object Request Broker – ORB*);
- De outros serviços CORBA, tais como:
  - Transações;
  - Segurança;
  - Persistência;
  - Eventos;
  - Entre outros.

A Figura 2.2 apresenta a estrutura do componente CCM:

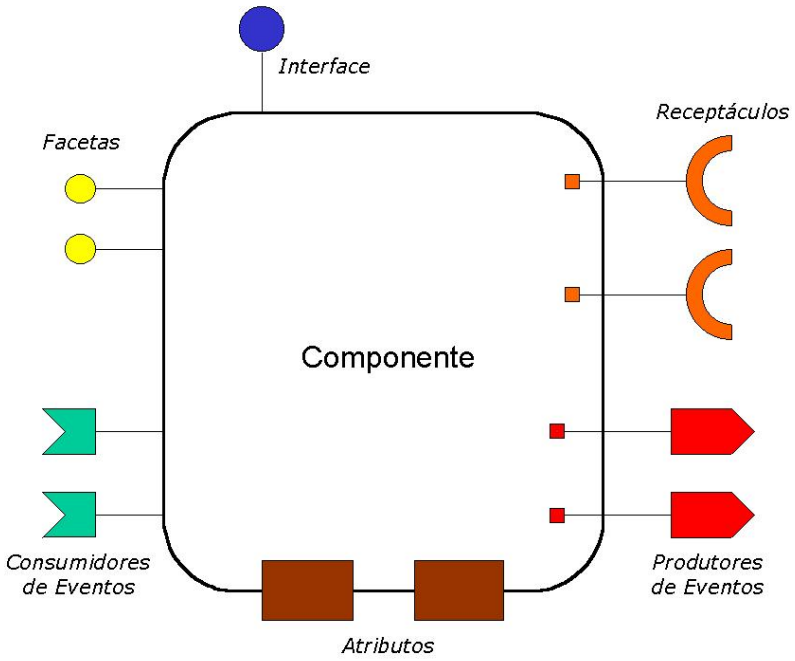

Figura 2.2 – Componente CCM

A estrutura básica de um componente deste tipo é composta por (OMG, 2002):

- Uma interface, que define os métodos e atributos que são acessados externamente, e que é processada pelo compilador IDL (*Interface Definition Language*);
- Propriedades, que são qualificadores do componente. Assim como um objeto, eles são usados para armazenar valores, constantes ou variáveis, e podem ser utilizados por qualquer um dos métodos do componente;
- Facetas, que são interfaces de acesso a determinados métodos. O padrão CCM diz que as facetas podem ser vistas como uma maneira dos projetistas associarem "métodos de contexto" a um componente. Um método definido em uma faceta só pode ser acessado por componentes que possuam portas associadas a ela;
- Receptáculos, que são uma maneira abstrata de se criar uma porta de comunicação em um componente para que este receba comunicação de determinado tipo. Os receptáculos necessitam de facetas associadas para se comunicar;
- Mecanismos de produção e consumo de eventos, que se subdividem em:

- Produtores de Eventos: incorporam o potencial que o componente possui de gerar eventos de determinado tipo, e disponibilizam mecanismos de associação de consumidores a esses eventos;
- Consumidores de Eventos: englobam o potencial que o componente possui de receber eventos de determinado tipo.

A Figura 2.3 mostra a arquitetura CCM:

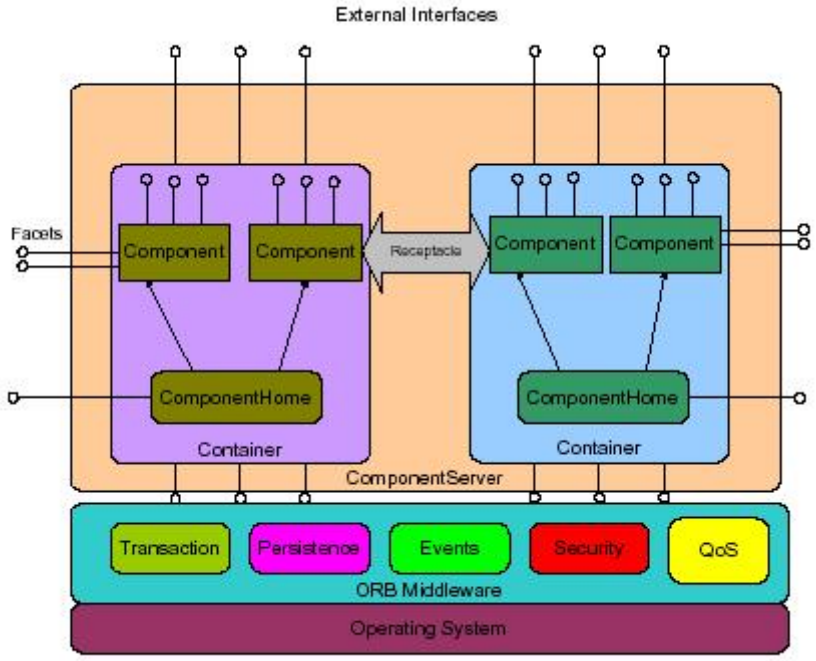

Figura 2.3 – Arquitetura CCM (Balasubramanian, 2002)

Nota-se aqui a existência de *containers*, que possuem as mesmas funções de seus similares EJB. A interface *ComponentHome* gerencia o ciclo de vida dos componentes e suas funcionalidades são acessadas pelo mundo externo através de suas interfaces externas. Os componentes CCM podem ser escritos em qualquer liguagem, e comunicam-se com aplicações externas ao *container*. Além disso, o CCM necessita de um ORB (*Object Request Broker*) para funcionar. Um ORB é o suporte que trata de vários aspectos ligados à implementação das semânticas do RMI do CORBA. Portanto, pode-se concluir que o CCM é essencialmente dependente do CORBA.

O servidor CCM (*ComponentServer*) disponibiliza um ambiente de execução, multiprocessamento, balanceamento (*load-balacing*), acesso, serviços de transações e

identificação e torna os *containers* visíveis (Raj, 1999). Um *container* CCM funciona como uma interface entre o componente e o mundo externo, um cliente CCM nunca acessa um componente CORBA diretamente. Qualquer acesso é feito através de métodos gerados pelo *container*, que por sua vez, invocam métodos dos componentes. Existem dois tipos de *containers* CCM (Raj, 1999):

- Transitórios: Que contêm componentes transitórios ou não-persistentes, ou seja, aqueles cujo estado não é salvo pelo *container*;
- Persistentes: Possui componentes persistentes, cujo estado é salvo entre uma transação e outra.

O cliente CCM é aquele que faz uso de componentes CCM nas suas operações. Ele procura a identificação do *container* que possui o componente desejado através do serviço de nomes CORBA *COSNaming*, e utiliza a interface *ComponentHome* para obter uma referência para este *container*. Feito isto, o cliente utiliza então o *container* para invocar os métodos do componente desejado.

Os componentes CCM se subdividem em quatro tipos:

- Componentes de serviço (*Service Components*): Cada componente de serviço é geralmente associado a um cliente CCM e seu ciclo de vida é restrito a uma única operação (ou única invocação de método). Ele é criado e destruído pelo cliente associado a ele. Componentes de serviços não “sobrevivem” a um desligamento do sistema;
- Componentes de sessão (*Session Components*): Um componente de sessão também está associado a um único cliente CCM, e da mesma forma que os componentes de serviço, é criado e destruído por seu cliente. Componentes de sessão podem possuir ou não estado e são muito semelhantes aos *Session Beans* (Seção 2.1.2). Os componentes de sessão também não “sobrevivem” a um desligamento do sistema;
- Componentes de processo (*Process Components*): Componentes de processo sempre possuem estado. Entretanto, um componente de processo pode ser compartilhado por mais de um cliente CCM. Seus estados é salvo ao longo das diversas invocações, portanto, “sobrevive” aos desligamentos;

- Componentes-entidade (*Entity Componentes*): Um componente-entidade é bastante similar a um componente de processo, pois possui estados persistentes entre diversas invocações, pode ser compartilhado entre múltiplos clientes e "sobrevive" a um desligamento do sistema. Contudo, um componente-entidade possui um identificador exclusivo associado e é muito semelhante a um *Entity Bean* (Seção 2.1.2). Desta forma, os componentes-entidade podem ser explicitamente identificados pelos clientes, diferentemente dos componentes de processo.

Para suportar todas essas características, o CCM expande a Linguagem de Definição de Interface CORBA (IDL) e cria a Linguagem de Definição para a Implementação de Componentes (CIDL) (Bartlett, 2001). A CIDL adiciona novas palavras-chave para descrever o comportamento dos componentes CCM, além da especificação necessária para interação com o *container* CCM.

Interoperabilidade com o EJB

De maneira geral, o CCM e o EJB são modelos que possuem muitas semelhanças, porque foram projetados com os mesmos objetivos (iCMG, 2004). Ambos possuem um ambiente de execução e gerenciamento de componentes (*containers*), e utilizam diferentes tipos de componentes (por exemplo, persistentes e não-persistentes). O CCM, inclusive, possui um adaptador (*bridge*) para se comunicar com componentes EJB.

Entretanto, durante a fase de desenvolvimento, um componente EJB é originalmente definido na linguagem Java, e um componente CCM é definido através da IDL e da CIDL. Por causa disso, diferentemente do EJB, o CCM é independente de linguagem.

2.1.4 Microsoft COM / DCOM / .NET

Um outro importante modelo de componentes que pode ser citado é o Microsoft .NET (NETDev, 2004). O .NET é mais recente que seus similares Sun e OMG, lançado em 2002. Ele é uma evolução e ao mesmo tempo uma alternativa aos modelos anteriores da Microsoft (COM / DCOM). A primeira especificação Microsoft de componentes foi o

Component Object Model (COM) (COM, 2004). O COM surgiu da necessidade de se obter comunicação entre módulos de *software* construídos em diferentes linguagens de programação. Cada componente passou a ser identificado com um valor exclusivo de 128 bits, chamado de *Global Unique Identifier* (GUID). Os GUIDs são armazenados no registro do sistema operacional, de maneira que podem ser referenciados por qualquer aplicação. A cada referência, uma instância é criada dinamicamente e pode então ser usada pelo programa cliente. Os componentes COM permitiram aos desenvolvedores em plataformas Microsoft um nível de flexibilização de *software* muito maior do que os até então mais utilizados, objetos. Entretanto, apesar disso, alguns problemas persistiram, como por exemplo, um muito similar ao conhecido "*DLL Hell*" (Pratschner, 2001). O problema do "*DLL Hell*" poderia acontecer sempre que uma nova versão de biblioteca dinâmica (*Dynamic Linked Library – DLL*) vinculada a mais de um programa fosse substituída por outra, obrigando que todas as referências a ela fossem revistas e adaptadas, geralmente através de um *shut down* (desligamento) de todas as aplicações que a referenciassem. Da mesma forma pode acontecer com um componente COM. Sempre que for necessário substituí-lo por uma versão mais recente, todos os clientes que o referenciam deverão ser reinicializados de maneira que atualizem as suas interfaces com a referência mais nova.

Mas apesar destes problemas, o modelo COM mostrou-se confiável na sua utilização. Tanto que logo foi adaptado para ser utilizado em sistemas distribuídos. A esta nova arquitetura, chamou-se *Distributed COM* (DCOM) (Horstmann *et al.*, 1997). O DCOM manteve inalterado o esquema de identificadores, que agora deveriam estar registrados de maneira exclusiva em todas as máquinas do sistema. Ele basicamente é um protocolo baseado em *Remote Procedure Calls* (RPCs), utilizado para a comunicação remota entre componentes distribuídos. A grande desvantagem do DCOM é que ele não utiliza a porta HTTP padrão (80), mas outra que precisa estar sempre aberta para viabilizar a comunicação. E esta é uma característica que dificulta muito a sua utilização em sistemas que possuam *firewalls* (Nelson, 1998).

Somente o acesso remoto não bastava para se ter uma plataforma realmente voltada à utilização de componentes. Assim, a Microsoft lançou também o *Microsoft Transaction Server* (MTS), que funciona como uma espécie de *container* para os componentes COM. O MTS processa agrupamento de componentes, chamados de pacotes, e oferece toda a infra-

estrutura necessária para a realização de transações entre os mesmos, da mesma forma que os *containers* EJB e CCM também o fazem. Quando foi lançado o Windows 2000, a Microsoft resolveu integrar o MTS a um novo serviço chamado de *Microsoft Message Queue Server* (MSMQ), e deu a este novo serviço o nome de COM+. O MSMQ possibilita a comunicação assíncrona entre cliente e servidor e é potencialmente interessante para absorver o impacto da inacessibilidade deste último (Thakkar, 2001).

Componentes .NET

O DCOM e o COM+ ainda são largamente utilizados pelos desenvolvedores em plataforma Microsoft. Mas, apesar disso, um novo modelo foi disponibilizado, o .NET (NETDev, 2004). Seu objetivo foi simplificar a maneira como os componentes são referenciados no sistema e assim, facilitar a sua plugagem e contornar problemas existentes, como o "*DLL Hell*". Os componentes .NET são também conhecidos por *Assemblies*. Cada *assembly* possui um arquivo de *metadados* denominado *Manifest*. Os *manifests* contêm as assinaturas dos métodos que o componente disponibiliza. A única condição para a utilização de um *assembly* é que ele esteja no mesmo diretório que o cliente. E como toda a informação a respeito deste componente se encontrará no seu *manifest*, não haverá mais a necessidade de registrá-lo, e múltiplas versões do mesmo componente poderão co-existir sem problemas na mesma máquina.

Em linhas gerais, o que se nota nessas arquiteturas tradicionais é a preocupação em dotar seus modelos de uma plataforma de suporte, que integre seus componentes em um ambiente voltado à interação, seja através de acesso remoto, seja através de eventos, seja através de filas de mensagens. A idéia padrão por trás do desenvolvimento baseado em componentes é a de se possuir uma estrutura completa, pronta para receber e gerenciar módulos de *software*, que serão acessados remotamente. Pode-se dizer que o desenvolvimento deste tipo de sistema tomou este rumo principalmente para ser um contraponto à Programação Orientada a Objetos (POO). Surgida como paradigma em meados dos anos 80, a POO logo tornou-se a principal metodologia de análise e de implementação de *software*. Ainda hoje, a grande maioria dos sistemas é projetada tendo-se em mente, pelo menos em algum momento do processo de análise, a interação entre objetos e seus equivalentes (classes, métodos, atributos, herança, etc.). Os sistemas baseados em componentes surgiram como uma alternativa, e ao mesmo tempo um

aperfeiçoamento à POO, principalmente nas questões de autonomia e independência dos módulos. Assim, pode-se entender toda a preocupação em dotar os componentes com o mínimo possível de tarefas não-funcionais, que ficariam à cargo das próprias arquiteturas (através de seus *containers*). Contudo, esta opção acabou por dotar os modelos de componentes de estruturas de suporte, tendo estes de qualquer forma, de ser compatíveis com elas. A aglomeração em um único servidor, se facilitou o controle do fluxo das tarefas e a comunicação, acabou por centralizar o modelo, e possibitou desta maneira a existência de todas as desvantagens deste tipo de abordagem centralizada (Seção 2.3).

2.1.5 Comparação Entre Tecnologias

Plataforma	CORBA	J2EE	.NET
Modelo de Componentes	CCM	EJB	.NET
Padrão	Aberto (*Open Source*).	Aberto (*Open Source*).	Não-Aberto / Proprietário.
Linguagem de Implementação	Independente (Necessita de um ORB).	Java.	Independente (Necessita da Plataforma .NET).
Plataforma	Independente.	Independente.	Microsoft.
Implementações-Padrão Disponíveis	Semi-completa (Algumas características-padrão ainda não estão presentes em todas as implementações).	Diversas.	Diversas.
Container	Sim, disponibiliza acesso simplificado ao ORB.	Sim.	Não explicitamente. Os serviços relativos ao "*container*" são disponibilizados pelo sistema operacional (MS Windows).
Documentação e suporte	Razoáveis.	Bons.	Bons (Mas faltam especificações formais).
Implantação Estática	Componentes são implantados nos *containers* como pacotes compactados (zip).	Componentes são implantados nos *containers* como pacotes compactados (jar).	Componentes são registrados na *Global Assembly Cache (GAC)* e disponibilizados em um diretório compartilhado.
Plugagem Estática	Componentes plugam-se em outros usando suas facetas e receptáculos, como um "Lego".	Clientes usam componentes através de suas interfaces. Não existe o conceito de plugagem.	Clientes usam componentes através de suas interfaces. Não existe o conceito de plugagem.
Plugagem Dinâmica	Parcial.	Parcial.	Parcial.
Complexidade de Desenvolvimento	Alta.	Alta.	Média (Possui uma "noção" mais simples do que seja um componente.
Mobilidade de Código	Componentes usam o suporte do ambiente de programação para efetuar a mobilidade.	Código pode ser dinamicamente carregado de qualquer lugar, local ou remotamente.	Existe na forma de aplicações que podem ser carregadas através dos controles ActiveX.
Protocolos de Suporte	IIOP, Java RMI sobre IIOP.	Java RMI, IIOP, SOAP.	SOAP, COM / DCOM.
Restrição aos Clientes	Estar de acordo com o protocolo do servidor.	Estar de acordo com o protocolo do servidor.	Estar de acordo com o protocolo do servidor.

Tabela 1 – Comparação entre Modelos Tradicionais de Componentes

2.1.6 Plug-Ins

Plug-ins são pequenos módulos de *software* que são adicionados a um programa para estender a sua funcionalidade (Plug-Ins, 2004) (Chastain, 2004). Ou seja, *plug-ins* são componentes de *software*. Apesar desta definição, entretanto, considera-se de maneira geral que *plug-ins* são componentes construídos para serem acessados especialmente via um navegador Internet. E isto fica claro quando se verifica que existem pacotes específicos para a criação e implementação de plug-ins, como o *Java Plug-in Component* (JavaPlugIn, 2004), que faz parte da linguagem Java.

Um plug-in é acessado pelo navegador através de referências (*tags*) da linguagem HTML. O navegador carrega dinamicamente o módulo e o executa. Esta execução se dá executando-se diretamente o *plug-in* (se este for um outro programa) ou através da integração com o navegador (utilizando-se *applets*, por exemplo). Existem dezenas de *plug-ins* dos mais variados tipos, mas principalmente aqueles relacionados a tarefas multimídia, como por exemplo:

- Acrobat Reader: Leitor de arquivos pdf;
- Real Player: Tocador de vídeo e áudio;
- Shockwave Player: Usado em animações 3D e jogos;
- Windows Media Player: Tocador de vídeo e áudio;
- QuickTime: Tocador de vídeo e áudio;
- Flash Player: Usado em animações e jogos;
- Etc.

A lógica da utilização de *plug-ins* está assentada na idéia de que as empresas não precisam disponibilizar todas as funcionalidades possíveis em seus programas. Parte delas, ou novas funcionalidades (no caso de ser desenvolvida por terceiros), podem simplesmente ser disponibilizadas na Internet para transferência. Ao usuário que possua tal interesse, é indicado o local de onde ele deve buscar o *plug-in*. Em determinadas vezes a aplicação exige apenas uma confirmação, mas em outros casos, o usuário deverá executar

manualmente todo processo. Isto quer dizer que terá que acessar o endereço indicado pelo navegador, trazer (*download*) o módulo e o instalar na sua máquina.

A tecnologia de *plug-ins* serviu de inspiração para os componentes dinâmicos (Seção 3.1.1).

2.1.7 Especificação de Componentes

A especificação de um componente é um dos pontos mais importantes do seu desenvolvimento. Ela envolve o detalhamento comportamental do pacote de *software* em questão e define os protocolos de acesso a seus dados e às suas operações. A interface – ou *plug* – é um conector por onde irão trafegar dados, mensagens e eventos. Para se aumentar o encapsulamento das funcionalidades destes componentes e se potencializar o reuso de *software* do sistema como um todo (via atualização ou substituição de um dos módulos), essas interfaces são projetadas para serem as únicas vias de comunicação com o mundo externo. Este fato acaba por definir requisitos básicos que devem ser levados em conta no projeto do componente.

Primeiramente, há a necessidade de se ter um rígido formalismo sintático. A implementação de um componente, situada em um módulo fechado, não é acessível externamente. Por isso deve-se tentar garantir o sucesso da sua comunicação. É essa garantia que possibilitará a plugagem de componentes em tempo de execução. Com a certeza de que o novo módulo poderá ser acoplado sem trazer nenhum tipo de interferência ou problemas ao *kernel* (ou ao sistema legado, se for o caso), os componentes poderão ser plugados ou desplugados a qualquer momento. Desta maneira, os limites lógicos para a escalabilidade do sistema se tornam enormes, haja vista que tudo se resumirá ao fato de quantos componentes poderão ser plugados a este sistema. Ou mesmo, tais limites praticamente desaparecem se forem levados em conta os *n* níveis de plugagem inter e intra-componentes. O formalismo sintático tem como tarefa definir o padrão e o formato de dados (evitando-se, por exemplo, *bugs* de conversões de tipo), definir o escopo das operações (controle pró-ativo) que poderão ser acessadas e características dos eventos (controle reativo) que serão gerados.

Outro requisito que deve ser considerado no projeto de um componente é o aspecto semântico. Somente o formalismo sintático não é suficiente para garantir o sucesso da interação com componentes; este apenas garante que haverá comunicação. É necessário, portanto, se verificar também se o conteúdo desta comunicação é consistente e satisfatório. Para isso deve-se definir um conjunto de restrições (*constraints*) para cada uma das propriedades, além de se efetuar uma verificação na consistência dos dados (de entrada e de saída) antes e depois de cada operação (Cheesman *et al.*, 2001). Por exemplo, dada uma operação que altera um registro em uma base de dados e que só pode ser executada se determinada propriedade for maior ou igual a zero, neste caso será necessário haver uma restrição nesta propriedade para se garantir a execução da operação. Além disso, o registro desatualizado e o registro alterado deverão ser verificados antes e depois de se efetuar a atualização, respectivamente. Fowler *et al.* (2000) mostra um exemplo de restrição aplicada em uma classe através de um diagrama de classes UML (Figura 2.4).

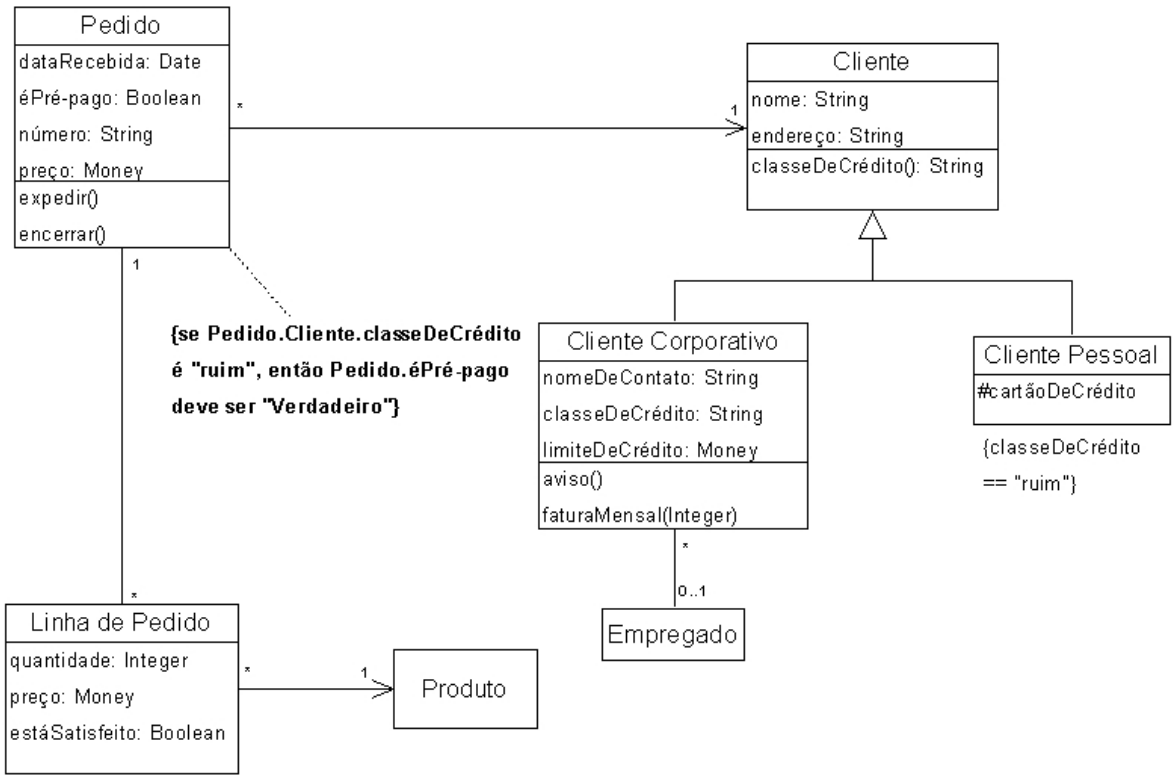

Figura 2.4 – Restrição / *Constraint* (Fowler *et al.*, 2000)

Tentativas de se obter um padrão para a especificação de componentes vêm sido feitas nos últimos anos e, apesar de ainda não existir uma especificação que sirva para todos os casos, algumas boas propostas surgiram neste tempo. Han (1999) propõe uma

abordagem que visa uma base para o desenvolvimento, gerenciamento e uso de componentes. Esta base é composta de quatro aspectos:

- A assinatura do componente. Fazem parte desta os requisitos sintáticos dos mecanismos de interação como, por exemplo, propriedades e eventos. Ela é o conjunto das assinaturas (declarações formais em linguagens de programação) de todas as interfaces. Um dos pontos principais deste item é explorar as propriedades que são observáveis externamente, pois são consideradas parte essencial de uma interface. A idéia é que os desenvolvedores possam usar (observar e eventualmente alterar) seus valores para entender a sua influência sobre o comportamento do componente. Esta abordagem serve para se efetuar uma customização e configuração em tempo de execução, tanto da forma tradicional (pró-ativa, através de uma invocação explícita ou de uma mensagem), quanto de forma reativa (através de eventos gerados pelo componente, que poderiam ser configurados também pelo usuário). O que está por trás disso é dotar o sistema com a capacidade de ser configurado dinamicamente.

- A semântica da interação do componente, incluindo-se aí a especificação semântica de cada um dos elementos da assinatura e as restrições existentes na sua assinatura em termos do seu próprio uso. Nesta abordagem são focada as restrições que dizem respeito aos relacionamentos entre elementos da assinatura. Segundo Han (1999), a assinatura e as restrições semânticas definem a “capacidade global” de um componente.

- O empacotamento das assinaturas de interface. Segundo esta idéia, o uso de um componente pode abranger duas situações: a) aquela em que ele executa diferentes papéis em um mesmo contexto e, b) aquela em que ele pode ser utilizado em diferentes contextos. Para as duas situações, seria necessário haver um empacotamento da assinatura de maneira que esta se tornasse flexível e adaptável a cada uma. Sugere-se, para cada componente, a definição de protocolos orientados a papéis ou perspectivas. Esses papéis seriam alterados via configuração dinâmica da interface.

- A caracterização do componente em termos de suas propriedades não-funcionais ou atributos qualitativos, tais como segurança, desempenho e confiabilidade. Visa endereçar duas questões: a) como caracterizar a qualidade de um componente e, b) como analisar o impacto de sua utilização em um determinado contexto.

Toda a especificação contida no modelo de Han (1999) é baseada na linguagem de definição de interface (IDL) CORBA (OMG, 2004a), incluindo as propriedades de tempo de projeto e tempo de execução, operações e eventos. E utiliza a especificação de componentes UML proposta por Cheesman *et al.* (2001).

Liu *et al.* (2002) propõe a especificação de componentes através da utilização de "*Design* por Contrato" (Meyer, 1992). Sustenta que para o desenvolvimento baseado em componentes obter sucesso dentro de uma organização, é necessário que os desenvolvedores dêem especial atenção ao projeto (*design*) destes, considerando-os abstrações independentes e possuidores de comportamentos bem definidos. Sua abordagem é direcionada para o desenvolvimento de componentes que possam ser facilmente substituídos por outros que suportem as mesmas interfaces. Seu objetivo final é construir componentes reutilizáveis e confiáveis e, para tal, utiliza técnicas de *design* por contrato para identificar as especificações necessárias à criação de componentes substituíveis e reutilizáveis. Utiliza também UML (*Unified Modeling Language*) (OMG, 2004b) para projetar as relações estruturais e a sua linguagem associada OCL (*Object Constraint Language*) para projetar os contratos (Cheesman *et al.*, 2001).

O início desta abordagem se dá através de uma análise de requisitos. São construídos um modelo de casos de uso (*use case model*) e um modelo de domínio (*domain model*), baseados na UML, para definir as interações entre o que é esperado pelo usuário e aquilo que o sistema irá oferecer. Depois, o modelo de domínio é refinado com o intuito de se obter um modelo de tipos de negócios (*business type model*) do sistema, ou seja, a identificação dos conceitos de negócios (por exemplo, tipos de dados) que estão dentro do escopo do *software* a ser construído. Segundo Liu *et al.* (2002), o ponto chave para o desenvolvimento baseado em componentes reside na identificação dos tipos centrais (*core types*). Cada um destes é um conjunto de dados e de funcionalidades ao redor dos quais uma interface é definida, isto é, cada um destes tipos dará origem a uma interface. Posteriormente um tipo central é selecionado pelo fato de ser um identificador de negócios diferente de outros tipos centrais e de possuir relativa autonomia em relação a eles. Neste caso, cada interface correspondente será também independente das outras. Esses tipos podem também estar associados a tipos de suporte, que por si sós, não determinam um identificador de negócios. Por exemplo, um modelo de tipos de negócio que possuísse um tipo central Curso Universitário, poderia por sua vez possuir agregados os tipos de suporte

Arquitetura, Medicina e Direito. Contudo, em outro modelo onde o curso Medicina fosse, por si só, um identificador de negócios, este então não seria mais visto como um tipo de suporte e sim como um tipo central.

A partir do modelo de tipos de negócio é construído um diagrama de responsabilidades de interfaces (*interface responsibility diagram*) (Figura 2.5), que dará origem a um modelo de informações de interface (*interface information model*) para cada interface (Figura 2.6). Cada um destes contem as informações de tipo que são necessárias para especificar completamente todo o comportamento das operações da interface. O conjunto destes modelos de interface formaria então o componente.

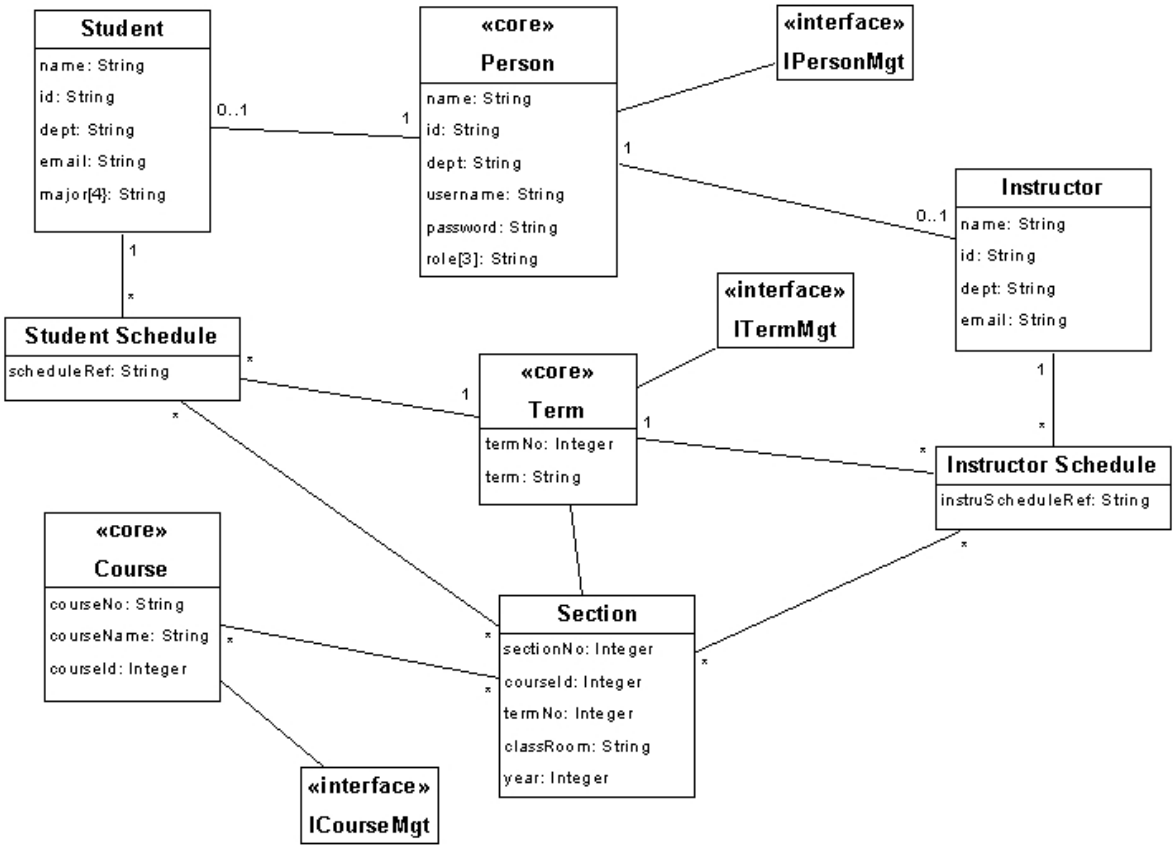

Figura 2.5 – *Interface Responsability Diagram* (Liu *et al.*, 2002)

Liu *et al.* (2002) acha que deveria ser possível se entender precisamente o comportamento de um componente baseando-se nas operações em suas interfaces, que seria possível se substituir um componente por outro que implementa o mesmo conjunto destas e que, assim, podería-se reusar um componente de maneira confiável em diferentes clientes e em diferentes contextos. O *design* por contrato é utilizado, então, como ferramenta para se obter esta confiabilidade. Sua idéia se baseia na utilização de

invariantes (*invariants*) ou restrições ligadas aos tipos ou às interfaces (no caso de restrições mais abrangentes), e no uso de requisitos de pré e pós-condições (*pre-conditions*, *post-conditions*) que todas as chamadas de operações devem satisfazer para estarem corretas (Meyer, 1992).

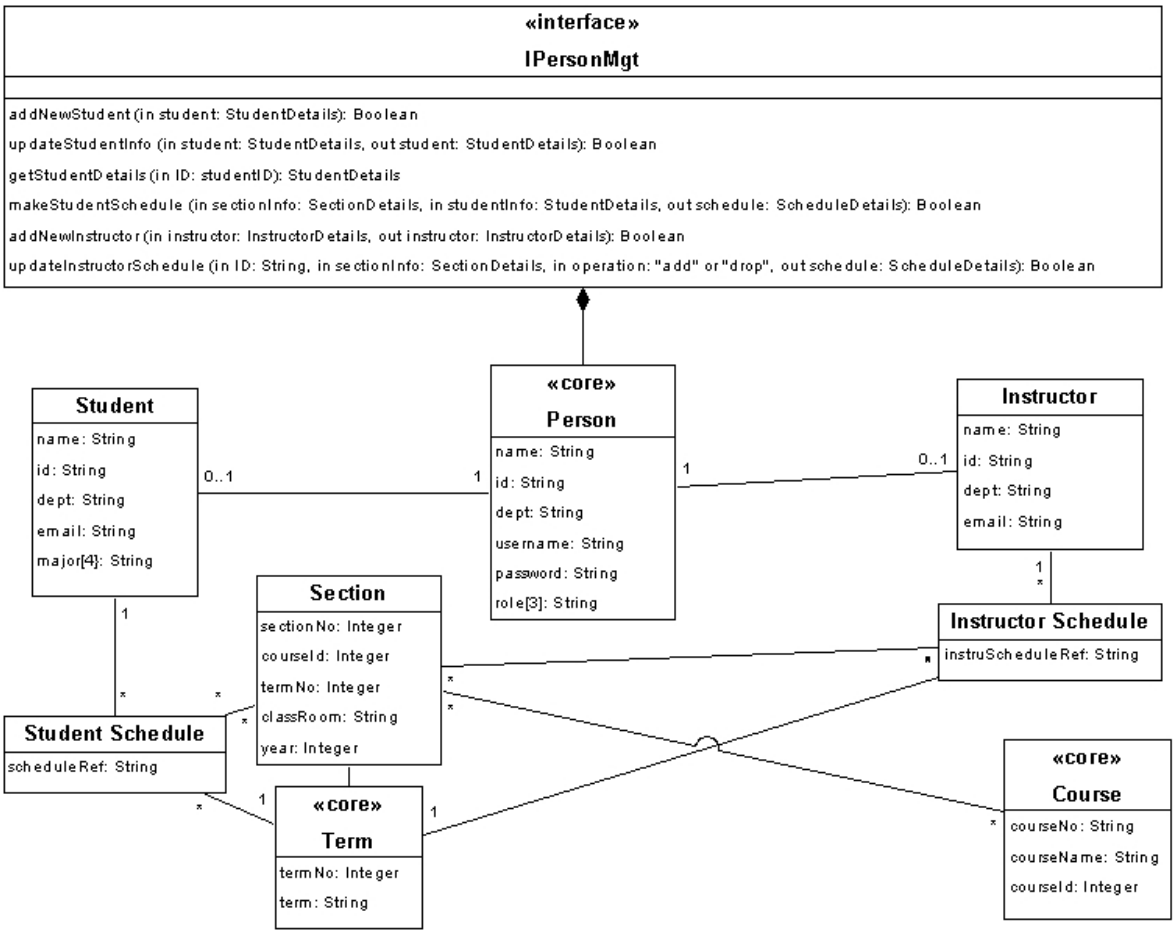

Figura 2.6 – *Interface Information Model* para a Interface *IpersonMgt* (Liu *et al.*, 2002)

Toda essa abordagem sustenta que devem haver expressões lógicas ou afirmações testadas em tempo de execução, de forma a aumentar a confiabilidade do código. A violação de alguma pré-condição indicaria a existência de um *bug* no utilizador do componente, já que este não teria observado todas as condições que tornariam a chamada correta. Já uma violação de pós-condição significaria um *bug* no componente, que falharia, então, em entregar seus serviços. Nos dois casos, haveria uma quebra de contrato. É este ponto, fundamentalmente, que esta técnica pretende evitar. Liu *et al.* (2002) direciona o uso de sua técnica para as tecnologias padrões de componentes: *Enterprise JavaBeans* – EJB (Sun, 2004), COM+ (Microsoft, 2002) e CORBA Component Model (OMG, 2002).

Entretanto, deixa claro que apesar deste enfoque, ela pode ser utilizada com outros tipos de componentes e, inclusive, com programas orientados a objetos. Estes foram os primeiros objetos de estudo de Meyer (1992) ao começar a elaborar a sua teria de *design* por contrato, no início dos anos 90.

Este uso é demonstrado em Liu *et al.* (2004), através de uma aplicação da abordagem anterior, utilizando-se a tecnologia Enterprise Java Beans (EJB) (EJB, 2004). Argumenta-se que, apesar de se focarem na especificação de componentes para alcançar todas as vantagens da teoria de desenvolvimento baseada em componentes, a maioria dos trabalhos de pesquisas nesta área presta pouca atenção ao mapeamento especificação-implementação. Dada determinada especificação, propõe-se três abordagens para a implementação da mesma. Cada abordagem possui mais ou menos entidades EJB na montagem de sua estrutura. A idéia é demonstrar que uma especificação bem feita pode ser adaptável a qualquer tipo de implementação. Demonstra que a abordagem mais rígida, a que possui apenas entidades EJB, é a mais confiável, entretanto, é a mais complicada e seu tempo de desenvolvimento é o maior de todos. A mais flexível destas três abordagens, com poucas entidades EJB e com classes Java (*Java Package*) compondo a estrutura, é a mais simples de ser implementada e seu tempo de desenvolvimento é muito inferior ao das outras duas. Todavia, sua grande desvantagem é que seus desenvolvedores terão que lidar com todos os problemas de transações e concorrências inerentes ao modelo, problemas esses, que são transparentemente tratados pelo *container* EJB (Seção 2.1.2), inexistente neste caso.

O que pode ser concluído da abordagem de Liu *et al.* (2004) é que uma especificação bem detalhada e correta adapta-se a qualquer ambiente de utilização de componentes, inclusive naquele onde a plugagem (composição) ocorre de maneira dinâmica e transparente ao usuário. A idéia da utilização de diagramas UML (Fowler *et al.*, 2000) para a modelagem dos requisitos e do uso da OCL para determinar as invariantes e os testes pré e pós-condições são perfeitamente adaptáveis a componentes genéricos, construídos para se adaptarem a qualquer ambiente de *hardware* e de *software*. Pode-se considerar também que a composição dinâmica seja um contrato, da maneira que descreve Meyer (1992). Segundo este, as verificações e condições que são postas no código para garantir a sua integridade são apenas um subconjunto do contrato. Seu principal propósito é ajudar a construir *software*s melhores através da organização da comunicação entre seus elementos,

especificando, tão preciso quanto possível, as obrigações mútuas e benefícios que estão envolvidos nessa comunicação. A essas especificações é que se dá o nome de contratos. A metáfora dos contratos é usada para garantir que essas comunicações ocorram não somente no campo de vagas expectativas de execução dos serviços, mas no campo de especificações precisas daquilo que esses serviços irão efetivamente disponibilizar (Meyer, 2003).

A lógica do *design* por contrato também reside na maneira como é organizado o conteúdo de uma biblioteca. Conforme esta lógica e tomando-se o paradigma da programação orientada a objetos pode-se, por exemplo, possuir uma biblioteca composta por excelentes classes para árvores, listas, pilhas e arquivos. Contudo, quando reunidas, elas podem não formar uma excelente biblioteca se forem inconsistentes. Se usarem convenções diferentes não podem ser parte de um mesmo projeto. E aqui se pode traçar um paralelo com um repositório de componentes. Estes precisarão ser construídos de maneira que seus componentes possam ser plugados em qualquer sistema (ou componente) que utilize a composição dinâmica.

As abordagens de Han (1999) e de Liu *et al.* (2002), entretanto, lançam seu enfoque em sistemas com poucos ou nenhum nível de especialização intra-componentes. São mais voltados para sistemas *flat* ou nivelados. Para cenários onde existe uma grande especialização de componentes, a plugagem de componentes a outros já plugados anteriormente pode alcançar vários estágios. Atkinson *et al.* (2000) propõe uma abordagem que se adapta a esses cenários, a abordagem KobrA. O projeto KobrA descreve um método para a engenharia de *software* baseada em componentes que procura tornar o componente uma parte integral do ciclo completo de vida do *software*. Apresenta como justificativa para tal o fato da maioria das tecnologias de componentes envolver-se apenas com a fase de implementação do *software*, e não dar maior importância às fases iniciais de análise e projeto (*design*) dos grandes projetos (*projects*) de *software*. Põe os componentes no centro do processo de desenvolvimento de *software*, não concentrando-se apenas nas fases de implementação, organização e implantação (*deployment*), mas também adotando estratégias para a criação, manutenção e para a disponibilização. Utiliza representação baseada em UML (Cheesman *et al.*, 2001) para capturar o conjunto completo das características e dos relacionamentos dos componentes. Isto não apenas torna a análise e *design* atividades “orientadas a componentes”, como permite também que a estrutura

essencial e o comportamento de sistemas baseados em componentes sejam descritos de maneira independente (porém compatível) de tecnologias específicas de implementações de componentes, tais como COM, CORBA ou JavaBeans.

A principal estrutura KobrA é o Komponent (Kobra *component*). Um Komponent difere de outros tipos de componentes pelo fato de ser descrito pela UML e por documentos de texto em um nível de abstração semelhante a da análise e projeto tradicionais (Atkinson *et al.*, 2000). Fabiunke (2001) diz que a descrição de um Komponent é divida em duas partes: a da especificação, que descreve as características externamente visíveis do Komponent e assim define os requisitos necessários à sua implementação, e a da realização, a qual descreve de que maneira o Komponent satisfaz esses requisitos em termos de interação com outros Komponents, ou seja, captura a arquitetura do Komponent. A especificação de um Komponent descreve suas propriedades em termos de classes UML e diagramas de objetos, diagramas estruturais UML e especificações textuais. A fase de realização utiliza classes UML e diagramas de objetos mais detalhados, diagramas de interação UML e diagramas de atividades UML. Na essência, KobrA vê um Komponent como um sistema completo por si só. Um Komponent deve ser capaz de ser executado como um sistema independente, ou de ser combinado com outros para formar um sistema ainda mais poderoso (mas ainda um Komponent).

KobrA pode ser visto como dois sistemas distintos, o dos *frameworks* KobrA e o das aplicações KobrA. Um *framework* KobrA disponibiliza uma genérica descrição dos elementos de *software* construindo uma família de aplicações. Em contraste com a maioria das abordagens, um *framework* KobrA engloba todas as variações concretas de uma família, não apenas as partes em comum. Isto é possível através da detecção de todas as características que estão dentro do *framework* e usando modelos de decisão para descrever as escolhas que distinguem os membros (aplicações) da família. Um *framework* KobrA é uma hierarquia de Komponents em forma de árvore, na qual cada relação pai-filho representa uma composição (um pai é composto por seus filhos). As especificações e as realizações contidas no Komponent são cuidadosamente controladas de maneira que apresentem consistência, capacidade de percorrimento na árvore (*trace-ability*) e relações de realização (Fabiunke, 2001).

O método KobrA define todos os processos envolvidos na criação e aplicação de um *framework* (ou aplicação) de um Komponent KobrA. Este método é baseado no princípio

fundamental da separação entre processo e produto, e também na separação entre desenvolvimento e manutenção. O resultado é um método que possui arquitetura altamente centralizada, incremental e escalável. Uma vez completo, um *framework* pode ser instanciado para disponibilizar aplicações específicas, construídas segundo a necessidade de clientes específicos, o que resulta numa aplicação com a mesma forma e estrutura de um *framework*, porém com todas as suas genericidades e características indesejadas removidas.

Todas essas propostas, apesar de diferentes, procuram apresentar soluções para a questão da especificação de componentes de *software*. O aspecto mais positivo a ser abordado de todas é a utilização da UML para formalizar as suas especificações. O modelo proposto por Cheesman *et al.* (2001), que serviu de base para elas, se mostra bastante viável por ser uma espécie de extensão da modelagem de objetos. Na Seção 3.1.1 será apresentada a especificação de *componentes dinâmicos*, que se baseou neste modelo.

2.1.8 Modelos de Plugagem

A grosso modo, pode-se dizer que a plugagem de um componente em outro módulo (componente ou não) equivale à plugagem de um objeto em outro. Se se considerar que um componente é um tipo de objeto mais sofisticado (D'Souza *et al.*, 1999), pode-se pensar em modelos de plugagens que tomem como ponto de partida a comunicação entre dois objetos.

Nos modelos tradicionais, a figura do *container* ou do sistema operacional se encarrega de estabelecer a comunicação do componente com o mundo externo. Os modelos EJB e CCM (Seções 2.1.2 e 2.1.3) implantam seus componentes através de pacotes compactados (*zip* ou *jar*). Uma vez implantados esses pacotes, os componentes estão aptos para a comunicação. De maneira geral, o *container* (EJB ou CCM) fornece toda a infra-estrutura necessária para que esta comunicação ocorra, seja através das facetas e dos receptáculos, seja através de eventos ou das interfaces externas. O modelo .NET (Seção 2.1.4) é um pouco mais simples. Para se implantar um componente .NET basta se copiar o arquivo para um diretório compartilhado. O sistema operacional Windows, através da plataforma .NET, gerenciará toda a comunicação.

Nestes casos, portanto, as infra-estruturas dos modelos oferecem o suporte necessário para a plugagem. Porém, se se pensar em um modelo onde tais estruturas não existam, se constatará a necessidade se elaborar alternativas para que a comunicação entre dois módulos de *software* (componentes) continue existindo. Assim, esta comunicação poderia ser, por exemplo:

- Via Eventos ou Mensagens Registradas no Sistema Operacional;
- Via *Application Programming Interfaces* (APIs) (Coach 2004);
- Via DLLs (*Dynamically Linked Libraries*);
- Via *Middlewares*, como o CORBA (Calim 2001) (Combine 2002);
- Via “Acesso Direto”;
- Entre Outros.

O método de comunicação de componentes via acesso direto foi utilizado no protótipo que será apresentado no Capítulo 4. Este método baseia-se na proposta de Liang *et al.* (1998), que fala sobre o carregamento dinâmico de um objeto na Máquina Virtual Java. Segundo Liang *et al.* (1998), a linguagem Java permite que se implemente objetos como classes abstratas a ser instanciadas durante a execução. Estas classes abstratas possuem as mesmas assinaturas dos componentes, isto é, suas declarações de métodos e de atributos são exatamente as mesmas. Estas classes abstratas não possuem qualquer implementação. Durante a execução, a Máquina Virtual Java as especializa após carregamento dos pacotes que possuem as implementações, e estas por sua vez passam a ter as suas funcionalidades disponíveis para o acesso via as interfaces das classes abstratas.

Desta forma, este método permite que se construa aplicações compostas apenas de funcionalidades básicas e por classes abstratas, representando as interfaces dos componentes passíveis de serem dinamicamente plugados.

2.1.9 Desenvolvimento Baseado em Componentes

As vantagens do desenvolvimento baseado em componentes (DBC) são bem conhecidas e os seus resultados, já bem evidentes (Gimenes *et al.*, 2000). O DBC oferece abordagens mais flexíveis dos processos de concepção, desenvolvimento, testes e distribuição de *software*, em contraponto à abordagem modular tradicional – aquela onde o sistema é concebido e desenvolvido como um grande módulo principal, composto por módulos menores dentro de uma mesma aplicação. No DBC, ao contrário, a aplicação é "montada" por componentes distintos e de origens diferentes (Gimenes *et al.*, 2000).

Componentes mais complexos e genéricos podem ser projetados não apenas tendo como alvo determinadas tarefas específicas, mas também relações de negócios (por exemplo, componentes de larga granularidade que encapsulem subsistemas). Alguns dos motivos que ajudaram a alavancar o uso do DBC nos processos de negócios são a simplificação, a integração e a transformação. Simplificar significa eliminar os passos que tornem o processo desnecessariamente complicado, e reduzir a variedade desnecessária. A simplificação é obtida através da substituição de componentes. Integração é a transformação de um ou mais processos. Isto pode ser alcançado através da adição de componentes para a criação de novos *links*. A transformação é a criação de um processo totalmente diferente. Neste caso, os componentes seriam desmontados e montados novamente de forma a agilizar toda a operação (Gimenes *et al.*, 2000).

Diversos modelos de sistemas e ambientes baseados em componentes têm surgido nos últimos anos. Vários destes utilizam componentes como suporte para o desenvolvimento de sistemas distribuídos que se comunicam via redes de computadores. Entre os sistemas com essa característica pode-se citar aqueles que disponibilizam serviços pela Internet. Esses serviços podem ser tanto uma simples consulta a um servidor de banco de dados, uma pesquisa em um cadastro, por exemplo, como uma interação entre empresas que cooperam entre si para a realização de um determinado processo de negócios.

Componentes também são utilizados na automação do processo de testes de sistemas. O projeto AGEDIS (AGEDIS, 2003) tem como objetivo utilizar esta funcionalidade para aumentar a eficiência e competitividade da indústria de *software* européia. A idéia consiste do uso de uma metodologia e de ferramentas que automatizem o processo de testes, de

maneira que se alcance um aumento da qualidade do produto final, obtida com a redução de custos. A ênfase desta abordagem está na distribuição de sistemas baseados em componentes.

Outra iniciativa (CARTS, 2002) propõe a construção de uma ferramenta de análise de sistemas de tempo real, baseados em componentes. Segundo o consórcio, a crescente necessidade de se reduzir custos e tempo de desenvolvimento de sistemas, aumentando suas qualidades e desempenho, vem se satisfazendo com aplicações baseadas em componentes e com o extenso reúso de *software*, que vêm cada vez mais sendo consideradas técnicas chave na aquisição destes objetivos.

Existem também abordagens de desenvolvimento de *software* baseado no reúso de componentes de propósito geral já existentes. E ainda aquelas que visam a construção de sistemas bancários que atendam aos seus requisitos com a utilização de um sistema baseado em componentes (BankSEC, 2002). Este sistema visa a definição e implementação de um ambiente gerenciado de *software* para a seleção de componentes, sua montagem e validação no contexto de uma instituição bancária. Aqui também existe a idéia de que utilização de componentes irá proporcionar um menor tempo das transações, redução de custos de desenvolvimento e aumento de qualidade do serviço.

MECASP (2001) é um ambiente de desenvolvimento voltado à manutenção, adaptação e atualização de aplicações compostas, visando aperfeiçoar estas tarefas. Esta proposta, como outras, utiliza o conceito de “componentes” para serviços de *software* (Seção 2.2). ECC (2001) propõe uma arquitetura baseada em componentes para possibilitar a cooperação entre empresas, como o intuito de se formar “empresas virtuais”. A idéia por trás disso é que as pequenas e médias empresas passem a ter as mesmas condições de competitividade, e assim, atinjam uma melhor posição no mercado.

Uma outra abordagem na linha de aplicações compostas baseadas em componentes é proposta por Estublier *et al.* (2001), a *Federação de Softwares*. Esta proposta baseia-se na utilização de componentes *COTS* (Seção 2.1.1). Procura analisar o porquê da reutilização dos *COTS* ainda não ser uma realidade, e apresenta soluções para problemas que ainda existem nesta área com o objetivo de tornar este reúso efetivo e disponibilizá-lo para a contrução de novas aplicações.

A abordagem se baseia na criação de uma unidade chamada Universo Comum. Este universo comum, ou UC, seria um conjunto de instâncias que possuíssem conceitos comuns, compartilhadas por uma federação de *COTS*. Seguindo esta idéia, o UC armazenaria o último estágio conhecido de um sistema descentralizado. Cada componente se encarregaria de atualizar a sua "imagem" neste universo, através de livre e espontânea iniciativa. Estublier *et al.* (2001) procura alcançar um cenário ótimo onde os componentes *COTS* pudessem fazer transações compartilhadas com outros componentes sem se preocupar com a consistência destes, utilizando para isso, os dados comumente conhecidos e disponíveis no UC. Também são detalhados alguns mecanismos para evitar a sobreposição de dados, e uma espécie de *workflow* para controlar as transações. Existe um protótipo industrial que utiliza essa idéia, construído sobre o *Java Message Server* (JMS), e utilizando Enterprise JavaBeans (EJB). De maneira geral esta abordagem propõe conceitos, técnicas, arquitetura e ferramentas bem diferentes daquelas propostas pelas abordagens mais "comuns" de composição de componentes, pois foi pensada exclusivamente para os componentes *COTS*. Segundo Estublier *et al.* (2001), os futuros ambientes de desenvolvimento da Engenharia de Software permitirão que aplicações sejam contruídas através da integração de vários componentes de naturezas distintas, e serão dotados de rígidos requisitos sobre flexibilidade, evolução do sistema e controle, de maneira a evitar ao máximo conflitos nesta integração.

Essas abordagens são interessantes do ponto de vista de apresentar alternativas para a plugagem dinâmica. Entretanto, mostram-se bastante dependentes de estruturas pré-construídas. No capítulo 3 será apresentada uma proposta que não apresenta essas limitações.

2.2 Serviços de Software

Serviços de *software* são pequenas aplicações que possuem interfaces baseadas em mensagens e chamadas enviadas através de uma rede. As arquiteturas baseadas em serviços podem distribuir melhor a lógica de suas aplicações através dos computadores da rede, em vez de tê-las centralizadas em apenas um único deles (Mullender *et al.*, 2002).

2.2.1 Serviços Baseados na Web

Serviços Web (*Web Services*) compõe uma tecnologia integradora, que permite uma programação distribuída de aplicações independentes de plataforma e de linguagem, e que são disponibilizadas, publicadas e localizadas na Web na forma de serviços (Vasudevan, 2001). Serviços Web são também unidades discretas de código, que englobam um conjunto limitado de tarefas, permitem o compartilhamento de dados entre aplicações e geralmente são baseadas na linguagem XML (NETWeb, 2003).

Os serviços Web possibilitam aos seus desenvolvedores escolher entre construir juntas todas as partes das aplicações ou consumí-las através de serviços, criados por eles ou por terceiros. Segundo Macdonald (2002), os serviços Web disponibilizam as mesmas oportunidades de reusabilidade de código que os componentes. Entretanto, diz que os serviços Web são construídos prioritariamente para compartilhar funcionalidade através de diferentes computadores e plataformas espalhados pela Internet. Diz também que os modelos tradicionais de componentes não possuem essa facilidade de utilização em larga escala e via Internet. Por esta razão, não são concorrentes direto dos serviços Web, podendo inclusive, serem combinados com aqueles. Macdonald (2002) acha que os componentes (tradicionais) seriam mais apropriados para a utilização *in-house*, ou seja, que seriam mais eficientes para o compartilhamento de dados entre aplicações de uma mesma empresa ou entre diferentes páginas Web que estejam situadas no mesmo servidor.

Em contrapartida a esta idéia, o Capítulo 3 apresenta uma arquitetura de utilização de *software* onde componentes são utilizados em larga escala como ferramenta principal para que se possibilite o desenvolvimento de aplicações “enxutas” e distribuídas via Internet.

Segundo Sparling (2001), o desenvolvimento da composição de aplicações baseada em serviços Web está inteiramente relacionado com a idéia de se expandir os processos de negócios dentro e fora das organizações, independente da tecnologia utilizada por cada uma delas. Diz também que são descritos por uma linguagem cada vez mais conhecida (WSDL – *Web Service Definition Language*) (WSDL, 2001), utiliza tecnologias-padrão de acesso via Internet (SOAP – *Simple Object Access Protocol*) (SOAP, 2003), geralmente estão catalogados em um diretório de serviço (UDDI – *Universal Description, Discovery and Integration*) (UDDI, 2004) e são utilizados hoje para substituir as antigas chamadas de procedimento remotas (*Remote Procedure Calls* - *RPCs*) por outras baseadas em XML.

Atividades que utilizam comércio eletrônico também se beneficiam do uso de aplicações compostas. Um exemplo é o projeto ACE-GIS (ACE-GIS, 2004), que propõe o desenvolvimento de ferramentas de desenvolvimento, distribuição, descoberta e utilização de serviços Web distribuídos, utilizando como chaves de busca informações geográficas e serviços de comércio eletrônico. O projeto LIVE@WEB.COM (LIVE@WEB.COM, 2000) propõe o desenvolvimento de uma plataforma de comércio eletrônico (*e-commerce*) para a transmissão de vídeo *broadcast* através da Web. A idéia é dotar esta plataforma de eficientes algoritmos de busca por conteúdo através da utilização de serviços, de maneira que os usuários possam utilizá-la para assistir TV através da Internet, com todas as facilidades de procura por trechos de exibição dos vídeos. Pode-se citar também esforços na área de segurança de aplicações Web (HARP, 2000), na área de Web semântica (KWeb, 2004), entre outros.

A Federação de Serviços

A proposta de utilização da *Federação de Serviços* por Camarinha-Matos *et al.* (2001) é um exemplo de aplicação composta que utiliza serviços baseados na Web. Ela é uma abordagem para suportar a interoperabilidade entre entidades autônomas, heterogêneas e geograficamente distribuídas, através da definição de um conjunto de princípios e mecanismos comuns que permitam um acesso harmônico aos serviços e à informação, independentemente da diversidade das origens destes. Seu objetivo geral é a de prover *Serviços de Valor Agregado* (SVA) na indústria do turismo. Conforme Afsarmanesh *et al.* (2000), os SVAs são compostos por outros serviços básicos e / ou por outros serviços de valor agregado. Em outras palavras, serviços complexos podem em

princípio ser construídos no topo de (ou sobre) outros serviços mais simples, enquanto estes últimos podem ser de fato disponibilizados por diferentes provedores.

Camarinha-Matos *et al.* (2001) dizem que o conceito de federação envolve características comuns que servem de suporte para a cooperação entre entidades distribuídas, sem a necessidade de que clientes preocupem-se com detalhes de comunicação. Em outras palavras, procura-se a formação de um conjunto (*cluster*) de provedores de serviços que cooperem entre si com o intuito de oferecer SVAs, através de aplicações compostas. Um *cluster* representa um agrupamento de empresas (provedores), instituições de suporte e afins que possuem um potencial e que irão cooperar umas com as outras através do estabelecimento de um acordo de cooperação. A federação de serviços propõe a criação de *clusters* industriais, utilizando para isso ferramentas e informações avançadas com o intuito de preparar a base para a criação mais rápida de empresas virtuais dinâmicas em respostas às oportunidades do mercado. As entidades envolvidas devem negociar termos e condições de sua interoperação com outras entidades, isto é, definir um contrato de utilização da federação. Quando aplicado ao gerenciamento de um *cluster*, este conceito permite que se considere os vários membros deste *cluster* como provedores de serviços que, independentemente da maneira em que seus serviços estejam implementados ou de onde estejam localizados, tornam-se acessíveis em uma espécie de “mercado virtual controlado” (Camarinha-Matos *et al.*, 2001).

Esta abordagem foi implementada no projeto FETISH (FETISH, 2003). Este projeto reúne diferentes prestadores de serviços da área do turismo, e tem por objetivo agregar valor aos serviços envolvidos no planejamento de uma viagem. A idéia é prover um completo planejamento *on-line* da viagem, de maneira que inclua diversas possibilidades de itinerário, disponibilidade de hotéis, aluguel de carros, passeios turísticos, etc. Os prestadores de serviços, conectados em rede, fazem o papel de provedores. Existe um grande servidor central que cadastra todos os serviços oferecido por cada provedor. Esses serviços estão disponíveis em um servidor Web e são acessados através de navegadores conectados à Internet. Durante um acesso, a federação se encarrega de fazer uma pesquisa no seu cadastro de maneira a encontrar todas as possibilidades de serviço que se encaixam com as expectativas e pré-requisitos definidos pelo usuário do sistema.

Desta forma, estará configurada a utilização de uma aplicação composta. Quem for utilizar este sistema não precisa ter o menor conhecimento da sua infra-estrutura de

comunicação, tampouco da localização geográfica de cada provedor de serviços. Para o usuário basta que o planejamento da sua viagem seja o mais prático e eficaz possível. Ele não precisa saber onde se localiza o servidor de determinada compania aérea. Desde que ela lhe ofereça um bom traslado de ida e volta, ele já se dará por satisfeito. Mas não é apenas o usuário que tem benefícios. O projeto FETISH incluiu entre os seus participantes dezenas de micro-empresas de turismo européias. Como exemplo, pode-se citar os pequenos hotéis familiares, que puderam anunciar os seus serviços de maneira muito mais eficiente e integrada.

2.2.2 Provedores de Serviços de Aplicação

Um Provedor de Serviços de Aplicação (*Application Service Provider – ASP*) ou PSA é um servidor que provê serviços de *software* (geralmente serviços Web) baseados em contrato de utilização, armazenamento e acesso a aplicações, através de um gerenciamento centralizado delas (Dewire, 2002). Por uma determinada quantia periódica, o PSA disponibiliza conteúdo e outros serviços para usuários conectados através da Internet ou de qualquer outra plataforma de rede. Os usuários não precisam se preocupar com detalhes, como novas versões e atualizações de *software*. Os PSAs oferecem acesso a aplicações localizadas fora da área de trabalho de seus clientes. Muitos especialistas acreditam que, com o surgimento da Internet, faria mais sentido disponibilizar *software* como um serviço do que vendê-lo como um produto “fechado em uma caixa” (Stardock Corporation, 2000).

O conceito dos PSAs já vem de muitos anos. Desde o início dos anos 60, a indústria de computadores procura uma maneira de vender as máquinas como serviços. A idéia sempre foi copiar a indústria de utilidades que há tempos adotara um modelo comercial de venda de recursos e serviços. Exemplos que ilustram esse fato são os recursos disponibilizados pelas empresas de eletricidade, água e gás. Já as empresas de telecomunicações oferecem, além de recursos como a largura de banda, outros serviços de valor agregado, como identificação de chamadas, chamadas em espera, rediscagem automática, entre outros. Este modelo comercial compartilha recursos, agrega valor aos serviços, disponibiliza um controle de qualidade e padroniza estes recursos e serviços em geral, conseguindo, com isso, diminuir os custos com a infra-estrutura e assim podendo proporcionar melhores produtos aos usuários (Giotto, 1999) (Network, 2000).

Devido à limitação tecnológica, a indústria da computação sempre adotou o modelo comercial de venda de computadores onde estes são geradores de recursos, licenças e aplicações. Os usuários são responsáveis pela instalação e manutenção da infra-estrutura e das aplicações, tarefas que podem se converter em um penoso trabalho. Este modelo pode prejudicar o desempenho das pequenas e médias empresas em detrimento das maiores, que possuem melhor estrutura interna.

Para a indústria da computação se beneficiar com o modelo de venda de serviços, alguns problemas de infra-estrutura e de tecnologia devem ser superados, e nos últimos anos um enorme avanço nesta direção já foi observado (Giotto, 1999). Primeiramente, os recursos computacionais deveriam ser divididos; assim o tempo de processamento poderia ser compartilhado por um largo número de usuários. Segundo, deveria haver um extenso sistema de entrega para os serviços. Terceiro, as aplicações deveriam ser desenhadas para suportar hospedagem em servidores, pois esta seria a melhor maneira de manipular as aplicações. Recentemente, com o aparecimento dos provedores de serviços de aplicação, iniciou-se uma nova onda no desenvolvimento da Internet: o seu uso, ou de uma WAN (*Wide Area Network*), para disponibilizar serviços de aplicações *on-line* baseado em aluguel de *software*.

Os PSAs originaram-se de três tendências distintas (Booth, 2002):

- Indústria de Serviços em Tecnologia da Informação (TI): Tem se procupado nos últimos anos com o nível de granularidade oferecidos aos clientes. Em vez de entregar a sua completa estrutura de TI a um provedor externo, as organizações tem selecionado partes específicas da TI e as distribuído, utilizando redes de dados para gerenciar as aplicações. Isto tem sido combinado com a tendência pague-por-uso (*pay-per-use*), frequentemente disponibilizada na forma de uma assinatura mensal. A Figura 2.7 mostra um sistema operacional que foi disponibilizado via um PSA, o WebOS (WebOS, 1998). Na Figura 2.8 é mostrado o I-Dox (Cylex, 2001), que é um disco virtual, que aluga armazenamento de dados e também permite a seus clientes que executem aplicações remotamente para manipular esses dados como, por exemplo, um editor onde possa alterar seus textos.

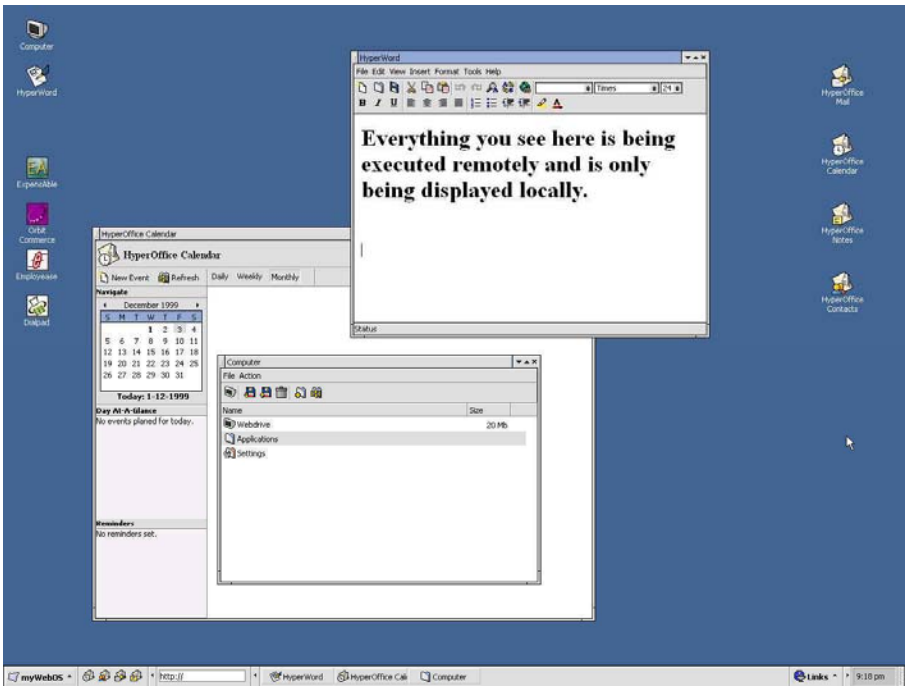

Figura 2.7 – WebOS (WebOS, 1998)

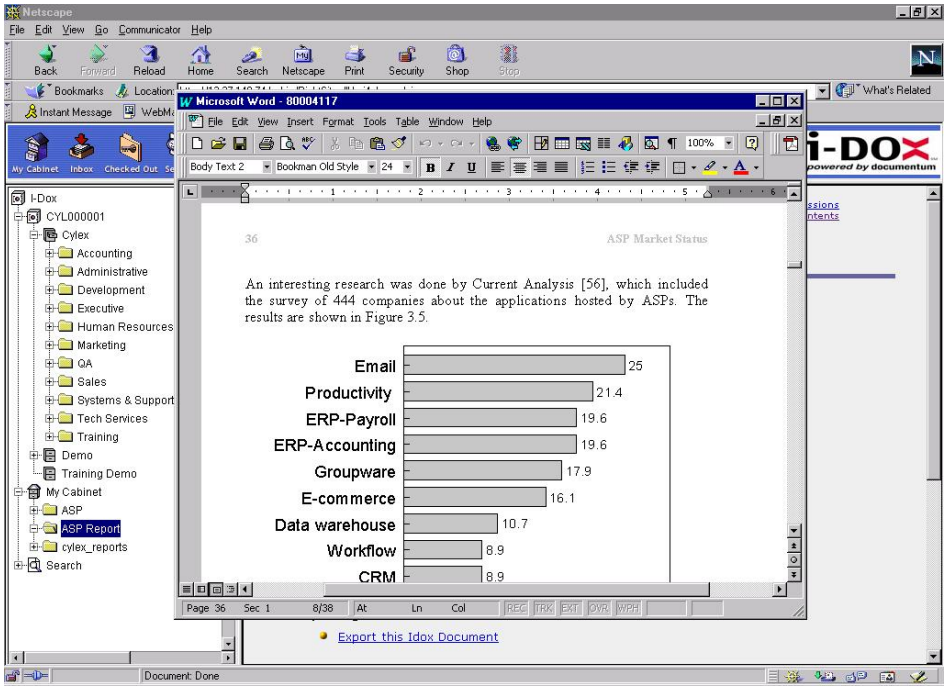

Figura 2.8 – I-Dox (Cylex, 2001)

- Provedores de Serviço de Internet (*Internet Service Providers – ISPs*): Os provedores de Internet podem ser considerados PSAs se se levar em conta que a hospedagem do correio eletrônico e os servidores Web são serviços de aplicação. Neste contexto, comércio eletrônico (*e-commerce*), trocas de mensagens

(*messaging*), e outros complexos serviços hospedados pela Web, podem ser considerados como partes integrantes de um PSA. A Figura 2.9 apresenta um PSA que disponibiliza correio eletrônico (*webmail*) (DAS, 2004).

Figura 2.9 – Webmail (DAS, 2004)

- Empresas Baseadas na Internet: Web *sites* surgiram na Internet como destinos que ofereciam somente conteúdo estático, com poucas palavras e algumas imagens. Hoje, os usuários têm a sua disposição aplicações inteiras. Para garantir que os usuários voltarão a visitá-los, os *sites* adicionaram aplicações que possibilitam experiências dinâmicas e interativas. Entretanto, uma nova geração de desenvolvedores de *software* está adaptando suas aplicações para o mercado de serviços baseados na Web, acessados diretamente através da Internet. Essas tendências convergem servindo necessidades empresariais especializadas ou mercados verticais da indústria. Este é o advento da computação baseada em navegador. Mas apesar de ser a forma mais conhecida e difundida, os serviços Web não necessitam necessariamente ter um navegador como cliente.

Existe uma estratificação da arquitetura dos PSAs em quatro camadas interligadas (Figura 2.10), cada uma responsável por um núcleo de atuação (Booth, 2002).

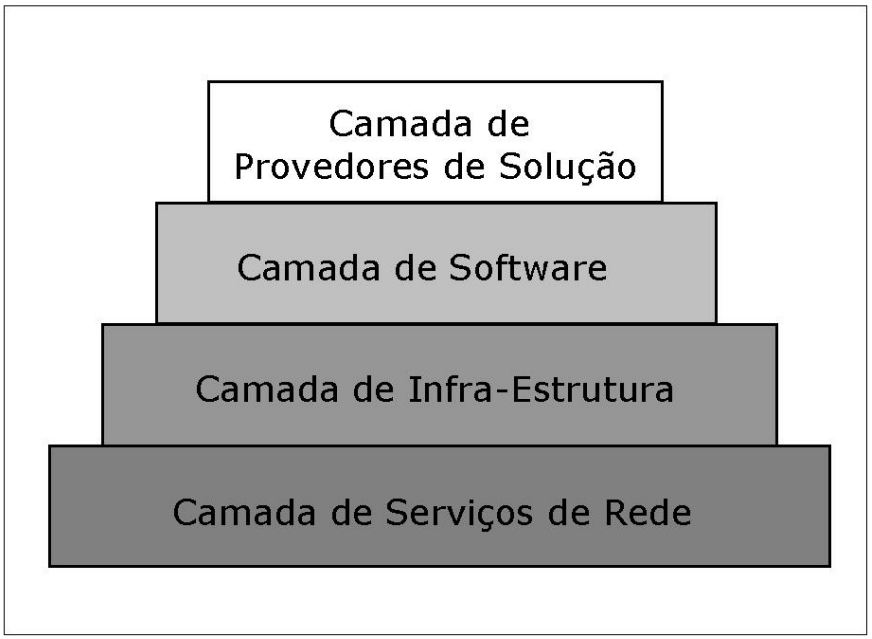

Figura 2.10 – Estratificação da Arquitetura PSA

i. Camada de Serviços de Rede: Disponibiliza os serviços básicos de comunicação, os recursos centrais do servidor e os serviços de IP (*Internet Protocol*). Inclui a conexão física, os roteadores que manipulam o tráfego IP e o seu desempenho associado, aplicações de confiabilidade e segurança. Os recursos centrais englobam a alocação de espaço, materiais de proteção para eletricidade, segurança física e os serviços de manutenção. Os serviços de IP incluem a rede privada virtual (*Virtual Private Network – VPN*), *cache* de rede, *firewalls*, *streaming* de mídia e serviços de diretórios.

ii. Camada de Infra-Estrutura: Inclui a coordenação da rede e sistemas de gerenciamento, a provisão da demanda, operação e gerenciamento dos sistemas de *hardware* e *software*, o gerenciamento das assinaturas PSA, faturas e o suporte para os clientes. Esta camada é responsável pela monitoração do nível de serviço, pelas mensagens de alerta do sistema e pela distribuição de informação de ajuda entre os parceiros conectados ao canal PSA.

iii. Camada de Software: Disponibiliza o núcleo vital do serviço de aplicação. O *software* pode ser uma aplicação comum que foi adaptada para o modelo PSA, ou pode ter sido especialmente desenvolvida com este propósito. Há um grande número de plataformas de servidores de aplicação apropriadas para a criação de

um PSA, embora apenas algumas ofereçam um conjunto completo de funcionalidades como a disponibilização dos serviços propriamente ditos, gerenciamento dos assinantes e das faturas, suporte e controle de qualidade. Apesar da maioria do desenvolvimento, ainda hoje, ser feito *in-house* por desenvolvedores independentes, um crescente número de companias e integradores de sistemas estão começando a projetar e construir serviços de aplicação que podem ser acessados *on-line*.

iv. Camada de Provedores de Soluções: Esta camada cumpre o passo final desta estratificação. Ela é o núcleo do Provedor de Serviço de Aplicação. É ela quem empacota o *software* e os ingredientes de infra-estrutura juntos com o negócio e os serviços profissionais para criar um produto de serviço completo e apresentá-lo ao usuário final.

Os benefícios dos PSAs derivam do fato de as aplicações estarem sendo distribuídas através de vários servidores, e não mais entre vários clientes. Os benefícios para os desenvolvedores e para os provedores incluem (Sweeney, 1999) (Brain, 2002):

- Custo Zero de Distribuição: Não há necessidade de se imprimir manuais, gravar discos de instalação, produzir caixas para empacotar o produto, gerenciar e distribuir o estoque, entre outros.

- Maior Proteção de Direitos Autorais: Os usuários geralmente não baixam o *software*, logo, não podem copiá-lo.

- Atualizações Instantâneas: Os provedores implementam correções de problemas (*bugs*) e novas características (*features*) sem precisar notificar os usuários e nem esperá-los baixar o novo código de instalação.

- Base de Usuários Consistente: A proliferação de diferentes versões e níveis de distribuição de *software* na base de usuários é quase totalmente reduzida ou completamente eliminada. Ou seja, procura-se certificar de que cada vez mais um número maior de usuários estará usando a mesma versão de uma aplicação.

- Monitoração de Uso: Provedores podem monitorar o uso e adquirir valiosas informações sobre a interação dos clientes com os seus produtos. Pode-se

descobrir quais características são mais populares, quais causam mais problemas, quais precisam ser melhoradas, etc.

- Potencial Fluxo de Renda Constante: Os provedores não necessitam lançar novas distribuições do *software* a cada ano para manter estável o fluxo de vendas do produto e assim manter a lucratividade do negócio. Para alcançar isto, basta manter um usuário estável e satisfeito com a qualidade do serviço.

Os benefícios para os usuários incluem (Sweeney, 1999) (Brain, 2002):

- Previsibilidade de custos: Em um mercado PSA estável, fica mais fácil prever quanto se gastará com os serviços utilizados e assim poder se programar melhor.
- Modelo de solução um-para-muitos (*one-to-many*): Soluções reutilizáveis e maior produtividade da equipe de apoio, devido à menor ocorrência de viagens e maior familiaridade desta com a infra-estrutura do sistema.
- Baixos custos de um mercado PSA maduro: Nos estágios iniciais, a falta de padronização de formatos de dados e de APIs dificulta o desempenho dos novos PSAs, pois cada um será responsável pelo seu próprio padrão.
- Uma expansão do mercado da tecnologia da informação: Por causa dos seus potenciais baixos preços, os PSAs têm a oportunidade de expandir o mercado deste tipo de *software* para pequenas e médias empresas. Isto, é claro, vai depender também dos custos que essas mesmas empresas precisem arcar com as infra-estruturas de rede utilizadas, por exemplo, os tempos de execução, que são dependentes da carga da rede.
- Capacidade de vender com alta margem de chance serviços com valor agregado na base de clientes: PSAs terão uma base potencialmente cativa de clientes para vender serviços adicionais.

Apesar de ser um bom modelo, apresenta algumas limitações relevantes quando projetado em um cenário de completa flexibilidade funcional: o seu processamento é lógica e fisicamente centralizado (o “componente” é executado no PSA); a granularidade de seus módulos é grande; e eles geralmente não são adaptados às necessidades de cada cliente.

Outras desvantagens incluem as suas dificuldades de oferecer segurança aos dados dos clientes e as suas limitações de desempenho devido à limitada largura de banda da Internet.

2.2.3 Arquitetura Orientada a Serviços

A Arquitetura Orientada a Serviços ou AOS (*Service-Oriented Architecture – SOA*) (SOA, 2004a) (SOA, 2004b) é uma abordagem de projeto para aplicações cliente-servidor na qual estas aplicações consistem de serviços de *software* (*software services*) e de consumidores de serviços (*service consumers*), também conhecidos como clientes. A AOS difere do modelo geral de cliente-servidor fundamentalmente por causa da sua ênfase na Acoplagem Fraca (*Loose Coupling*) entre componentes de *software* (serviços), pelo uso destes através de suas interfaces. É um padrão de arquitetura para sistematizar o *design* de aplicações do tipo *request-reply*. Seu propósito é obter modularidade de *software* no nível de negócios e possibilitar um intenso reuso deste *software* em novos contextos e ambientes de execução (Natis *et al.*, 2003) (Dietzen, 2004) (BEA 2004).

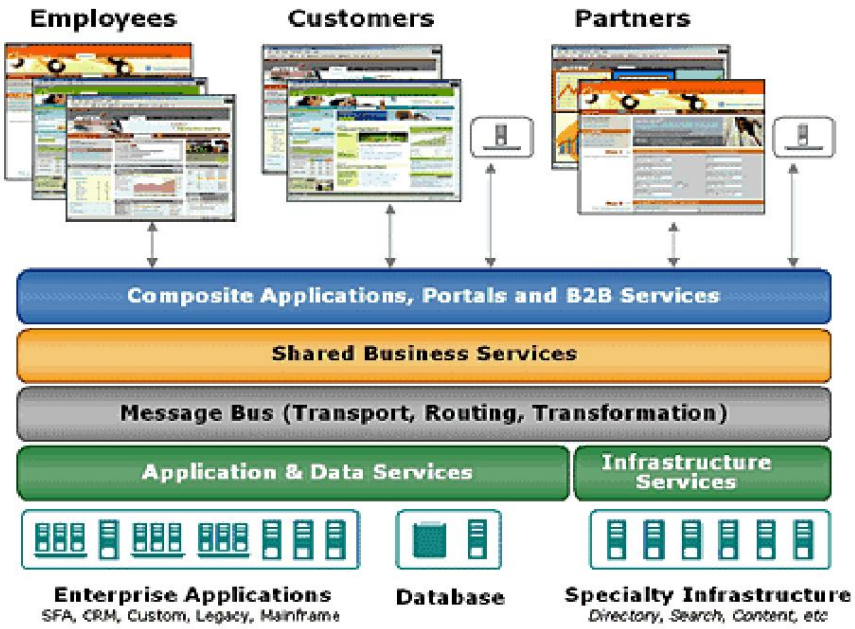

Figura 2.11 – A Plataforma AOS (BEA, 2004)

A Figura 2.11 mostra a estrutura da plataforma AOS. A AOS é concebida para compartilhar serviços entre uma empresa (*Employees*), consumidores (*Customers*) e

empresas parceiras (*Partners*). A interface de usuário ou *front-end* (camada *Composite Applications, Portals and B2B Services*) compreende aplicações compostas, portais de acesso e de serviços *Business-to-Business* (B2B). A camada imediatamente inferior – *Shared Business Services* – contem as interfaces *back-end* para as aplicações que requisitam dados e / ou funcionalidade dos serviços. A próxima camada – *Message Bus* – possui o núcleo dos serviços de integração: troca de mensagens, transporte, coordenação (*workflow*), gerenciamento contratual (*brokering*), entre outros. A camada *Application & Data Services* oferece acesso aos serviços propriamente ditos e aos dados compartilhados. Os serviços de infra-estrutura (camada *Infrastructure Services*) oferecem funcionalidades adicionais de suporte à plataforma, como por exemplo, busca por conteúdo.

Embora a completa definição de AOS possa variar de artigo a artigo e de desenvolvedor a desenvolvedor, um termo que é usado frequentemente é “fracamente acoplado” (*loosely coupled*) (Wilkes, 2004a). Quando utilizados corretamente, sistemas fracamente acoplados disponibilizam muitos benefícios tecnológicos e de negócios, como por exemplo, robustez, capacidade de manutenção, de adaptação, de recuperação, escalabilidade e independência de plataforma (Hocker, 2004c).

Interfaces de Serviço

Uma interface de serviço AOS é um contrato que estabelece a identidade do serviço e as regras de invocação deste serviço (Wilkes, 2004a). A essência da AOS baseia-se na utilização de serviços e estes, por sua vez, são definidos em uma especificação. Esta especificação não deve conter ambiguidades e é um contrato de interface independente de linguagem. Uma interface de serviços pode, por exemplo, através da *Web Services Description Language* (WSDL) (WSDL, 2001) (Wilkes, 2004b), definir operações que o serviço disponibiliza e os dados necessários para que cada uma destas operações seja executada. A interface de serviço deve conter informação suficiente para que um serviço seja identificado (isto é, localizado e interpretado) e usado, sem se considerar o seu projeto e conteúdo interno.

Segundo Wilkes (2004a), essa natureza de caixa preta é essencial para o serviço. Isto significa que todos os dados de entrada, todos os dados de resposta (caso haja algum) e todas as condições de exceção devem estar detalhados na interface. Significa também que *metadados* suficientes devem ser disponibilizados na interface para identificar o propósito

e a função do serviço. Um repositório consultável de definições (especificações) de serviços pode suportar um grande número de serviços com baixa granularidade e provindos das mais diversas fontes (BEA, 2004) (Dietzen, 2004).

A implementação do serviço fica em segundo plano na AOS. A prioridade é dada à interface deste. Portanto, a AOS é essencialmente um fluxo de relacionamentos e interações entre interfaces de serviços (Natis *et al.*, 2003). Seu *design* envolve a definição das interfaces de serviço assim como as suas interações com outras interfaces e com os consumidores de serviços.

Projeto, Implementação e Execução

Os princípios da AOS diferem durante as fases de projeto da aplicação, desenvolvimento e execução (Natis *et al.*, 2003). Esses princípios compartilham o essencial do encapsulamento e do acoplamento fraco, mas se diferenciam em alguns detalhes. A intenção principal da AOS é o reúso não-intruso (*nonintrusive*), ou seja, aquele que abstrai a implementação interna do serviço, em novos contextos de execução. O projeto e o desenvolvimento da AOS é executado com o propósito de se alcançar agilidade e flexibilidade em tal ambiente.

Na fase de projeto, um serviço é um componente de *software* (de negócios) composto por dois elementos: a interface e a implementação do serviço. Enquanto a interface do serviço define um contrato programático de acesso a ele, a implementação cumpre a funcionalidade propriamente dita. Dependendo do nível de detalhe do projeto, a implementação do serviço pode ser concebida como uma abstração, a ser conhecida somente na fase de desenvolvimento. Em outros casos, pode ser desenhada como um novo módulo, que encapsula acesso a um módulo anterior (módulo pré-AOS), ou ainda como a composição de chamadas múltiplas e condicionais para módulos de *software* novos e pré-existentes.

Na fase de projeto, o fraco acoplamento significa que os serviços são desenhados sem afinidade com nenhum consumidor em particular. Desta forma, um serviço é completamente desacoplado de qualquer consumidor de serviços. Entretanto, o mesmo não acontece com o consumidor, que é dependente do serviço porque engloba referências às suas interfaces. Assim, pode-se dizer que a AOS é uma arquitetura semi-acoplada.

Na fase de desenvolvimento, uma interface de serviço é implementada como um arquivo de definição independente (Wilkes, 2004b). Os arquivos de interface podem utilizar qualquer sintaxe que seja previamente acordada, mas uma sintaxe baseada em padrões possibilita uma utilização do serviço numa escala bem maior. Quando a WSDL é usada para codificar as interfaces de um serviço, este serviço é um serviço Web (WebServices, 2004). A WSDL é uma linguagem-padrão utilizado para a definição de interfaces na maioria das novas ferramentas de desenvolvimento AOS. Segundo Natis *et al.* (2003), antes da WSDL, a AOS era baseada em linguagens como a CORBA *Interface Definition Language* (IDL) (OMG, 1992) (OMG, 2004c), a COM/DCOM Microsoft IDL (MIDL) (MIDL, 2004), o *Customer Information Control System* (CICS) (CICS, 2001), entre outras.

O desenvolvimento de ferramentas orientadas à AOS volta o seu foco para a definição, descoberta e interconexão de interfaces de serviços (Hocker, 2004c). Na fase de desenvolvimento a implementação de um serviço pode ser um novo programa, um encapsulador (*wrapper*) que se conecta através de um adaptador (*plug*) a uma aplicação antiga, ou ainda, a composição dos dois. O desenvolvimento de implementações de serviço pode ser simplesmente um novo projeto de programação, uma tentativa de se adaptar sistemas legados ou uma integração de aplicações. Cada um destes exige diferentes ferramentas e diferentes habilidades do desenvolvedor.

O fraco acoplamento na fase de desenvolvimento envolve implementações de serviços que podem ser invocadas em múltiplos contextos. Os desenvolvedores de implementações de serviço devem entender que a essência de um serviço é o seu potencial de ser reusado em novos contextos. A implementação de um serviço não deverá assumir detalhes a respeito das tecnologias ou da lógica de negócios dos clientes. Quanto menos isolado for o projeto interno de um serviço, mais custoso será utilizá-lo (Natis *et al.*, 2003). O isolamento de um serviço potencializa seu reúso, porém aumenta a sua complexidade. Na maioria dos casos, a acoplagem de serviços requer tecnologias adicionais de integração para resolver as diferenças existentes entre eles e para gerenciar o contexto comum e o tratamento de falhas.

Na fase de execução, uma interface de serviço é implementada como um par de programas: *stub* e *proxy* da interface de serviço. O *stub* e o *proxy* são tipicamente gerados por uma ferramenta de desenvolvimento a partir do arquivo de definições de interface, mas

também podem ser desenvolvidos manualmente e integrados ao *software* durante a execução. Tanto o *stub* quanto o *proxy* interagem com a implementação local (seja do serviço no caso do *stub*, seja do consumidor – *service consumer* – no caso do *proxy*) através de métodos de comunicação locais e da codificação de dados previamente estabelecida (Hocker, 2004b).

De acordo com Hritz (2004), teoricamente, o par gerado de *stub* e *proxy* pode utilizar XML (Wilkes, 2004c) (Orchard, 2004), mensagens SOAP, HTTP, *Middleware* Orientado a Mensagens (*Message-Oriented Middleware – MOM*) (MOM, 1997) ou memória compartilhada para passar os parâmetros da interface. Esta comunicação é privada mas, para preservar a independência das ferramentas, na maioria das vezes, protocolos-padrão são utilizados. Na realidade, entretanto, esses protocolos não garantem interoperabilidade automática. Para otimizar o desempenho também é necessário o uso de protocolos de comunicação não-padrão e mais flexíveis (Hocker, 2004a).

Tecnicamente falando, na fase de execução, o fraco acoplamento é obtido através de comunicação *sessionless*, ou assíncrona (Natis *et al.*, 2003). Porém, também podem ser utilizadas tecnologias fortemente acopladas, como o JAVA *Remote Method Invocation* (JAVA RMI) (JavaRMI, 2004) ou o Microsoft .NET *Remoting* (NETRemoting, 2004). Contudo, se a implementação interna dos serviços for logicamente fracamente acoplada, então a agilidade da AOS está garantida em tempo de execução, apesar do uso de *middlewares* fortemente acoplados (Natis *et al.*, 2003).

A AOS é útil especialmente em aplicações onde a comunicação *request-reply* seja a tônica do relacionamento entre os serviços (Hritz, 2004). Todavia, a AOS exige esforços adicionais de projeto e de desenvolvimento e, ainda, que alguns requisitos de *middleware* sejam atendidos.

XML e *Web Services* na AOS

XML é uma tecnologia interessante de ser usada para o fraco acoplamento inter-aplicação por ser: auto-descritivo; independente de *hardware*, linguagem de programação, *container*, etc.; adapta-se bem a mudanças de versões; é uma espécie de denominador comum, assim como o HTML (Dietzen, 2004).

XML e serviços Web são nas tecnologias-base da AOS, ambas voltadas para a comunicação intra e inter-empresas (Dietzen, 2004) (Hritz, 2004). A proposta original da AOS é desacoplar os serviços de negócios compartilhados da atual estrutura monolítica e de aplicações isoladas na qual se encontram. Objetos e componentes têm obtido sucesso em prover reúso de módulos intra-aplicações. A AOS, no entanto, procura alcançar este reúso entre aplicações (BEA, 2004).

O que torna a integração de aplicações tão difícil e, consequentemente, faz a indústria pensar em soluções no estilo AOS, é que as aplicações são construídas por diferentes grupos de pessoas, em diferentes lugares e que utilizam diferentes tipos de planejamento. Qualquer abordagem que dependa de múltiplas aplicações que compartilham objetos comuns e modelos de dados vivenciam esta realidade (BEA, 2004).

WebLogic

O BEA WebLogic foi projetado para ser o primeiro framework a tornar a programação J2EE mais dinâmica quanto à utilização, assim como o PowerBuilder ou o Visual Basic.

Assim que foi lançada a sua primeira versão, em 2002, o WebLogic tornou-se o primeiro ambiente de desenvolvimento integrado (*Integrated Development Environment* – IDE) Java especialmente destinado à AOS, utilizando XML e serviços Web (Dietzen, 2004). Na última versão, lançada em 2003, este produto transformou-se num ambiente de desenvolvimento para autenticação, utilização e coordenação de aplicações baseadas em AOS. O WebLogic possibilita a implementação de uma variedade de códigos orientados à AOS, incluindo simples aplicações *Web*, programas J2EE (J2EE, 2004), portais, automação do processo de negócios, agregação XML (Wilkes, 2004c) (Orchard, 2004), coordenação de mensagens, etc (Hritz, 2004).

Segundo Dietzen (2004), existe um forte precedente de que novos padrões agregam mais tecnologias já existentes e bem testadas do que inovações propriamente ditas. Assim sendo, o WebLogic se baseia no uso de TCP/IP, SQL, da *Web*, de Java, XML e serviços Web. Também introduziu *metadados* para tornar a autenticação de serviços tão simples quanto a escrita de um pequeno objeto Java.

WebSphere

A plataforma IBM WebSphere (IBM, 2004a) é um *software* de infra-estrutura para a Internet (*middleware*) que possibilita a empresas implantar e integrar aplicações *e-business*, do tipo comércio eletrônico B2B, com o intuito de gerar aplicações de negócios distribuídas pela Web (IBM, 2004b) e com um alto grau de customização destas (MobileInfo, 2001).

A plataforma WebSphere é composta por quatro componentes (MobileInfo, 2001):

a) Base: Consiste de servidores de aplicação Web e de servidores de transação;

b) Serviços Básicos: Serviços de desenvolvimento, TI e a estrutura dos *websites*;

c) Serviços de Aplicação: Funções de colaboração, comércio eletrônico e serviços B2B;

d) Aplicações Parceiras: Aplicações desenvolvidas por empresas participantes da transação e-business.

A idéia é criar implementações modulares dos processos de maneira a dispô-los pela Web, sem desfazê-los, e ao mesmo tempo prover segurança, confiabilidade e escalabilidade. De acordo com IBM (2004b), formam a base da plataforma WebSphere o Servidor de Aplicação WebSphere (*WebSphere Application Server*), que consiste de um servidor para aplicações baseadas em Web construído sobre a plataforma *Java 2 Enterprise Edition* (J2EE, 2004); e o Servidor de Transações CICS (*CICS Transaction Server*), que é a base da maioria das aplicações de *mainframes* atuais e suporta o desenvolvimento em diversas linguagens de programação, pois especifica os seus serviços através das ferramentas CICS (CICS, 2001).

Para o desenvolvimento de aplicações voltadas para a plataforma WebSphere, a IBM disponibilizou o WebSphere Studio, que é um ambiente de desenvolvimento, multi-linguagem, multi-plataforma e multi-dispositivo para a construção, teste e implantação de aplicações dinâmicas (IBM, 2004c).

Ican Suite

O *Integrated Composite Applications Networks (ICAN) Suite* (SeeBeyond, 2004b) é a aposta da SeeBeyond para competir no mercado da AOS. Assim como seus similares, a

idéia por trás do ICAN Suite é suportar serviços Web baseados em padrões para a automação de processos de negócios e além de melhor compatibilidade com outros *middlewares* (LaMonica, 2003). Assim como seus pares, o ICAN Suite também se baseia no uso da linguagem Java.

A última versão disponibilizada pela SeeBeyond, o ICAN Suite 5.0, agrega uma série de aplicações para suportar esta plataforma, entre as quais se pode destacar (SeeBeyond, 2004b):

- eGate Integrator 5.0: Distribui e integra aplicações baseadas em serviços Web, oferecendo garantias de transações, intercâmbio de dados e serviços de mensagens; e executa sobre a plataforma J2EE;
- eInsight Business Process Manager 5.0: Automatiza e gerencia os processos de negócios e os *workflows* dos serviços Web;
- eInsight Business Process Manager: Coordena os serviços Web, através da utilização da UDDI (UDDI, 2004);
- eVision Studio 5.0: Possibilita a criação de aplicações compostas por usuários finais.

Granularidade

A granularidade (Capítulo 1) de um serviço AOS possui um grande impacto no grau de acoplamento entre o serviço e seus consumidores. Um serviço que requer muitas operações em determinada sequência para ser executado é mais fortemente acoplado do que um serviço que possui menos operações complexas (BEA, 2004). Isto é passível de se observar na comparação com esquemas de integração que utilizam também RPCs (*Remote Procedure Calls*), como por exemplo CORBA, EJB, COM+, objetos, etc. Se propriedades adicionais são requisitadas ou alguma alteração no objeto é efetuada, todos os consumidores (clientes) deverão executar alterações substanciais em seus códigos. (Wilkes, 2004a).

Localização de Serviços na AOS

A localização de um serviço é feita através da publicação e assinatura do serviço ou de mensagens *broadcast*. No início da utilização dos serviços Web, a maioria dos serviços era codificada via RPC e SOAP. Recentemente, a tendência tem sido na direção de se encorajar a utilização de serviços baseados em mensagens, que utilizem autenticação via documentos (BEA, 2004). Existem vários padrões para a localização e invocação de serviços, bem como para os protocolos de comunicação, essenciais para a fraca acoplagem. A seguir serão apresentadas algumas abordagens para a localização de serviços AOS (Wilkes, 2004a):

- Registro de Serviços e Associação Estática (*Service Registry and Static Binding*): Uma abordagem inicial para o problema da localização de serviços foi a introdução de um registro de serviços. Segundo SeeBeyond (2004a), quando os serviços Web tornaram-se mais populares, as principais especificações de suporte eram o SOAP (*Simple Object Access Protocol*) (SOAP, 2003), a WSDL (*Web Service Definition Language*) (WSDL, 2001) e a UDDI (*Universal Description, Discovery and Integration*) (UDDI, 2004). O SOAP é um protocolo baseado em XML para trocar informações em um ambiente de execução descentralizado e distribuído. A WSDL é uma especificação baseada em XML para descrever um serviço Web. A noção por trás de um registro UDDI é a de que serviços podem publicar informações sobre suas interfaces e implementações de maneira que os consumidores possam descobrí-las. Apesar de serem, inicialmente, em sua maioria repositórios globais replicados, atualmente registros privados UDDI são recomendados para gerenciar serviços internos das empresas (BEA, 2004). O modelo mais simples é voltado para a fase de projeto somente. Assim, um provedor de serviços já cria e publica informações a respeito de suas interfaces (em WSDL). O consumidor descobre o serviço de interesse, usa o WSDL para criar uma invocação específica para ele e o associa (*bind*) estaticamente. Porém, apesar dessa facilidade, este modelo estático não é fracamente acoplado.

- Registro de Serviços e Associação Dinâmica (*Service Registry and Dynamic Binding*): A UDDI possibilita que múltiplos provedores implementem a mesma interface de serviço. Isso permite uma abordagem mais fracamente acoplada tendo-se em conta que

é menos suscetível a falhas por causa de sua natureza mais balanceada (*load balacing*) (Beatty, 2004). Nesta abordagem, o serviço é descoberto apenas em tempo de execução. Isto gera alguns problemas quanto ao desempenho do sistema. Uma abordagem alternativa é utilizar a busca no registro somente quando o serviço original falhar pois, desta forma, o sistema ficaria menos sujeito a falhas. O uso de um registro provê um mecanismo de associação (*binding*) direta. Ou seja, o consumidor usa o registro para localizar o serviço e depois o conecta diretamente, baseado na informação existente no registro. Este esquema já possui um bom grau de acoplamento fraco, pois os consumidores utilizam interfaces de serviços sem conhecimento prévio de quem as implementou.

- Roteador de Serviços (*Service Broker / Router*): Nesta abordagem, em vez de acessar o serviço diretamente do provedor, o cliente o acessa através de um negociador (*broker*) ou roteador (*router*). Isto provê fraca acoplagem através da quebra da conexão direta entre consumidor e provedor. Em linhas gerais, a tarefa do *broker* é localizar um provedor que contenha o serviço requisitado e rotear a invocação do cliente até ele. Esquemas mais complexos podem incorporar filtragem e busca por conteúdo, regras de roteamento, tratamento de falhas, recuperação de transações, entre outros. Este modelo é recomendável para a segurança, controle de dados e registros de transações (*logs*), já que todas as invocações passam pelo *broker*. Entretanto é desvantajoso quando se verifica que toda a comunicação se concentra em um único ponto de falha (o roteador) e que, além disso, pode se transformar em um gargalo em sistemas com grande fluxo de invocação de serviços (Sengupta, 2004).

- Roteador de Serviços Distribuído (*Distributed Service Broker / Router*): Esta abordagem consiste na introdução de pequenos *brokers* locais, em cada provedor e consumidor do sistema. Neste caso, esses *brokers* locais se comunicam com o *broker* central, com o intuito de replicar os dados já registrados e de lhe enviar detalhes dos serviços presentes localmente. Também podem servir como parâmetro para o *broker* central, indicando se determinado serviço está *on-line* ou não. Em vez do tráfico fluir todo para um único *broker*, ele se torna *peer-to-peer* entre *brokers* locais cada vez que um serviço é localizado. Neste modelo, o *broker* central é relegado à coordenação das informações de serviços entre os nós (*brokers* locais). Se o *broker* central falhar, ainda

assim os *brokers* locais serão capazes de localizar e de invocar os serviços. Se algum provedor falhar, essa informação será repassada ao *broker* local, que então optará por outra alternativa. Apesar desta abordagem requerer dos *brokers* locais um forte acoplamento em relação à plataforma tecnológica na qual provedores e consumidores se encontram, os consumidores propriamente ditos não precisam se preocupar com o provedor de determinado serviço, assim como os provedores não sabem quem está consumindo seus serviços (BEA, 2004).

- Barramento de Serviços (*Service Bus*): Uma característica em comum que torna as abordagens anteriores fortemente acopladas é a necessidade que existe de que os serviços estejam disponíveis no mesmo momento em que os consumidores os requisitam. Isto pode ser atenuado através de um mecanismo público de assinaturas. O princípio básico deste mecanismo é que os provedores assinam e os consumidores publicam suas requisições a um barramento de serviços (Wilkes, 2004a). Este barramento é uma extensão de um *broker* que provê armazenamento e capacidade de redirecionamento de mensagens, assegurando-se que todos os eventuais provedores interessados irão recebê-las. Dentro de cada requisição publicada está uma informação (possivelmente um endereço de um serviço Web) que indica como o provedor poderá contactar o consumidor (se for o caso). Quando um provedor assina o barramento, indica quais as mensagens em que possui interesse. Após a publicação de uma requisição por algum consumidor, somente as partes interessadas são notificadas. Qualquer resposta é enviada via barramento para o consumidor. A vantagem aqui sobre o *broker* é que o provedor não precisa estar ativo no momento da requisição (Wilkes, 2004a).

- Barramento de Serviços Distribuído (*Distributed Service Bus*): Similar à do roteador de serviços distribuído, esta abordagem adiciona barramentos locais em cada plataforma de serviço. Esses barramentos locais irão interagir com um barramento central, e redirecionar as mensagens a seus devidos destinos. Este cenário provê o mais baixo nível de acoplamento da AOS (Wilkes, 2004a). As mensagens podem ser postadas assincronamente no barramento. As partes interessadas aceitam as mensagens, as processam e retornam as respostas (se necessário) para o barramento. Os consumidores

não conhecem os provedores e vice-versa, e outros barramentos podem se conectar ao barramento central, externos ao ambiente da empresa ou não.

De maneira geral pode-se dizer que a abordagem da AOS preza pela integração de sistemas e pelo reúso de componentes de *software* na forma de serviços. Sua proposta de fraca acoplagem vai ao encontro dos recentes esforços no sentido de se obter modelos que suportem operações dinâmicas e em tempo de execução. Por operações dinâmicas entenda-se associação (*binding*), especialização, transparência de utilização e escalabilidade entre os módulos (de qualquer granularidade). A AOS é potencialmente interessante para se implementar modelos de negócios *end-to-end* que envolvam vários parceiros. Além do mais, o surgimento de plataformas de desenvolvimento para a AOS, tais como o WebLogic (Dietzen, 2004) (Hritz, 2004), o WebSphere (IBM, 2004a) e o Ican Suite (SeeBeyond, 2004b), proporcionou a grandes empresas (BEA, 2004) bons ganhos de qualidade e de produtividade em seus sistemas.

A proposta apresentada nesta dissertação se assemelha em vários pontos com a AOS. No próximo capítulo essas semelhanças serão mostradas, bem como os pontos divergentes entre as duas abordagens.

2.3 Peer-to-Peer

Uma característica será bastante importante à composição dinâmica: a descentralização. Através dela, pode-se montar uma rede de comunicação mais eficaz para o propósito da plugagem no sentido de que, tanto os clientes, quanto os PAs, poderão ser desligados e / ou reconectados ao sistema sem a intervenção de um servidor central. Isso é uma grande vantagem para um sistema que se projeta em larga escala e geograficamente distribuído. A abordagem *peer-to-peer* torna essa descentralização possível. O *peer-to-peer* (P2P) consiste de uma arquitetura onde o compartilhamento de recursos e serviços é feito diretamente entre os sistemas envolvidos sem a intervenção de um servidor central (Parameswaran *et al.*, 2001). O termo *peer-to-peer* se refere à classe de sistemas e aplicações que utilizam recursos distribuídos com o objetivo de executar uma funcionalidade de maneira descentralizada (Milojicic *et al.*, 2002). Segundo P2P (2004), P2P significa um modelo de comunicação no qual todas as partes têm as mesmas potencialidades e qualquer uma delas está apta a iniciar uma sessão de comunicação. Mais recentemente, o P2P tem sido usado como arquitetura para aplicações nas quais os usuários podem usar a Internet para trocar arquivos com outros usuários, diretamente ou através de um servidor mediador. O P2P se propõe a ser um contraponto à arquitetura centralizada cliente-servidor.

O modelo cliente-servidor descreve o relacionamento entre dois programas de computador dos quais um deles, o cliente, efetua uma requisição de serviço para ser executada no outro, o servidor (ClienteServidor, 2004). Embora esta abordagem possa ser aplicada a uma única máquina, torna-se uma idéia mais atraente quando é aplicada em rede, pois provê uma maneira rápida e simples de interconectar programas distribuídos por diferentes localidades. Tipicamente, múltiplos programas clientes compartilham os serviços de um programa servidor em comum (ClienteServidor, 2004).

Segundo Sundsted (2001), existem duas tendências que possibilitaram o aparecimento do P2P: a tendência natural à descentralização existente na engenharia de *software* e o surgimento de redes de computadores cada vez mais potentes, que dispõem de largura de banda cada vez maiores. Apesar da expressão *peer-to-peer* ser relativamente

nova, o P2P não é uma tecnologia recente. As primeiras aplicações em estilo P2P apareceram há cerca de 25 anos atrás (Minar *et al.*, 2001).

A USENET, criada em 1979, é uma aplicação distribuída que até hoje disponibiliza grupos de discussão (*newsgroups*) ao redor mundo. No princípio, os arquivos eram trocados *off-line* (em *batch*) através de linhas telefônicas e geralmente durante à noite, quando o tráfego era menor. Consequentemente, não havia uma maneira efetiva de centralizar os serviços da USENET em um único ponto. O resultado natural disso foi uma aplicação extremamente descentralizada e distribuída, estrutura que permanece até hoje (Sundsted, 2001). Em 1984, outra importante aplicação foi criada, a FidoNet. A FidoNet é uma aplicação descentralizada e distribuída que se destina à troca de mensagens entre diferentes usuários de sistemas BBS (*Bulletin Board System*). Assim como a USENET, a FidoNet cresceu rapidamente e ainda permanece em uso atualmente. Sundsted (2001) diz que esses dois projetos foram de grande importância porque enfrentaram com sucesso, há mais de duas décadas, problemas que modernos sistemas P2P enfrentam hoje em dia, tais como escalabilidade, segurança, problemas de endereçamento, entre outros.

O expressivo crescimento da Internet a partir de 1994 deslocou a rede mundial de computadores da exclusividade de seus nichos originais (militares e acadêmicos) e a tornou acessível ao cidadão comum (Minar *et al.*, 2001). Rapidamente a Internet se tornou um fenômeno cultural de massa e passou a existir, cada vez mais, pessoas interessadas em trocar correspondências por correio eletrônico, acessar páginas de hipertexto e comprar produtos em lojas virtuais. Foi necessário, então, se rever toda a arquitetura de rede existente e prepará-la para suportar um número cada vez maior de usuários. Quesitos como segurança e largura de banda e tecnologias como os NATs (*Network Address Translators*) e os *firewalls* tornaram-se cruciais neste planejamento. Todavia, apesar da enorme expansão, o projeto da rede permanecia praticamente o mesmo da sua concepção original.

Uma mudança significativa ocorreu no ano 2000. O aparecimento do programa de troca de músicas Napster (Napster, 2004) possibilitou a milhões de usuários em todo o mundo compartilhar arquivos diretamente entre os seus, cada vez mais potentes, computadores pessoais. A Internet começava então a deslocar o seu foco das aplicações cliente-servidor para aquelas onde fosse possível criar grupos de colaboração, sistemas de arquivos e supercomputadores virtuais (Minar *et al.*, 2001). O surgimento do Napster trouxe de novo à tona a idéia de descentralização dos sistemas.

Todos os sistemas P2P são sistemas distribuídos, porém a recíproca não é verdadeira. O P2P vem sendo cada vez mais utilizado como modelo para compartilhamento de dados, processamento e de transferência de informações em rede. No P2P os clientes também são servidores. Apesar das preocupações que desperta com relação às trocas de arquivos ilegais (aquelas que desconsideram os direitos autorais dos seus criadores), grandes empresas têm visto vantagens em utilizar sistemas P2P para compartilhar arquivos internos entre seus funcionários, evitando assim o desgaste envolvido na criação e manutenção de um servidor central (P2P, 2004). A má reputação do P2P está associada a programas como o Morpheus (Morpheus, 2004) e o Kazaa (Kazaa, 2004), sucessores do Napster na troca de arquivos (especialmente de músicas), que geralmente violam os direitos autorais. Garfinkel (2004) acha, no entanto, que o P2P pode ser usado de maneira positiva, por exemplo, possibilitando que qualquer pessoa possa publicar informações pela rede de um modo mais livre de censura e de controles de conteúdo. Segundo Peer-to-Peer Working Group (2002), a melhoria em três fatores colaboraram para o reavivamento da arquitetura P2P nos últimos anos: custo do processamento computacional, aumento da largura de banda e maior disponibilidade de armazenamento.

Pode-se destacar como vantagens do P2P (Parameswaran *et al.*, 2001): a transferência balanceada de conteúdo (uma vez que o quesito geográfico será levado em conta); os repositórios dinâmicos de informação (distribuídos por vários *peers* da rede); redundância e tolerância a falhas; endereçamento baseado em conteúdo; e pesquisas mais eficazes (envolvendo apenas os *peers on-line* e eliminando-se os *dead links*). Como principal problema do P2P destacam-se as diferenças de qualidade de conexão dos vários *peers*. Entretanto, algoritmos adequados de comparação de bandas, distâncias, etc., podem contornar esse problema, racionalizando e tornando mais rápida a comunicação.

Cada vez mais novas aplicações P2P vêm sendo desenvolvidas. Entre as mais recentes e potencialmente interessantes, pode-se destacar o Skype (Skype, 2004), o LionShare (LionShare, 2004a), o BitTorrent (BitTorrent, 2004), o Vipul’s Razor (Razor, 2004) e o Magic Mirror Backup (MMB, 2004).

O Skype (Figura 2.12), por exemplo, é um programa que disponibiliza telefonia gratuita via Internet. Segundo Garfinkel (2004), entre a sua estréia em agosto de 2003 e o mês de outubro de 2004, o Skype realizou mais de 1,7 bilhão de minutos de conversação P2P entre os seus usuários. Isto mostra toda a potencialidade que o P2P pode oferecer para

a área de tefonia via Internet. É importante ressaltar também que, neste caso, grande parte do sucesso deveu-se à gratuidade do serviço.

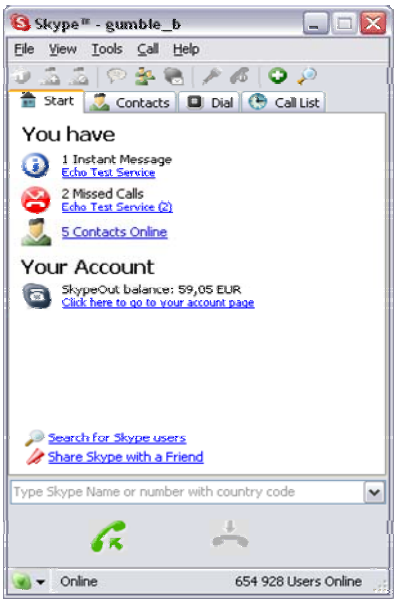

Figura 2.12 – Skype (Skype, 2004)

O LionShare é um projeto iniciado pela Penn State University com recursos da Mellon Foundation com o intuito de criar uma rede de informações acadêmicas compartilhadas (LionShare, 2004a). O sistema é projetado de maneira que cada um gerencia seus dados e seus arquivos pessoais e os torna acessíveis através de uma rede P2P. Sua meta é tornar públicos arquivos que alunos, professores, pesquisadores e bibliotecários possam ter, e que dados como fotos, sons, vídeos institucionais e apresentações de seminários possam ser localizados de maneira mais eficiente em comparação com outras ferramentas de busca (LionShare, 2004b).

O BitTorrent é um protocolo para a distribuição de arquivos. Sua busca por conteúdo é feita através de URL (*Uniform Resource Locator*) e é projetada para ser completamente integrada com a Web. Sua vantagem sobre o HTTP (*HyperText Transfer Protocol*) tradicional é que quando múltiplos *downloads* simultâneos de um mesmo arquivo ocorrem entre vários usuários, os arquivos incompletos que estes possuem já podem ir sendo compartilhados com os outros, de maneira que o tráfego na origem é dimuído significativamente (BitTorrent, 2004).

O Vipul's Razor é um sistema P2P utilizado para filtrar *spams* (*e-mails* indesejados). Um pequeno agente de *software* é executado em cada máquina da rede e detecta cada novo *e-mail* recebido. A teoria por trás disso é que sempre que uma mesma mensagem aparece em múltiplas localidades e mais ou menos ao mesmo tempo, é provavelmente um *spam* (Razor, 2004). Esta abordagem se torna potencialmente interessante se utilizada como complemento aos filtros *anti-spam* baseados em conteúdo.

Outra aplicação com bastante potencial é o Magic Mirror Backup (MMB, 2004), cuja primeira versão beta foi disponibilizada recentemente por um estudante de graduação do MIT (*Massachusetts Institute of Technology*). A idéia é utilizar o espaço em disco ocioso de computadores localizados em casa ou no trabalho para fazer cópias (*backups*) de arquivos entre si. Assim, toda a rede P2P passaria a ser um grande repositório redundante de dados.

De maneira geral, o P2P é uma tecnologia que pode ser combinada com várias outras na tentativa de se obter sistemas mais adequados para determinado tipo de aplicação. Segundo Hurley (2004), o elemento chave de um sistema P2P é o seu algoritmo, que define o comportamento de toda a rede pois impõe uma topologia lógica que cria um conceito de vizinhança entre os *peers*. Este conceito é de fundamental importância para o fluxo de dados na rede P2P, um requisito vital para sistemas que não possuem um recurso central. Também, o fato da rede poder ser flutuante, com o relacionamento entre os seus *peers* sendo redefinido a todo instante, é um dos principais fatores que tornam atrativos os sistemas P2P.

2.3.1 JXTA

Para se implementar de fato um ambiente descentralizado optou-se por uma plataforma voltada para o P2P: o JXTA. Esta recente tecnologia constitui-se de uma plataforma de programação em rede, projetada para atuar com computação distribuída, especialmente em uma abordagem P2P. O projeto JXTA é *open source*. Foi originalmente concebido pela Sun (JXTA, 2004a) e atualmente está sendo desenvolvido por várias pessoas ligadas à indústria e a entidades acadêmicas. A idéia principal consiste em se obter uma plataforma que seja interoperável (ou seja, que permita o agrupamento sistemas de diversas origens) e independente de sistemas operacionais, plataformas de rede e

linguagens de programação (Gong, 2002). O JXTA dispõe de um conjunto de protocolos de comunicação, sendo cada um definido por uma ou mais mensagens. Esta plataforma possui protocolos de localização, busca de conteúdo e transferência de dados entre os *peers*, além de requisitos de segurança tais como, confidencialidade, integridade, disponibilidade, autenticação, controle de acesso, auditoria, criptografia e segurança de dados.

A plataforma JXTA, e o P2P em geral, possibilitam que diferentes dispositivos, serviços e plataforma de rede interajam juntos. Baseada em padrões abertos, a tecnologia JXTA possibilita que estes dispositivos, em distintas plataformas de rede, se comuniquem sem nenhum tipo de problema interoperacional, e além do mais, oferece flexibilidade a operações que não estão restritas a sistemas proprietários (JXTA, 2004b). Através da diminuição da complexidade das operações em rede e da infra-estrutura disponibilizada para facilitar as operações, pode-se dizer que o JXTA oferece uma forma simples e fácil de se implementar sistemas P2P.

O JXTA cria um ambiente de utilização chamado Rede Virtual JXTA (*JXTA Virtual Network*) (Figura 2.13), que possibilita a qualquer *peer* se comunicar com os outros *peers* da rede, independente da sua localização, tipo e ambiente operacional (alguns *peers* podem estar localizados atrás de um *firewall*, por exemplo, ou em diferentes infra-estruturas de rede). Desta forma, o acesso aos recursos da rede não fica limitado por incompatibilidades de plataformas ou restrições hierárquicas da arquitetura cliente-servidor (JXTA, 2004b).

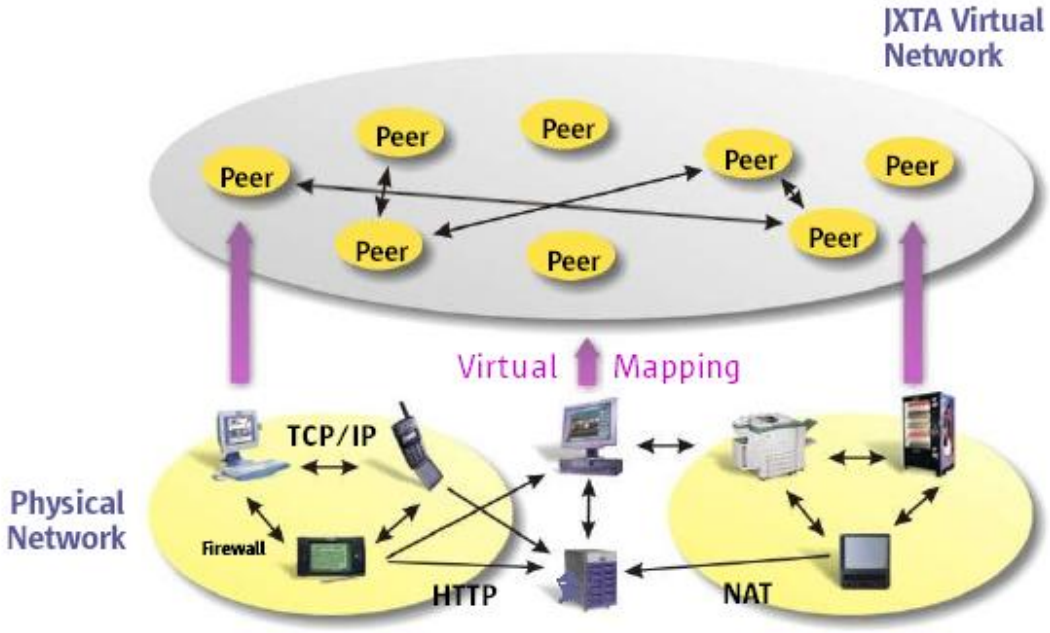

Figura 2.13 – Rede Virtual JXTA (JXTA, 2004b)

Para viabilizar a independência de plataforma de rede, sistema e linguagem de programação, o JXTA baseia-se em protocolos-padrão, como o HTTP, TCP/IP (*Transmission Control Protocol / Internet Protocol*) e XML. E também possui um sistema de roteamento e de redirecionamento de endereços dos *peers*. A Figura 2.14 mostra um exemplo de utilização destas ferramentas, que podem gerar vários cenários.

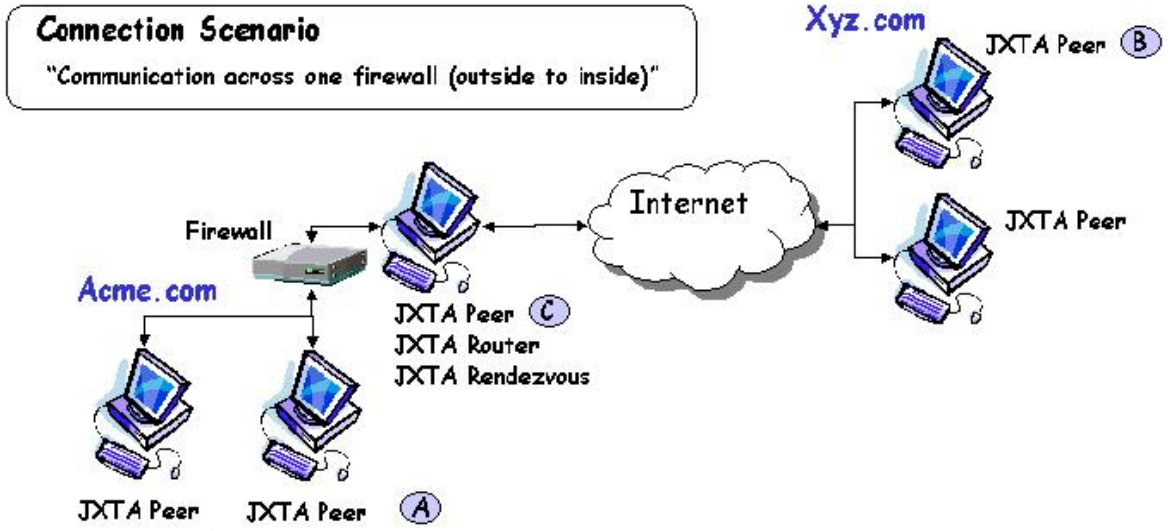

Figura 2.14 – Cenário JXTA – Comunicação Através de um *Firewall* (JXTA, 2004c)

Neste cenário de utilização, o *peer* B quer se comunicar com o *peer* A. Como B nunca se comunicou com A antes, não possui sua localização armazenada na memória ou em disco. A está atrás de um firewall, mas B não. B então percebe que A não é alcançável pela sua subrede local e então resolve mandar uma mensagem para todos os serviços de *rendezvous* (serviços que armazenam endereços dos *peers* com os quais se relacionam e que servem de referência para outros *peers*) que possui registrados perguntando sobre o *peer* A. C (*rendezvous*) responde a B, e envia o endereço de A. A abre uma coneção HTTP com C para ultrapassar o *firewall*. B então pode enviar a sua mensagem normalmente para C (via TCP), que esta será transportada até A pela conexão HTTP que foi aberta.

A Figura 2.15 mostra um cenário onde a descoberta de um peer é feita através da propagação de mensagens.

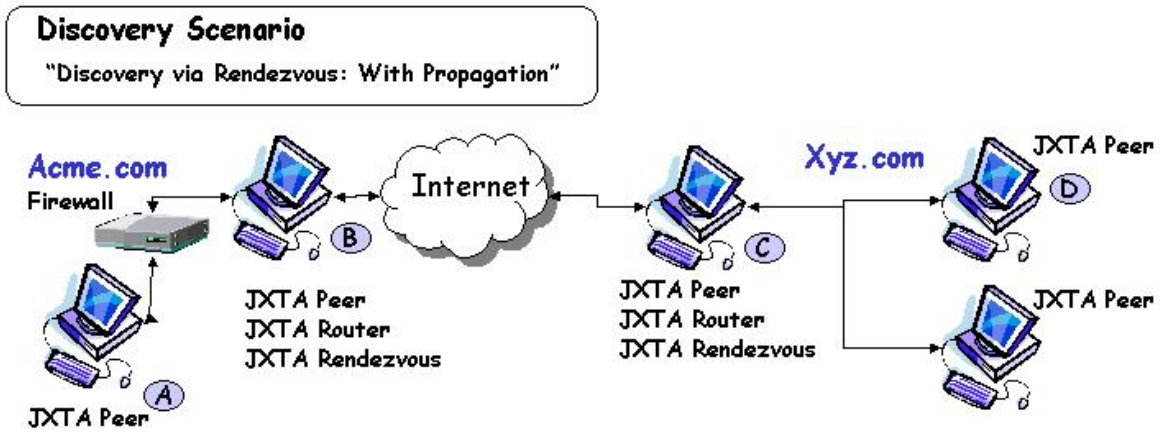

Figura 2.15 – Cenário JXTA – Descoberta com Propagação (JXTA, 2004c)

Neste cenário o *peer* A quer encontrar o *peer* D. B é o *rendezvous* registrado em A e não conhece D, C é um outro *rendezvous* e conhece D. A abre uma conexão HTTP com B e pergunta por D. Como B não conhece D, ele então propaga a mensagem para seus *peers rendezvous* registrados, entre eles, C. C então envia a informação sobre D (que sabe quem é) para B, que a repassa a A.

Em comparação com a plataforma Jini (Jini, 2004), por exemplo, que também é uma arquitetura distribuída de acesso a serviços baseada em Java (via RMI), o JXTA é mais abrangente. O JXTA utiliza sockets como unidade básica para a comunicação, é independente de plataforma, pode formar múltiplas comunidades dentro de uma mesma

rede virtual e também oferece acesso a serviços. O Jini utiliza um repositório de serviços (*lookup service*) onde estão armazenados os descritores de serviços e suas respectivas interfaces de acesso (*proxies*), contudo, este repositório é centralizado. Este mecanismo permite que fornecedores de serviços tornem seus serviços disponíveis, criando desta maneira, uma **Federação de Serviços** (Seção 2.2.1). Já o JXTA utiliza protocolos próprios de percorrimento em rede e de busca por conteúdo sobre a arquitetura *peer-to-peer*.

Pode-se citar também como principais diferenças entre as plataforma JXTA e Jini, segundo Vanmechelen (2003), as seguintes:

- O Jini é inteiramente baseado em Java e o JXTA é independente de plataforma, porque utiliza representação XML;
- O Jini considera os serviços como entidades essenciais, distribuídas e que executam funcionalidades remotas a um cliente, através de um *proxy* remoto. No JXTA os serviços não são essenciais e, além disso, não são diretamente endereçados pelo cliente;
- Ambas implementam o conceito de grupos. Porém, enquanto no Jini os grupos são usados exclusivamente no contexto da descoberta de serviços, no JXTA eles são mais elaborados, porque mecanismos de interação são disponibilizados de maneira a dotá-los de toda a funcionalidade disponível a um *peer*. Desta forma, os grupos JXTA são vistos como um único *peer* (*peer group*);
- Os dados Jini são acoplados a objetos Java e os dados JXTA são representados por documentos XML;
- O JXTA associa um identificador exclusivo para cada *peer* conectado à plataforma, o Jini não;
- O Jini não restringe buscas a nenhum tipo de *peer*, o JXTA oferece esta possibilidade;
- O JXTA não é limitado por *firewalls* na rede, ao contrário do Jini;
- O JXTA é mais flexível porque seus protocolos podem ser customizados, enquanto o Jini depende exclusivamente da comunicação RMI.

Entretanto, segundo Vanmechelen (2003), o Jini se encontra em um estado de maturação mais avançado em relação ao JXTA, possui melhor infra-estrutura (utilitários) para implementação e é desenvolvido diretamente pela Sun, enquanto o JXTA é *open-source*.

Apesar de ser uma tecnologia recente, diversos projetos que usam o JXTA têm surgido, como por exemplo (JXTA, 2004d):

- 312 LeanOnMe: *Backup* de arquivos;
- Zudha Instant Messenger: Comunicação instanânea;
- Tryllian Agent Development Kit: Aplicação para o desenvolvimento de processos de negócios distribuídos;
- Brevient Connect: Conferência Web;
- Etc.

Osório *et al.* (2003) propõe um modelo de negócios baseados em serviços JXTA que visa integrar diferentes empresas da área de transporte (postos de pedágio, postos de gasolina e estacionamentos) naquilo que foi chamado de Barramento de Integração (*Integration Bus – IBUS*). O objetivo deste projeto é que, através de comunicação via rádio entre os carros e as empresas, os débitos referentes aos serviços prestados serão efetuados automaticamente na conta corrente (ou cartão de crédito) dos usuários de maneira que estes não precisarão mais passar por um guichê ou caixa para pagá-los.

Além disso, grandes empresas como a Nokia e a Siemens (JXTA, 2004e) estão utilizando o JXTA em seus sistemas e cada vez mais universidades apresentam projetos que utilizam o JXTA, como o *radioJXTA* (RadioJXTA, 2004), que distribui áudio através de redes JXTA; e o projeto *shareddiary* (SharedDiary, 2004), que compartilha um diário entre um grupo de trabalho. Como estas, várias outras iniciativas têm despontado recentemente, tornando esta tecnologia muito promissora para o futuro.

Capítulo 3

3. A Federação de Provedores de Aplicação

Para viabilizar a composição de aplicações baseada na plugagem de componentes dinâmicos, observou-se a necessidade de se conceber uma arquitetura que se aproximasse o máximo possível de um modelo livre de arcabouços (*frameworks* e *containers*). Esta arquitetura deveria proporcionar uma estrutura que viabilizasse o transporte dos componentes aos seus destinos, quando da demanda por estes. Cada componente poderia originar-se das mais diferentes fontes, e todo o contato entre fonte e destino não deveria exigir maiores conhecimentos mútuos. Além disso, toda esta comunicação deveria ocorrer de forma transparente ao usuário da aplicação que requisitou o componente.

Por isso, com o intuito de se tentar atender a todos esses requisitos, propõe-se o conceito de uma *Federação de Provedores de Aplicação (FPA)* (Dutra e Rabelo, 2003) (Dutra e Rabelo, 2004). A FPA é uma entidade lógica que reúne vários repositórios de componentes. Ela é um aglomerado, um *cluster* de repositórios, onde cada um destes é visto como um *Provedor de Aplicação (PA)* (Seção 3.2.1).

Enquanto no modelo PSA tradicional (Seção 2.2.2) as aplicações ou serviços são executados (via rede) neles mesmos, no servidor, no modelo FPA / PAs as aplicações requeridas são trazidas (via rede) ao ambiente-cliente e executadas localmente. Portanto, o modelo FPA / PA oferece aplicações propriamente ditas e, complementando o modelo, serviços de suporte são oferecidos às aplicações-cliente para que procurem, localizem e tragam o componente necessário ao ambiente local de execução que o requisitou, de forma transparente. Essas aplicações serão modeladas na forma de componentes de *software*,

plugados sob demanda às aplicações-clientes quando estas, em tempo de execução, requisitarem as funcionalidades que necessitam.

O que se busca neste trabalho é o desenvolvimento de uma arquitetura que viabilize a criação de Aplicações Compostas (*Composite Applications*, Capítulo 2), tanto para a utilização com novos sistemas quanto para a utilização em sistemas legados. A arquitetura proposta nesta dissertação irá basear-se na utilização de componentes, pequenas aplicações que podem ser desenvolvidas à parte ou integradas a aplicações maiores. Devido ao fato de serem entidades independentes e genéricas (pois não fazem parte de nenhum sistema), são utilizados como peças de integração. Esta integração serve para especializar as aplicações e para torná-las mais aptas para o desenvolvimento de determinada tarefa. Esta arquitetura será compatível com as infra-estruturas de comunicação existentes hoje em dia. Desta forma, as empresas e usuários em geral não terão dificuldades na adaptação para o novo modelo. Agindo desta forma, uma aplicação composta protege os investimentos já existentes mesmo quando as práticas de negócios mudam (Krass, 2004).

Do ponto de vista dos *clientes*, a FPA é uma organização com a qual irão negociar a utilização de componentes. Pode-se fazer aqui uma analogia com uma locadora de filmes. Diferentemente daquelas, onde o cliente entra e pega os produtos que lhe interessam antes de passar no balcão do caixa, na FPA não existem “prateleiras” locais expondo seus “produtos”. Cada cliente deverá se reportar diretamente ao “balcão”, descrevendo todas as características do “produto” que lhe interessa. No caso da locadora, essas características envolveriam gênero do filme, protagonistas, ano de lançamento, duração, entre outras. Para a FPA, as características que importam são funcionalidade, versão, compatibilidade, portabilidade, etc. Assim sendo, depois de informado pelo cliente de suas necessidades, o “balcão” da FPA se encarregará de ir buscar o “filme” (componente) em outras “lojas da rede”. Estas “lojas” estarão geograficamente distribuídas em qualquer parte do mundo e o cliente não terá o menor conhecimento disso. Para ele, a visão física da FPA se resume ao “balcão”, onde ele detalha seus interesses. Os “filmes” pedidos na FPA serão “entregues” (plugados) posteriormente em sua “casa” (ambiente computacional), e virão com prazo de validade definido.

Desta forma, cada “loja” da rede equivalerá a um PA. E o equivalente ao “balcão” será o *PAC - Provedor de Aplicação Coordenador*. Haverá vários PAs distribuídos, transparentes aos clientes, flexibilizando os custos e tempos de rede para a plugagem uma

vez que os componentes virão dos PAs mais adequados para cada caso (por exemplo, considerando distância, banda, etc.). Os componentes armazenados em seus repositórios serão adaptáveis (pois serão desenvolvidos com esta caracaterística) a vários tipos de *hardware* e ambientes de execução, podendo ser executados tanto em PCs com sistemas operacionais tradicionais, como também em dispositivos móveis de computação (celulares e *palms*).

A granularidade (Capítulo 1) pretendida para estes componentes é a mais variável possível, porém num primeiro momento, será mais focada nos de baixa granularidade, objetivando maximizar o grau de adequação e adaptação destes a uma necessidade funcional (inclusive ligada ao *hardware*) específica dos clientes. Assim sendo, a plugagem ocorrerá dinamicamente e em tempo de execução – *plugagem por demanda*. Fazendo uma analogia aos sistemas de produção, pode-se dizer que existirá um *just-in-time* no *software* sendo usado por um cliente; à medida que o usuário vá acessando funcionalidades ainda não existentes na aplicação, os respectivos componentes que as implementam serão “puxados” para o “ambiente de produção” local.

De maneira geral, a FPA irá definir um contrato de utilização de componentes (visando a composição dinâmica), assim como no exemplo da locadora, onde isto também é necessário. O contrato com a FPA, como será visto mais adiante, detalhará coisas como periodicidade, atualização automática de versões, prazo de validade dos componentes, entre outros.

3.1 Proposta Conceitual

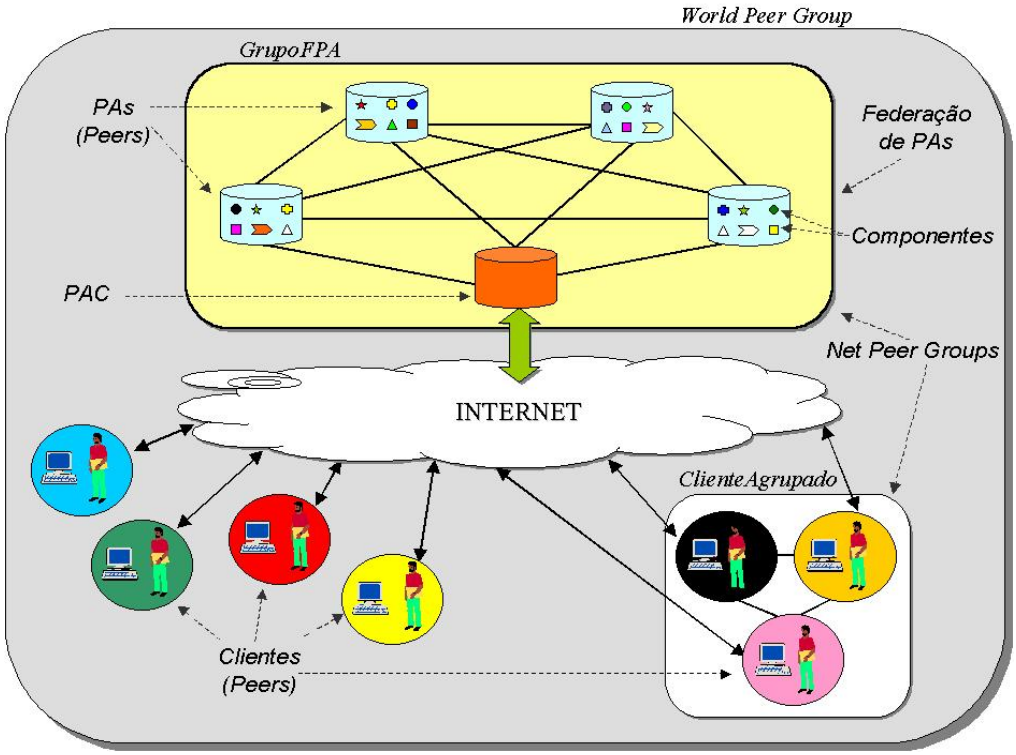

Figura 3.1 – A Federação de Provedores de Aplicação e seus Clientes

A Figura 3.1 mostra a visão geral da abordagem proposta. A idéia consiste da construção de aplicações compostas por dezenas de funções – desenhadas como componentes – disponibilizadas de forma logicamente plugável na federação de PAs.

O ponto de partida para a elaboração da FPA deu-se com os Provedores de Serviços de Aplicação (PSA / ASP) (Dewire, 2002). O modelo PSA mostrou-se desde o início ser uma abordagem muito útil pois a idéia proposta por ele de disponibilizar programas via Internet a clientes assinantes representou um passo significativo na direção de se flexibilizar a utilização de *software*. Em um cenário mais extremo, um usuário PSA não precisaria nem dispor de armazenamento local que não o estritamente necessário para hospedar o sistema operacional de seu PC e um programa cliente (*thin-client*) PSA, por exemplo, um navegador. Neste caso, este usuário estaria com todos os seus dados, além dos seus aplicativos, situados no provedor de serviços. Estes aplicativos e dados poderiam ser acessados via rede, sempre que necessário.

Além da vantagem da economia de espaço de armazenagem, objetiva-se obter uma grande economia de tempo e de dinheiro, pois novas versões e atualizações corretivas dos programas são instaladas não mais no computador local, mas diretamente no PSA, o que as torna disponíveis imediatamente ao cliente no seu próximo acesso. Os usuários dos PSAs também possuem uma grande vantagem: o suporte do provedor definido em contrato. Isto se mostra potencialmente útil para usuários leigos. Ao invés de se depararem com "caixas-pretas" na compra de um novo *software* – acompanhadas de manuais nem sempre inteligíveis e serviços de atendimento ao consumidor pouco prestativos, seja por telefone ou pela Internet – os usuários menos especializados ficam liberados para simplesmente negociar a utilização (aluguel) de determinados aplicativos e / ou alugar espaço em disco no provedor, os chamados discos virtuais. Todo o suporte necessário é providenciado pela empresa e a "instalação" local se limita à configuração de acesso remoto ao PSA.

Já no modelo FPA, tem-se inicialmente apenas o *kernel*, o núcleo da aplicação, composto pelas funcionalidades essenciais da aplicação do *software*. Uma vez que este esteja em execução, os componentes necessários conectar-se-ão diretamente a este *kernel* à medida que forem sendo invocados, considerando-se os termos contratuais do modelo de negócios estabelecido entre a FPA e o *cliente*. Quando o cliente tentar usar uma funcionalidade que não estiver presente no *kernel*, uma requisição será enviada ao coordenador da federação, o Provedor de Aplicação Coordenador (PAC), que será o responsável pela localização inteligente dos componentes dentro da federação. Esta requisição levará consigo a especificação do componente ausente – previamente definida pelo desenvolvedor da aplicação – e irá então efetuar uma busca, via Internet, com o intuito de encontrar o PA mais apropriado para suprir a necessidade. Uma vez encontrado este repositório, será estabelecida uma comunicação *peer-to-peer* entre ele e a aplicação *kernel*, de maneira que o componente seja transportado e plugado a ela, de acordo com a especificação da requisição. Toda esta operação ocorrerá transparentemente ao usuário da aplicação *kernel*.

Um esforço adicional de projeto e de desenvolvimento será necessário durante a fase de definição do *kernel* do cliente, analogamente à AOS (Seção 2.2.3), que necessita ainda de que certos requisitos de *middleware* sejam atendidos (Hritz, 2004). As aplicações clientes devem ser projetadas para conter apenas a funcionalidade básica. Toda a funcionalidade passível de ser expandida dinamicamente deverá ser composta por

referências (especificações disponibilizadas pelo desenvolvedor) a componentes dinâmicos.

Tal como o modelo PSA, a FPA apresenta algumas vantagens para usuários-empresa, entre as quais pode-se citar:

- A dimuição do custo com licenças de *software*;
- A redução da necessidade de profissionais de TI (Tecnologia de Informação);
- O controle do inventário de *software* existente em cada máquina.

As licenças podem ser negociadas de maneira diferente, por exemplo, por setor ou departamento. Os PSAs necessitam negociar licenças especiais com os desenvolvedores pois uma licença não significará mais apenas um, e sim muitos clientes utilizando o mesmo produto de forma compartilhada. Assim, as empresas que optam por utilizar serviços PSA podem negociar preços mais vantajosos por uma licença que será usada por vários funcionários. A outra vantagem se refere aos departamentos de TI das empresas usuárias, que podem ser reduzidos drasticamente. Este tipo de profissional fica menos sobrecarregado (e consequentemente, pode melhor se especializar) e assim se concentrar em outras áreas da infra-estrutura empresarial. O PSA se encarregaria desta engenharia em relação aos aplicativos disponibilizados a seus clientes, utilizando a sua própria equipe de TI. O modelo PSA também deve disponibilizar redundância e recuperação de dados, controle de retomadas de transações ou tarefas interrompidas, segurança e integridade dos dados (Network, 2000). Esta abordagem libera os clientes para efetuar processamentos paralelos em suas máquinas.

Apesar de todas essas vantagens, o modelo PSA possui algumas limitações em termos de flexibilização de *software*. Isto ocorre porque a granularidade modular das aplicações existentes nos PSAs permanece muito alta, quer dizer, o cliente normalmente necessita acessar toda a aplicação, mesmo que pretenda usar apenas um pequeno módulo ou funcionalidade dela. Isto significa que esta aplicação terá que ser carregada para a memória do servidor e que irá gastar processamento deste, pois irá carregar módulos desnecessários a quem a está acessando. Além disso, estará se pagando pelo uso de toda a aplicação e não pelo que realmente se usará.

Uma outra limitação tem a ver com a sua topologia. O modelo PSA é tipicamente centralizado. De certa forma, pode-se dizer que os PSAs acabam por representar um modelo de *mainframes* mais flexível, mais dinâmico e mais "comercial". Mais flexível porque a figura dos terminais locais sem capacidade de processamento foi substituída por um computador local (usualmente um PC), que possui muito mais autonomia e independência. Dinâmico porque o provedor pode ser acessado mais facilmente de lugares diferentes, por exemplo, de onde houver um navegador Internet instalado. E mais comercial porque o seu modelo de negócios é voltado para absorver grande demanda de serviços, enquanto que os *mainframes* geralmente limitavam-se a redes internas e / ou a poucas aplicações. Entretanto, os PSAs herdam a centralização dos *mainframes*, e esta herança causa problemas, como o processamento excessivo nos provedores, que precisam ser máquinas computacionalmente poderosas, além da usual pouca oferta de opções ao cliente PSA. Pode-se exemplificar esse último aspecto supondo-se um cenário em que um provedor não ofereça todos os serviços que determinado cliente necessita. A saída viável para este, então, está em comprar a aplicação no mercado, pela maneira tradicional, ou assinar os serviços de um segundo PSA, que a disponibilize. Além de tudo isso, o processamento excessivo em um PSA também acaba por potencializar problemas típicos de sistemas centralizados, como desempenho, segurança e controle dos dados. Para diminuir a vulnerabilidade de seus clientes, cada provedor terá que arcar com os custos de manter a sua infra-estrutura desobstruída (evitando o excesso de processamento) e eficaz.

Neste sentido, o modelo da FPA proposto nesta dissertação procura estender o modelo PSA de maneira a contornar essas desvantagens. Em primeiro lugar, os PAs de uma FPA trabalham sempre em conjunto, em uma rede *peer-to-peer*, o que representa um sistema distribuído e descentralizado por excelência. Esta característica é de extrema importância pois permite a escalabilidade do sistema. Neste caso, as chances do cliente ficar insatisfeito diminuem pois mesmo que determinado provedor sofra qualquer tipo de desconexão durante uma transação, sempre poderá haver outro que possua o componente desejado e que possa tomar o seu lugar. E quanto maior e mais distribuída for a FPA, menor será a chance de ocorrência desse tipo de problema. O problema do gargalo de processamento também praticamente desaparece no modelo FPA. Isto porque os PAs não executam mais as tarefas remotamente para os clientes, mas as transportam para que posssam ser executadas por eles localmente. O processamento nos PAs se restringe ao das transações de buscas por componentes, como mais à frente está detalhado na Seção 3.2.1.

O processamento do PA coordenador é um pouco mais complexo do que os dos outros PAs, mas ainda assim, muito inferior do aquele que tem lugar em um servidor central.

Pode-se citar também como vantagem da FPA em relação aos PSAs, o fato de que o modelo de negócios FPA é mais dinâmico e mais completo. Isto porque o Cliente FPA fecha um contrato com toda a entidade lógica e não mais com um exclusivo provedor. Desta forma, todo provedor que se associar à FPA passará automaticamente a ser um potencial fornecedor de todos os clientes previamente usuários da federação. É mais completo porque estes detalhes estarão estipulados no contrato firmado com o PAC, sem a necessidade de se conhecer cada repositório individualmente.

Entretanto, a principal vantagem do modelo proposto FPA em relação aos PSAs está na flexibilidade funcional. É dado ao cliente o direito de utilizar apenas o que for necessário para a execução do processo. Se se tomar como exemplo o sistema I-Dox (Seção 2.2.2) fica clara a diferença. O PSA tradicional disponibiliza a aplicação de edição de textos de maneira monolítica. Assim, quem quiser editar um arquivo presente no disco virtual, deverá fazê-lo acessando toda essa aplicação. Supondo-se que em determinado momento um usuário precise fazer apenas uma pequena alteração em uma linha de texto, ele vai ter que aguardar o tempo de carregamento (e *overhead* de processamento) necessário para que todo *software* seja carregado, aí incluindo-se por exemplo, o módulo gráfico. A pergunta que se faz é a seguinte: será que é adequado um sistema que exige do usuário que quer alterar apenas duas ou três palavras no seu texto arcar com os custos gerais de utilizar um programa que oferece opções indesejadas e infinitamente mais pesadas, como um pacote gráfico, e ainda ter que pagar por todo esse programa, e não apenas pelo módulo ou funcionalidade de interesse efetivamente necessários?

A FPA herda do modelo PSA a liberdade espacial de acesso, ou seja, de poder demandar componentes, mesmo estando em diferentes máquinas e em diferentes lugares, desde que estes possuam as estruturas internas que caracterizam um Cliente FPA. Todavia, os modelos se diferenciam bastante em outros itens, como já citado anteriormente. Desta forma, procurando-se resumir o objetivo da arquitetura proposta neste trabalho em única frase, pode-se dizer que a idéia é “usar apenas o necessário, no momento necessário e no local necessário”.

Analogamente à AOS (SOA, 2004a) (SOA, 2004b) (Seção 2.2.3), a Federação de Provedores de Aplicação (FPA) também pode ser considerada uma arquitetura voltada para a fraca acoplagem. Sua tarefa é implementar determinada(s) funcionalidade(s) que pode(m) ser das mais diversas granularidades. Apesar deste objetivo em comum, as duas abordagens diferem quanto ao estilo de acoplagem (plugagem) utilizado. Enquanto a FPA transporta os componentes para serem executados localmente nos seus clientes, a AOS baseia suas ações na utilização de serviços de *software*, acessados remotamente pelos seus clientes ("consumidores"), geralmente como serviços Web.

Uma Arquitetura Transparente ao Usuário

A abordagem de formar um aglomerado de repositórios de componentes surge da necessidade de se ter um ambiente cooperativo, ou seja, aquele em que cada repositório possa contribuir com o que disponha. Como será visto na Seção 3.5, cada provedor é livre para negociar o contrato que quiser com os desenvolvedores de *software*, e cada um deles pode usar isso da maneira que lhe convier. Assim, a federação como um todo não deixará de atender o cliente, mesmo que determinado desenvolvedor resolva negociar com o provedor *X* no lugar do provedor *Y*.

A proposta de aglomerar virtualmente entidades distribuídas tem a ver não apenas com flexibilidade, com execução adaptável, mas também com controle de operação. Esta abordagem também é utilizada por Camarinha-Matos *et al.* (2001) na sua proposta da *Federação de Serviços* (FS), usada no projeto FETISH (FETISH, 2003) no domínio do setor de turismo. Sua meta é dotar ao máximo o sistema de *Serviços de Valor Agregado* (SVAs). Isto é, na FS, o cliente negocia com muitos ao mesmo tempo, porém reportando-se a apenas um. A FS visa a resolução de um problema na forma de um pacote completo de soluções. No caso da indústria do turismo, este pacote englobaria o transporte, a alimentação, a hospedagem, o itinerário e os serviços em geral relacionados à viagem ou dependentes dela. Já para a FPA não faz a menor diferença se um componente será utilizado em uma tarefa exclusiva ou para agregar valor a um serviço; o que importa para ela é detectar que alguém o necessita e enviá-lo tão logo quanto possível ao ambiente computacional que o invocou.

Pode-se notar aqui a primeira semelhança entre estes dois modelos: ambos possuem uma certa máquina de inferência interna. Ambos necessitam, em determinado momento,

tomar decisões fundamentais para as operações que suportam. A FS, através de detalhes como preferências pessoais e histórico do cliente, decidirá quais opções se encaixam melhor no perfil de quem consulta o sistema. A máquina de inferência da FS toma como base os dados preenchidos pelos usuários além do banco de dados do seu servidor central. De maneira parecida, na FPA também existe um momento em que o provedor coordenador (PAC) precisa tomar uma decisão. A decisão do PAC ocorre sempre que sua busca retorna mais de um anúncio de componente. Neste momento, ele deverá escolher qual dos provedores (PAs) é o mais apropriado para servir o Cliente FPA. As especificações recebidas, bem como a largura de banda e latência de rede, serão analisadas com o intuito de se fazer essa escolha.

O PAC, enquanto *broker* da FPA, encontra um similar no servidor central do modelo FS. Este último é o responsável por todo o processamento que vise à agregação de valor dos serviços. Ele é dotado de um catálogo de serviços (Garita *et al.*, 2002), responsável por armazenar todos os serviços que estiverem disponíveis na FS. Cada provedor de serviços, ao registrar seu(s) serviço(s) neste catálogo, torna-se automaticamente apto a ser acessado pelos clientes. Os clientes da FS acessam o ambiente através de um navegador de Internet. O servidor central possui um servidor Web que será acessado remotamente pelos clientes. Cada cliente preenche vários itens de um formulário, com o objetivo de ter o seu perfil traçado pelo sistema. As suas escolhas serão fundamentais para o servidor central, que então irá comparar as preferências do cliente com a especificação dos serviços armazenados no catálogo. E através da interação com o cliente, definir claramente qual será o pacote de serviços agregados. Neste ponto existe um hiato com a FPA, porque o PAC não armazena nenhum tipo de informação, mas entretanto, também compara as especificações enviadas pelos clientes com aquelas anunciadas pelos PAs. Como será visto em detalhes no próximo capítulo, cada PA acaba por se registrar dentro da FPA e anunciar seus componentes.

O catálogo de serviços da FS também armazena as interfaces dos serviços. Quando necessitado, um manipulador que empacota a interface do serviço é copiado para o cliente que então faz a invocação diretamente ao provedor de serviços. De maneira análoga à FPA, em que o Cliente FPA não tem conhecimento do provedor que lhe transferiu o componente, na FS também não existe a necessidade desta informação. O cliente da FS simplesmente utiliza o serviço que lhe foi oferecido, indiferentemente de onde se localize

provedor. Uma vez transferidos os manipuladores de todas as interfaces a serem utilizadas, o cliente passa a transacionar diretamente com os provedores de serviço (Figura 3.2). Na FPA, o PAC finaliza o seu papel depois da plugagem dinâmica, e toda a utilização da funcionalidade do componente passa, então, a ser responsabilidade única e exclusiva do cliente.

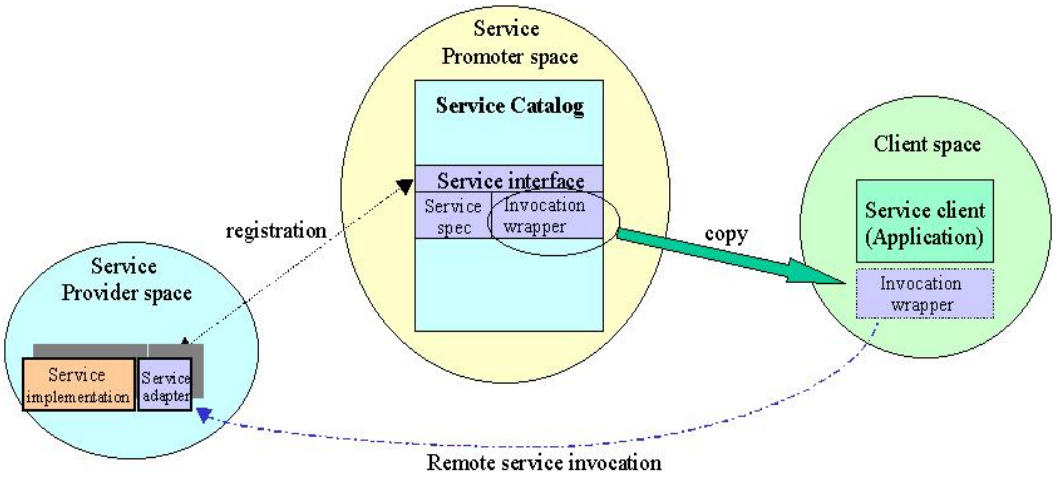

Figura 3.2 – O Catálogo de Serviços da FS (Garita *et al.*, 2002)

A FS baseia-se na utilização de serviços Web. Já a FPA utiliza componentes dinamicamente plugados. Contudo, isto não significa que os modelos não possam ser combinados. Considerando o fato de que o Cliente FPA pode ser qualquer tipo de aplicação, obter-se-ía um cenário onde poderiam ser clientes da FPA tanto o cliente FS, quanto os provedores FS, e até mesmo o servidor central. E essas aplicações poderiam utilizar a FPA para complementar alguma funcionalidade bem específica, que não fosse utilizada com frequência ou que fosse utilizada de maneira personalizada por diferentes tipos de usuário. Desta forma, atingir-se-ía um ambiente de utilização de *software* onde uma aplicação poderia ser ao mesmo tempo cliente da FS e da FPA.

Finalmente, identifica-se a diferença mais marcante para quem utiliza os dois modelos: a transparência da transação ao usuário. A FS é completamente “dependente” da interação com o usuário no seu processo de tomada de decisão. Na verdade, suas decisões se caracterizam em opções, que serão posteriormente avaliadas pelo usuário. São informações subjetivas, que variam de usuário para usuário. Já na FPA, todo o processo da composição dinâmica é feito de maneira transparente; no máximo – dependendo do

modelo de negócios adotado – um pequeno aviso ou confirmação pode ser emitido antes de se começar a busca na federação. Todas as outras questões relacionadas a especificações são atribuições exclusivas do PAC, que tem o poder de decisão. Neste caso, esta é a melhor solução para o usuário, que necessitará apenas de alguns instantes até possuir a funcionalidade desejada, completamente integrada à sua aplicação. Quando a estiver usando, ele não terá conhecido detalhes da especificação que foi procurada e muito menos a sua procedência. Também não conhecerá detalhes técnicos como nome ou tamanho de arquivo. A única coisa que saberá, e que é a que mais lhe interessa, é que o componente foi plugado e está em plena utilização, assim como era necessário.

3.1.1 Componentes Dinâmicos

Um *componente dinâmico* é um componente de *software* que agrega características inerentes ao modelo FPA. Basicamente se pode dizer que, se um componente está apto a ser localizado, transportado e plugado pela FPA, então ele é um componente dinâmico. Esta diferenciação conceitual é proposta aqui com o intuito de possibilitar uma melhor compreensão do tipo básico de estrutura a ser utilizada neste trabalho.

Um componente dinâmico documenta (através de *metadados*) características não-funcionais e subjetivas, que serão utilizadas tanto por desenvolvedores quanto por usuários que porventura no futuro utilizem este componente. A especificação de um componente dinâmico envolve alguns detalhes específicos, por exemplo, alguns itens não-funcionais precisam estar definidos formalmente, para facilitar a tomada de decisão no momento da plugagem. Entre estes itens pode-se destacar o propósito da construção do componente, seu potencial de expansão, sua interoperabilidade com outros componentes e até mesmo um histórico de sua utilização (sua ou do seu tipo) para determinada tarefa ou integração com determinado sistema. Baseado nesses dados, pode-se estabelecer um critério de qualidade, que poderá ser útil para futuros usuários e / ou desenvolvedores.

Este tipo de componente surge da necessidade de se suportar a **plugagem dinâmica**. Nesta abordagem, a composição final do sistema não é definida em tempo de projeto, mas em tempo de execução. Os componentes são plugados **por demanda**. A aplicação-núcleo se “expande” (e depois pode se “encolher”) em tempo de execução de acordo com o uso das suas funcionalidades, ou seja, busca-se atingir a escalabilidade total do sistema. O

processo de plugagem por demanda implica em se ter uma infra-estrutura mais robusta, adequada para suportar uma plugagem rápida e o mais transparente possível.

A idéia de se plugar componentes dinamicamente já havia sido proposta anteriormente. Fraga *et al.* (2003) propõem um modelo que troca dinamicamente componentes de uma aplicação. Através do travamento do cliente e da impossibilidade deste de fazer novas requisições foi possível se evitar a inconsistência da operação. Lauder (1999) propôs um modelo onde existiria uma fábrica plugável (*pluggable factory*) (Figura 3.3) que utilizasse classes abstratas associadas. Esta fábrica seria a responsável por receber uma requisição externa de instanciação de determinada classe abstrata e retornar um clone dela. Cada uma dessas classes abstratas possuiria uma ou mais classes classes concretas agregadas. Uma, e apenas uma, dessas classes concretas estaria registrada na classe abstrata. Assim sendo, quando a fábrica requisitasse uma instância de uma determinada classe abstrata, esta retornaria uma instância da classe concreta atualmente registrada. Este registro poderia ser alterado a qualquer momento e por qualquer uma das classes concretas existentes, especializando desta forma, a classe abstrata.

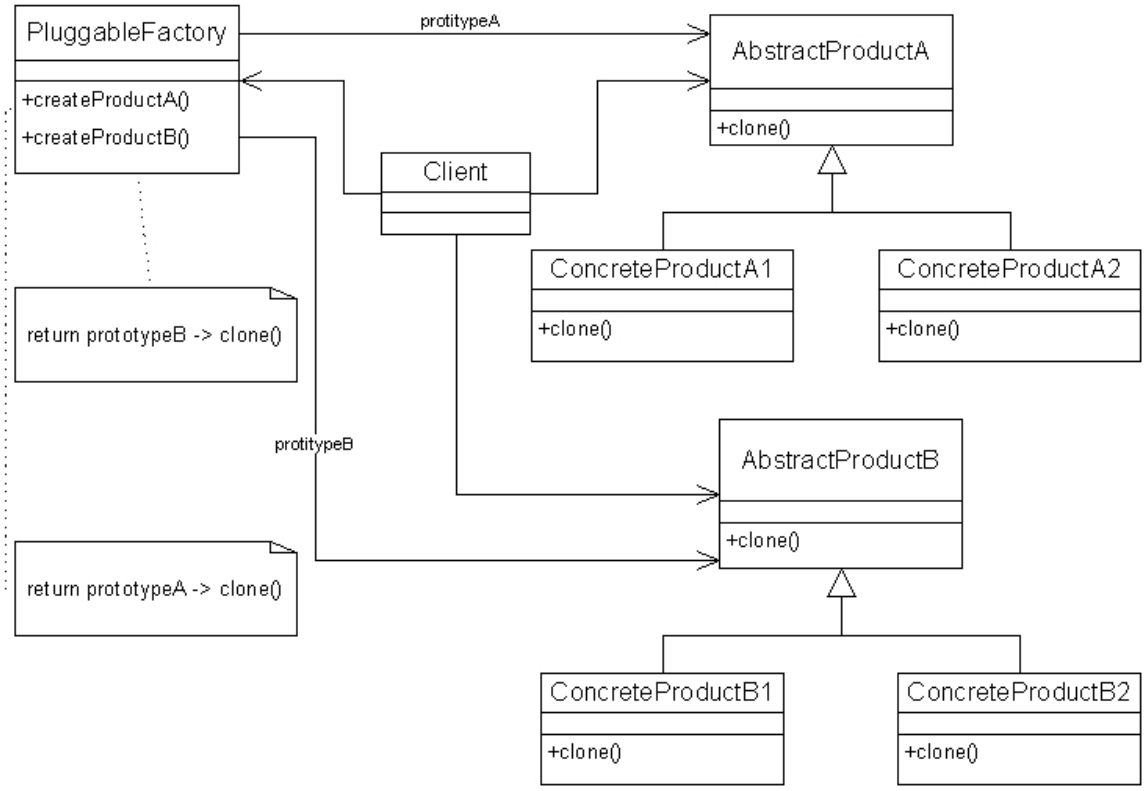

Figura 3.3 – *Pluggable Factory* em UML (Lauder, 1999)

Entretanto, ficou claro que este modelo era uma limitação para classes abstratas que possuíssem muitas classes concretas agregadas, que fatalmente precisariam ser utilizadas simultaneamente. Além do mais, seria necessário se conhecer todas as classes abstratas do sistema no momento de se requisitar uma instância. A solução achada foi criar uma classe intermediária, que mapeasse as classes concretas por nomes, que seriam repassados pela fábrica. Com isto, eliminou-se a necessidade de se conhecer todas as classes abstratas do sistema. Mas ainda restava um grande problema: o tempo de registro de novas classes concretas em sistemas que exigissem transações em tempo real – como um banco, por exemplo – continuava inaceitável. Era necessário que se pudesse plugar novas classes concretas de produtos à fábrica em tempo de execução, isto é, agregar classes que não existiam no momento em que a aplicação foi escrita.

Para tal, eliminou-se o registro das classes concretas do modelo, que passaram a ser associadas com suas equivalentes abstratas em um arquivo de configuração dinâmica. A fábrica então utilizaria este arquivo para descobrir a localização da classe requisitada, para depois passá-la a um plugador dinâmico (*dynamic linker*) do sistema operacional. O sistema utilizado, neste caso o UNIX e suas variações (incluindo Solarix e Linux), possui chamadas de sistema como a *dllopen()*, que abre um determinado arquivo no caminho especificado, e a *dllsym()*, que localiza o arquivo e depois que o acha, pluga-o na aplicação.

Seiter *et al.* (1999) propôs um modelo para a plugagem dinâmica baseado em *framework*. Esta proposta possui semelhanças com a de Lauder (1999), pois este *framework* seria constituído também por classes abstratas, que por sua vez serviriam como base para a instanciação de métodos concretos. Aqui também aparece o problema de ser ter que definir de antemão todas as classes abstratas do modelo, o que se não o inviabiliza, pelo menos diminui consideravelmente a sua eficácia. Seiter *et al.* (1999) toma como base a arquitetura de execução Java e foca seus esforços em uma programação orientada a componentes, onde módulos binários pré-existentes seriam compostos (*glued together*) com a aplicação em tempo de execução. A idéia consistiu em dotar este *framework* de métodos genéricos e imutáveis, ou seja, não haveria aqui a idéia de agregar classes abstratas. Este *framework* faria então o papel de fábrica. Cada nova classe poderia ser agregada simplesmente implementando internamente os métodos definidos pela fábrica e mapeando-os para seus próprios.

Um Modelo Dinâmico

Os modelos tradionais de componentes vêm sendo cada vez mais utilizados. Isto ocorre porque há determinados tipos de aplicações que se encaixam perfeitamente em suas arquiteturas, e para as quais as desvantagens existentes não afetam tanto o desempenho, ou simplesmente, são irrelevantes. É neste contexto que é pensada a FPA. Não como um modelo substitutivo aos já existentes, mas da mesma forma que eles, uma alternativa à Programação Orientada a Objetos (POO), porém com uma concepção diferente do que seja um componente. Por isso a nomenclatura de *Componente Dinâmico*: porque para o modelo FPA um componente não apenas se limita a ser composto em tempo de execução; ele está envolvido em uma transação (incluindo detecção de sua falta pelo cliente, busca, transporte, plugagem e retomada do fluxo de execução) que irá acontecer, na sua totalidade, de maneira dinâmica. Este é o grande diferencial em relação às abordagens tradicionais. Além disso, ao final do processo de plugagem, o cliente terá disponível o módulo localmente, não tendo a necessidade de acessá-lo de maneira remota.

Em linhas gerais, pode-se visualizar um componente dinâmico como um módulo a ser transportado até o cliente, no momento em que existir demanda por ele. Virá somente a funcionalidade desejada, que será plugada, utilizada, e depois descartada (de acordo com o modelo de negócios vigente). A FPA tentará garantir ao máximo (através das especificações) que o componente que venha seja a última versão disponibilizada pelo desenvolvedor. Ele poderá ter sido implementado em qualquer linguagem de programação, pois se o cliente possuir o conector apropriado, a composição será realizada. Sua execução local transformará a sua relação com a aplicação-cliente de um esquema cliente-servidor em um esquema orientado a objetos, inclusive podendo conter características originais deste, como herança de classes e polimorfismo. A FPA prima pela composição, não pela transação. Seu objetivo-maior é plugar o componente ao cliente, e não dotá-lo de estruturas de suporte que permitam disponibilizar suas funcionalidades como um serviço. A FPA é uma estrutura reativa à falta de componentes (*component fault* – tentativa da aplicação de tentar acessar uma funcionalidade ainda não plugada) (Capítulo 4). Sua concepção está intimamente ligada à descentralização e independência de utilização destes componentes. Se eles disponibilizam serviços ou não; se são funcionalidades essenciais ou meros detalhes personalizados; se se comunicam via mensagens, eventos ou chamadas diretas; se

serão plugados a outros componentes dinâmicos ou não; tudo isso será transparente à FPA no momento em que efetuar uma operação de composição dinâmica.

Portanto, pode-se considerar toda a federação como um grande repositório, um grande conjunto de componentes, aptos para serem buscados no momento em que se necessitar tê-los plugados a qualquer aplicação. Todo esse processo estará automatizado neste conjunto, bastando, a quem interessar usufruir de seus serviços, tornar-se seu cliente.

A especificação de componentes dinâmicos aproveita boa parte das idéias apresentadas na Seção 2.1.7, através do modelo UML de Cheesman *et al.* (2001), para especificar as suas funcionalidades, e assim como a AOS (Wilkes, 2004a) necessita também especificar *metadados* na sua interface para identificar a função e o propósito do componente. Portanto, a especificação de um componente dinâmico é composta por duas partes:

- Especificação das assinaturas dos métodos da interface, semelhante ao que fazem os modelos tradicionais (como o CCM) através de suas IDLs (OMG, 2004c);
- Especificação das características não-funcionais (*metadados*), como por exemplo:
  - ✓ Versão;
  - ✓ Plataforma de execução (sistema operacional);
  - ✓ Tipo do processador;
  - ✓ Compilador;
  - ✓ Linguagem de programação;
  - ✓ Autor;
  - ✓ Tipo de componente;
  - ✓ Local de desenvolvimento;
  - ✓ Etc.

Na Seção 4.2.3 será mostrado um exemplo deste tipo de especificação.

Outro ponto a ser ressaltado diz respeito à plugagem de um componente em outro componente. Para que isto ocorra, o componente precisa se comportar como um cliente da FPA. Neste caso, é preciso que possua, além de suas funcionalidades, algum mecanismo de comunicação para notificar o *kernel* da aplicação, de maneira a ativar o seu *Módulo Gerenciador de Componentes* (Seção 3.2.3). A partir do momento em que um componente dinâmico é plugado no *kernel*, logicamente, ele passa a fazer parte deste *kernel*, e portanto, se torna passível de ser dinamicamente composto com outros componentes, desde que tenha sido projetado para suportar a composição dinâmica.

Devido à diversidade de maneiras em que a plugagem dinâmica pode ocorrer (Seção 2.1.8), a desplugagem de um componente poderia acontecer também de diferentes formas. Entretanto, na tentativa de se uniformizar este processo, definiu-se que, para que a desacoplagem aconteça, é necessário se remover todas as referências existentes para o componente, tanto as que estão na memória quando as que estão em disco. A estrutura interna da FPA foi concebida para gerar um *component fault* sempre que não encontrar o componente presente fisicamente no disco. Desta forma, a cada novo acesso a um componente plugado, essas referências são verificadas no intuito de se respeitar o modelo de negócios definido para o cliente em questão.

A idéia dos componentes dinâmicos se baseou fortemente na idéia dos plug-ins (Seção 2.1.6). Os *plug-ins* são módulos de *software* que são adicionados a um programa para estender a sua funcionalidade. Apesar das muitas semelhanças existentes, alguns pontos valem a pena serem ressaltados:

- O cliente da FPA não conhece a fonte dos seus componentes;
- O processo de composição dinâmica é transparente enquanto que dos *plug-ins* poucas vezes o é. Em alguns casos, inclusive, o usuário final tem que fazer quase tudo manualmente;
- A fonte de um *plug-in*, que vem especificada na aplicação cliente geralmente é sempre a mesma;
- A concepção dos componentes dinâmicos os colocam com granularidade variada, enquanto que os plug-ins geralmente são projetados para fazerem tarefas específicas;

- Não existe um consenso sobre a definição de um *plug-in*.

3.2 Entidades

São três os atores principais da arquitetura FPA: o Provedor de Aplicação (PA), o Provedor de Aplicação Coordenador (PAC) e o Cliente FPA. Os dois primeiros estão agrupados e cooperam dentro do *GrupoFPA*, que é o agrupamento lógico que caracteriza a Federação de Provedores de Aplicação. Enquanto os PAs existem em número indefinido, o PAC é único dentro deste agrupamento. O que caracterizará um Cliente FPA é o seu módulo de comunicação com a federação.

3.2.1 O Provedor de Aplicação

Um Provedor de Aplicação (PA) é um repositório de componentes. Diferentemente dos Provedores de Serviços de Aplicação (Seção 2.2.2), estes aqui armazenarão componentes que serão plugados dinamicamente e em tempo de execução, o que significa que os PAs não disponibilizarão serviços, mas sim aplicações. Além disso, o modelo proposto é totalmente distribuído, com os componentes provindos dos mais diferentes locais (os diversos possíveis PAs), selecionados através de critérios como localização geográfica e largura de banda. Diferentes versões de um mesmo componente poderão estar disponíveis em um ou mais PAs, entretanto, todas deverão possuir a mesma interface pré-definida.

A tarefa de um PA compreende:

- Anunciar à Federação de Provedores de Aplicação (FPA) a existência de cada componente presente em seu repositório;
- Receber uma requisição enviada pelo PAC, dentro da FPA, em busca de determinado componente;
- Verificar se a especificação pesquisada corresponde à do componente existente em seu repositório.
- Notificar o PAC no caso de ser um potencial provedor. Esta notificação é uma confirmação de disponibilidade que o PA envia ao PAC. É uma garantia que este último tem de que o anúncio encontrado realmente confere.

- Aguardar pela ordem de envio do componente ao cliente ou por nova requisição.

Um PA possui seis módulos cooperativos, como pode ser visualizado na Figura 3.4.

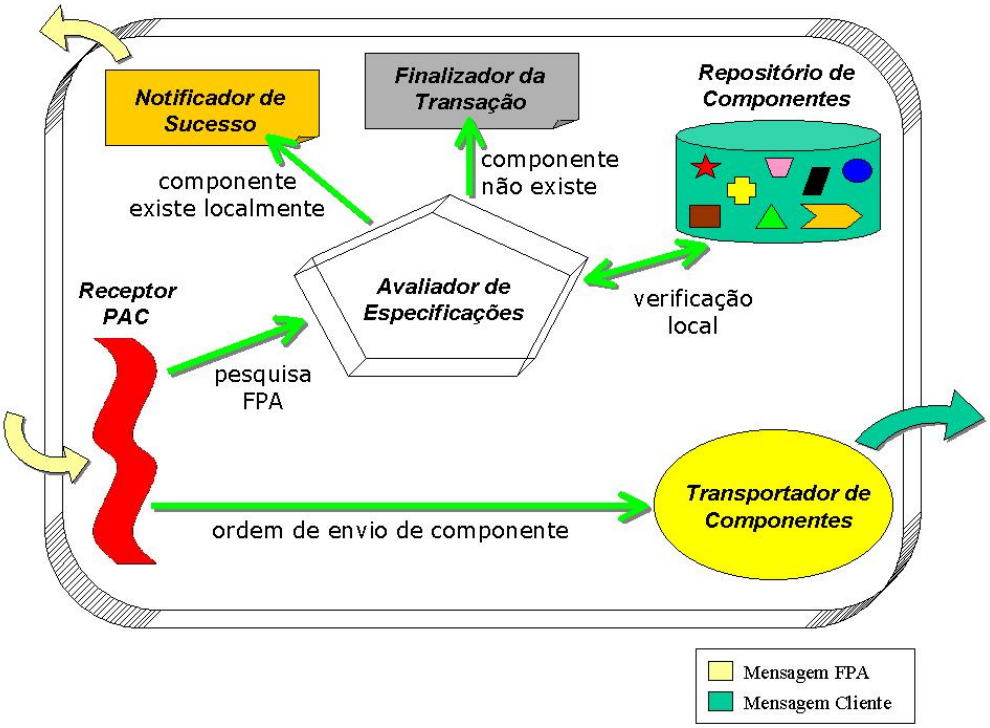

Figura 3.4 – Estrutura do Provedor de Aplicação

O primeiro módulo é o *Receptor PAC*, que irá receber as requisições redirecionadas pelo PAC. Cada requisição será verificada pelo *Avaliador de Especificações*, que a analisará e a aprovará ou não, segundo uma busca dentro do *Repositório de Componentes*. Se o componente pesquisado existir, a PA então avisará ao PAC através do *Notificador de Sucesso* de que seu componente é um potencial candidato a ser transferido para o cliente. Caso o PAC se decida por ele (Seção 3.2.2), uma ordem de envio de componente será recebida pelo *Receptor PAC* para ser repassada ao *Transportador de Componentes*, que irá efetuar a transferência. O *Finalizador da Transação* será acionado sempre que a especificação requisitada não corresponder a nenhum dos componentes presentes no repositório. É importante deixar claro que esta transação passível de ser finalizada é a que está ocorrendo dentro do PA. Isso significa dizer que a transação global continua em execução em outros membros da FPA (incluindo o PAC).

Cada PA possui um sistema interno de enfileiramento de mensagens que o possibilita receber mais de uma requisição ao mesmo tempo. Como será visto no próximo capítulo,

toda requisição identifica o cliente de origem, e esta informação chega ao PA junto com a especificação do componente desejado. A lógica interna de funcionamento de um PA se limita à comunicação com o PAC, que coordena as suas ações. Um PA age de maneira reativa às ordens recebidas do PAC, seja para pesquisar determinada especificação, seja para transportar um componente para o cliente. Outro item a ser ressaltado é que, da mesma forma que um Cliente FPA, um PA não conhece outros PAs. Apesar de coexistirem em uma plataforma *Peer-to-Peer* (Capítulo 4), em nenhum momento dos seu ciclos de vida dentro da federação eles necessitam comunicar-se diretamente com seus pares. E este desconhecimento mútuo pode ser uma grande vantagem, como ficará mais claro na análise do modelo de negócios da federação na Seção 3.5. A Figura 3.5 mostra o diagrama de atividades do PA.

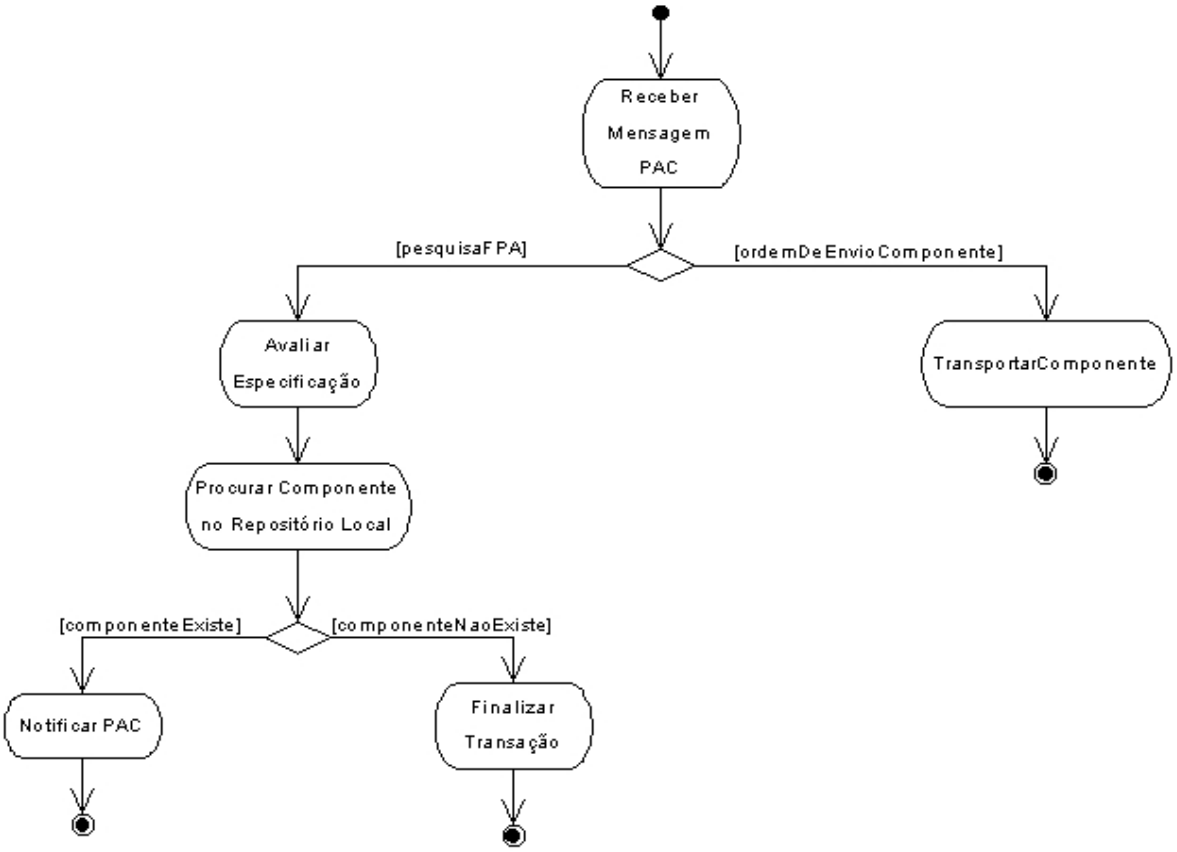

Figura 3.5 – Diagrama de Atividades do Provedor de Aplicação

3.2.2 O Provedor de Aplicação Coordenador

O Provedor de Aplicação Coordenador (PAC) (Figura 3.6) é o grande cérebro da Federação de Provedores de Aplicação (FPA). Externamente, ele é a visão da FPA para os clientes. No plano interno, é o gerenciador das ações dos PAs durante uma operação de composição dinâmica. O PAC tem o poder de decidir qual provedor enviará o componente requisitado pelo Cliente FPA. É ele quem recebe as requisições dos clientes, as valida e as envia para serem pesquisadas dentro da federação. Depois de recebidas as respostas, ele então decide qual PA possui o componente mais adequado para ser transferido.

Uma requisição de um cliente é composta pelos seguintes itens:

- Especificação do componente (que é disponibilizada pela empresa desenvolvedora);
- Nome do cliente;
- Nome do componente;
- Anúncio da porta de comunicação por onde este cliente espera receber o módulo a ser plugado.

Como será detalhado no próximo capítulo, todo o esquema de busca dentro da FPA, que é uma arquitetura P2P, é feita na base de anúncios. Tanto os *peers*, quanto as portas de comunicação, nomes de componentes, entre outros, são anunciados dentro da rede. Estes anúncios são a base da pesquisa por conteúdo utilizada pela da federação. Assim, quando o Cliente FPA envia uma requisição ao PAC, este extrai a especificação recebida anexa e a armazena para comparar com as respostas (que também são especificações) resultantes da pesquisa pelo nome do componente. Esta comparação servirá para a tomada de decisão, pois especificações inválidas ou incompletas são automaticamente descartadas. Em casos onde existam uma ou mais especificações compatíveis, é optada pela que oferece versão mais nova. E em casos em que duas ou mais especificações sejam idêntidas, entrarão na validação itens como largura de banda com o PA, localização geográfica, tempo de resposta (*ping*), preferência por determinado desenvolvedor, histórico de utilização, etc.

A Figura 3.6 mostra a estrutura do PAC, que é composta por sete módulos.

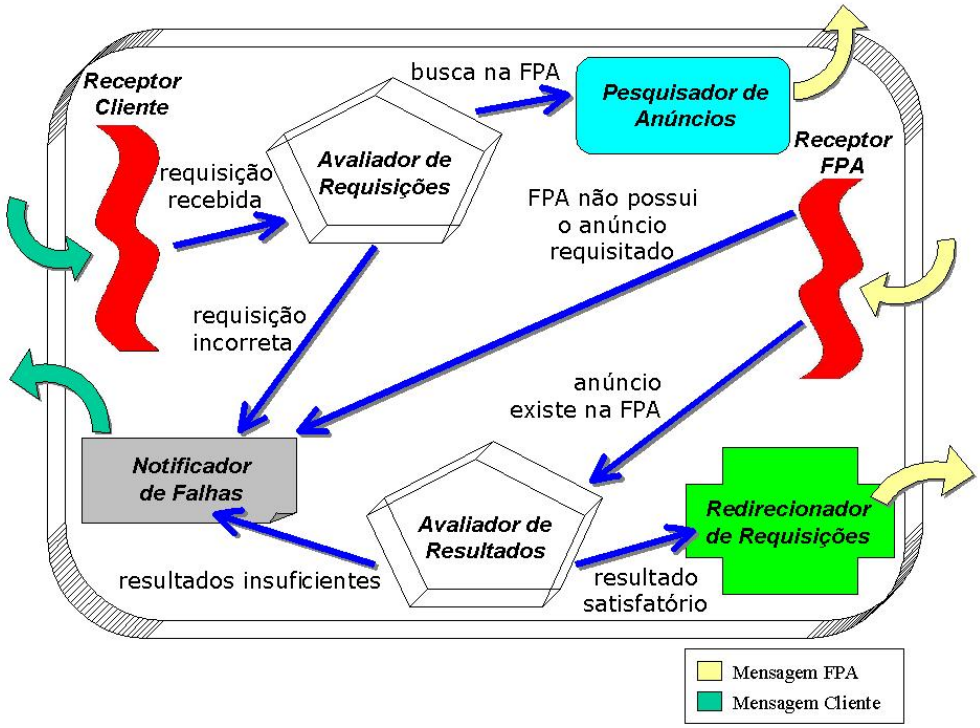

Figura 3.6 – Estrutura do Provedor de Aplicação Coordenador

O *Receptor Cliente* irá receber as requisições dos clientes. Estas requisições serão validadas pelo *Avaliador de Requisições*, que irá analisá-las em função de suas especificações e dos termos do contratos estabelecidos com cada cliente. O *Pesquisador de Anúncios* irá procurar pelo anúncio do componente dentro da FPA. O *Receptor FPA* aguardará pelos resultados desta procura, e enviará cada um recebido para o *Avaliador de Resultados*. Este irá fazer uma verificação dos resultados – segundo os critérios de avaliação da FPA – e decidir se existe algum deles que se adequa perfeitamente às características exigidas pelo cliente. Se algum resultado for satisfatório, enviará uma ordem ao *Redirecionador de Requisições* a fim de repassar a requisição para o PA escolhido como repositório. O *Notificador de Falhas* irá avisar o cliente no caso do componente não existir dentro da FPA, no caso da requisição não ter sido aprovada, no caso de todos os resultados serem insatisfatórios, ou ainda, no caso de algum outro problema de comunicação haver ocorrido.

O PAC também será o responsável por estabelecer dois tipos de contrato: um para caracterizar a relação com os Clientes FPA e outro para definir a relação com os Provedores de Aplicação. Desta forma, o PAC será um corretor de negócios, um *broker* do modelo. Na Figura 3.7 é mostrado o diagrama de atividades do PAC.

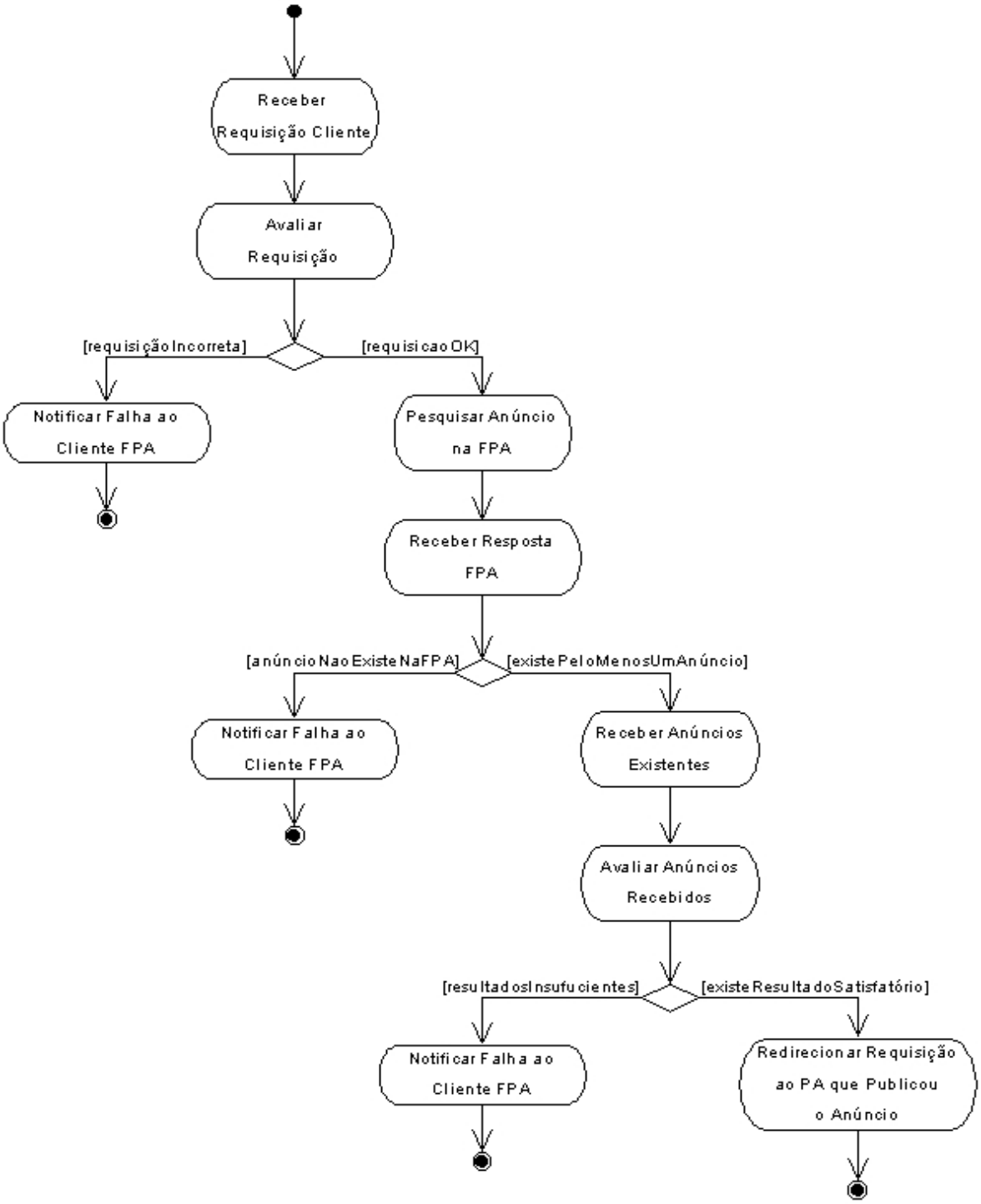

Figura 3.7 – Diagrama de Atividades do Provedor de Aplicação Coordenador

3.2.3 O Cliente da Federação

Um Cliente FPA é uma aplicação que requisita componentes à FPA. Uma máquina torna-se apta a hospedar aplicações-cliente no momento em que possuir um módulo de comunicação (um *binding*) que possibilite a cada aplicação caracterizar-se como um *peer* do modelo, como um Cliente FPA.

Cada cliente possuirá um *Repositório Local de Componentes (RLC)*, que servirá como uma espécie de depósito (*cache*) para os componentes já transferidos. O RLC pode ser visto como um "PA local" do cliente. Uma vez que estes forem descartados (Seção 3.1.1), e se porventura vierem a ser necessitados novamente, este repositório deverá ser consultado antes de ser feito novo pedido à federação. Esta operação visa oferecer mais rapidez e menor custo ao processo de plugagem dinâmica. As consultas a este repositório deverão ser feitas também segundo especificações bem detalhadas, para evitar por exemplo, que se utilize componentes com versões desatualizadas. Será definido um prazo de validade para os arquivos que estiverem neste repositório, e todas as operações que o envolver deverão estar detalhadas no modelo de negócios vigente.

O Cliente FPA possui um módulo denominado *Módulo Gerenciador de Componentes*. É ele quem dá início ao processo de requisição de componentes, a partir do momento em que for notificado pela aplicação de origem sobre essa necessidade. Ele é composto por cinco sub-módulos (Figura 3.8).

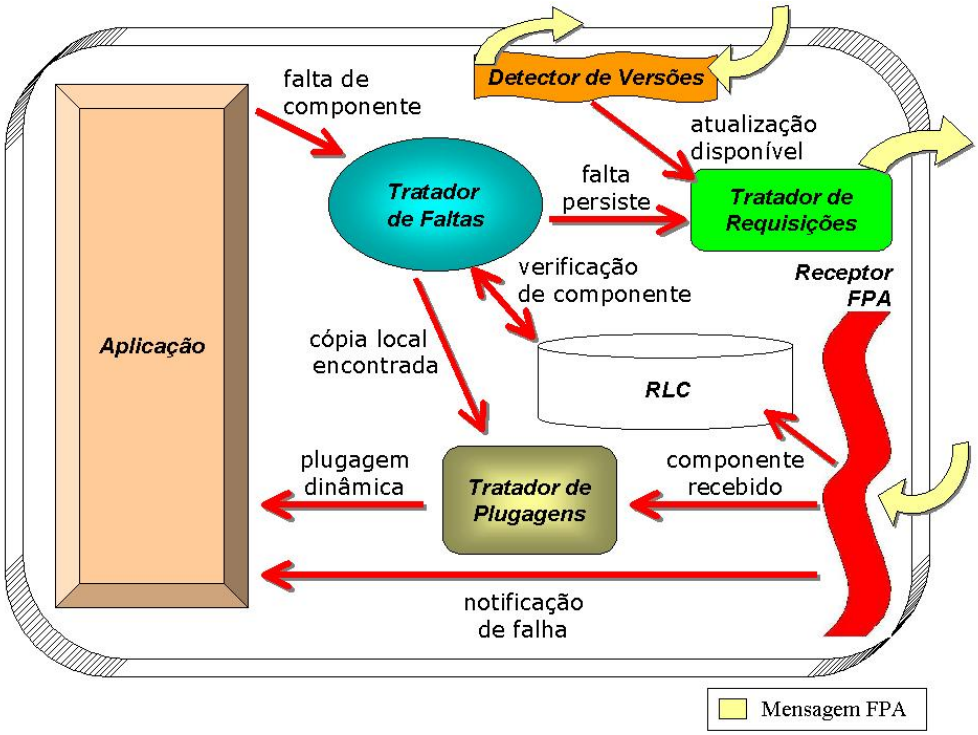

Figura 3.8 – Estrutura do Cliente FPA

- *Tratador de Faltas*: Age sempre que a aplicação detectar que necessita de um componente que não está presente localmente. O tratador então irá tentar localizá-lo no *RLC*. Se ele estiver lá, uma notificação será enviada ao *Tratador de Plugagens*; caso contrário o *Tratador de Requisições* será acionado;

- *Tratador de Requisições*: Monta a requisição do componente baseada no seu nome, na sua especificação e no nome do cliente. Depois, a envia para o PAC da FPA.

- *Receptor FPA*: Aguarda pela resposta da requisição enviada à FPA. Em caso de resposta positiva (ou seja, encontrou-se o componente esperado), o *Receptor FPA* recebe a mensagem, retira o componente anexo, envia uma cópia para o *RLC* e outra para o *Tratador de Plugagens*. No caso de falha, uma notificação será enviada à aplicação. O *Receptor FPA* também recebe as novas versões de componentes requisitadas pelo *Detector de Versões*. Nesta caso, os componentes recebidos são enviados exclusivamente ao *RLC* e não há notificação de falhas.

- Tratador de Plugagens: Executa a plugagem dinâmica propriamente dita. Carrega o componente para a memória da aplicação e desbloqueia o cliente de maneira que o mesmo passe a utilizá-lo.

- *Detector de Versões*: Este módulo verifica permanentemente na FPA se não existem novas versões dos componentes ora presentes no RLC. Esta periodicidade irá depender do modelo de negócios adotado pelo cliente, podendo ser de dias, semanas ou até meses. Resolveu-se criar este módulo de verificações em cada cliente (e não na FPA) para que se mantivesse a coerência com a descentralização da arquitetura P2P. Desta forma, cada requisição por nova versão possui um campo na mensagem que indica tratar-se de uma atualização. Este campo serve para que o *Receptor FPA* a identifique no momento em que chegar a resposta da FPA e, assim, não acionar o *Tratador de Plugagens*.

O diagrama de atividades do Cliente FPA pode ser visto na Figura 3.9.

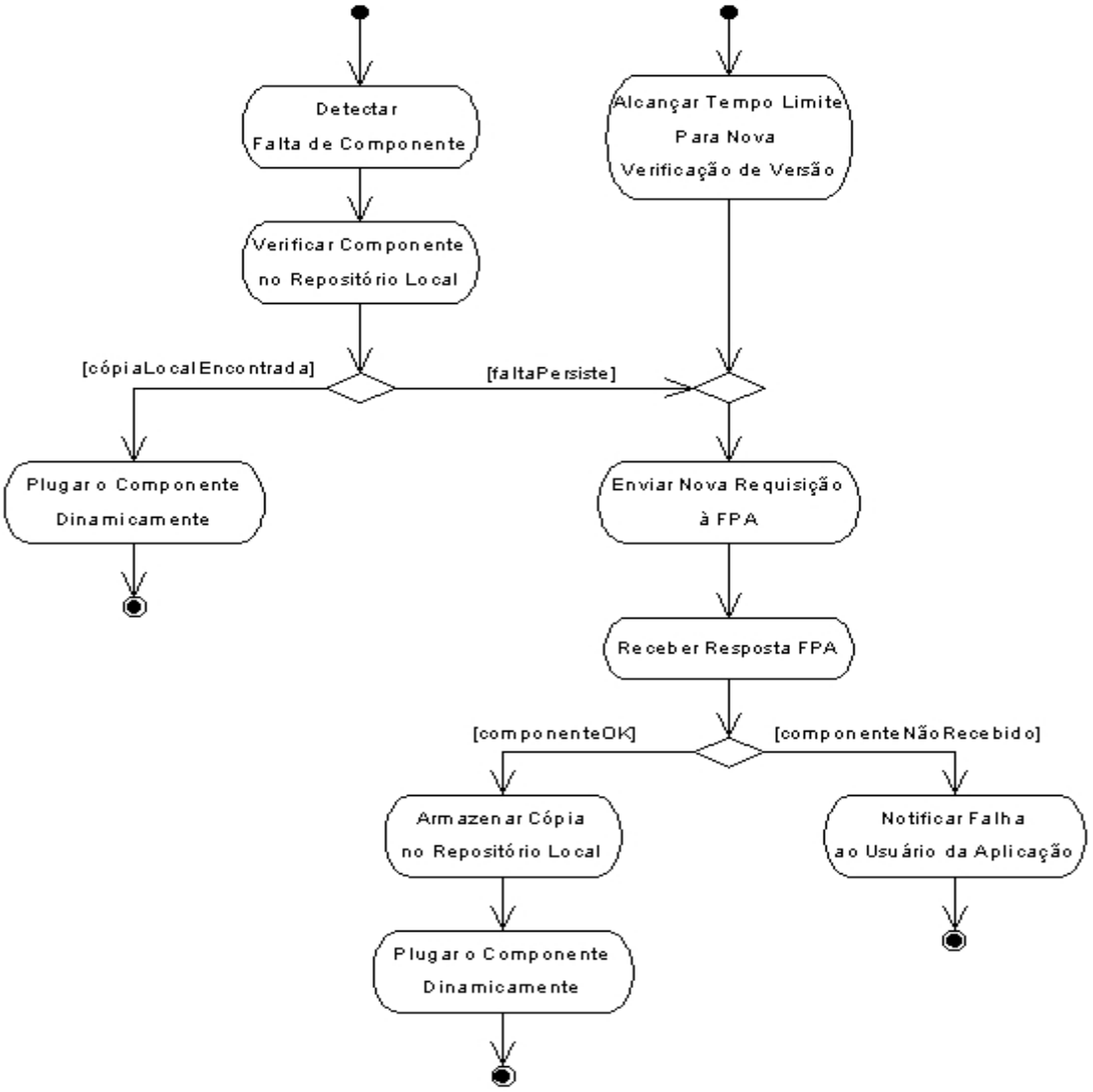

Figura 3.9 – Diagrama de Atividades do Cliente FPA

Clientes Agrupados

As máquinas que abrigam Clientes FPA podem estar sozinhas ou agrupadas com outras máquinas. Neste caso, denomina-se o conjunto destas (e suas aplicações) como ClienteAgrupado. Estas aplicações podem ser desenvolvidas em qualquer tipo de linguagem de programação e voltadas para qualquer plataforma. Dois ou mais clientes (aplicações) podem coexistir na mesma máquina. Por exemplo, considerando-se um cenário onde existam duas máquinas: uma executando o sistema operacional Linux e outra com o sistema Windows, e que possuam dois Clientes FPA, sendo um construído em Java e o outro em C++. Se se considerar que estes quatro componentes executam a mesma funcionalidade, ter-se-ía então quatro versões distintas desta funcionalidade em execução.

Isto significaria que o mesmo componente estaria implementado em Java e em C++ e que poderia ser executado tanto sobre a plataforma Windows quanto sobre a plataforma Linux.

A vantagem de se ter um ClienteAgrupado é que o *RLC* pode ser compartilhado entre eles. Se se imaginar um conjunto de aplicações de naturezas semelhantes, que se especializam com o mesmo tipo de componentes, isto pode ser de grande utilidade. Além de dar mais rapidez ao processo de plugagem (pois o componente poderia estar presente no *RLC* com uma frequência maior, já que outros clientes também estariam utilizando-o), esse agrupamento iria diminuir o tráfego de dados dentro da FPA, já que bastaria que um dos clientes executasse o processo de plugagem completo. A Figura 3.10 mostra a estrutura de um ClienteAgrupado.

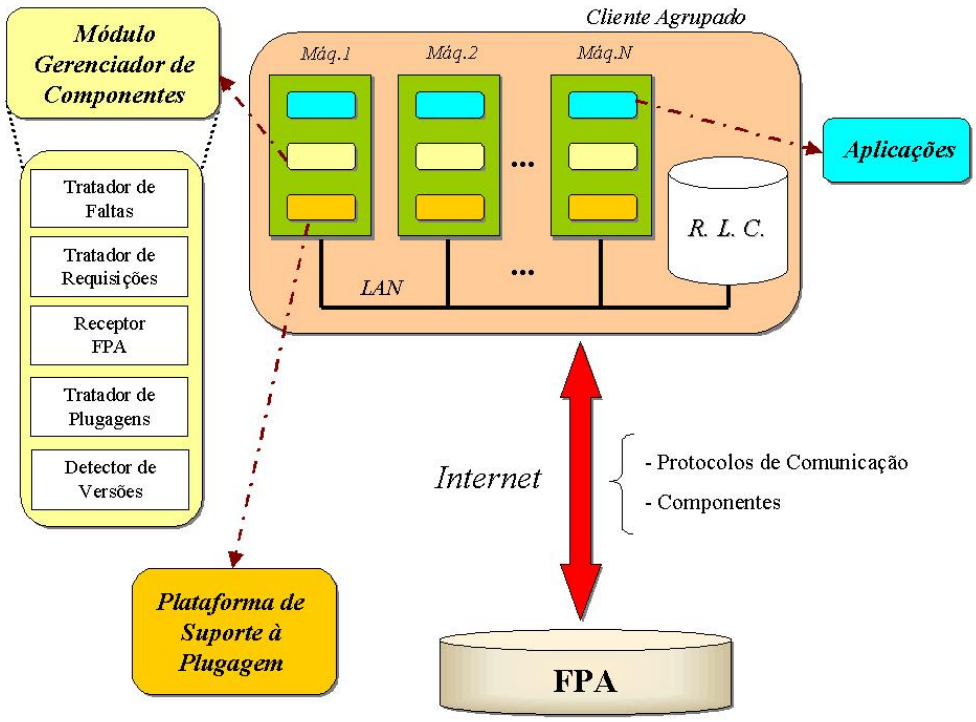

Figura 3.10 – O Cliente Agrupado

Pode-se notar que cada máquina possui, além de suas aplicações, um *Módulo Gerenciador de Componentes* (detalhado na Figura 3.8), e a *Plataforma de Suporte à Plugagem (PSP)*. A PSP engloba todos os requisitos necessários para que uma aplicação possa se comunicar através da arquitetura *Peer-to-Peer* (P2P), ou seja, para que se torne um *peer*. Portanto, diz-se que uma aplicação é um *peer* quando esta possui todas as ferramentas que a possibilitem interagir com outras aplicações (*peers*) em um ambiente

P2P. Na abordagem proposta nesta dissertação, utilizou-se a plataforma JXTA (Seções 2.3.1 e 4.2.2) para se implementar a PSP.

Para efeitos de contrato, o ClienteAgrupado deverá estar discriminado no modelo utilizado.

3.3 Caso de Uso

As Figuras 3.11, 3.12 e 3.13 mostram a relação dos atores da FPA através de diagramas de casos de uso do modelo.

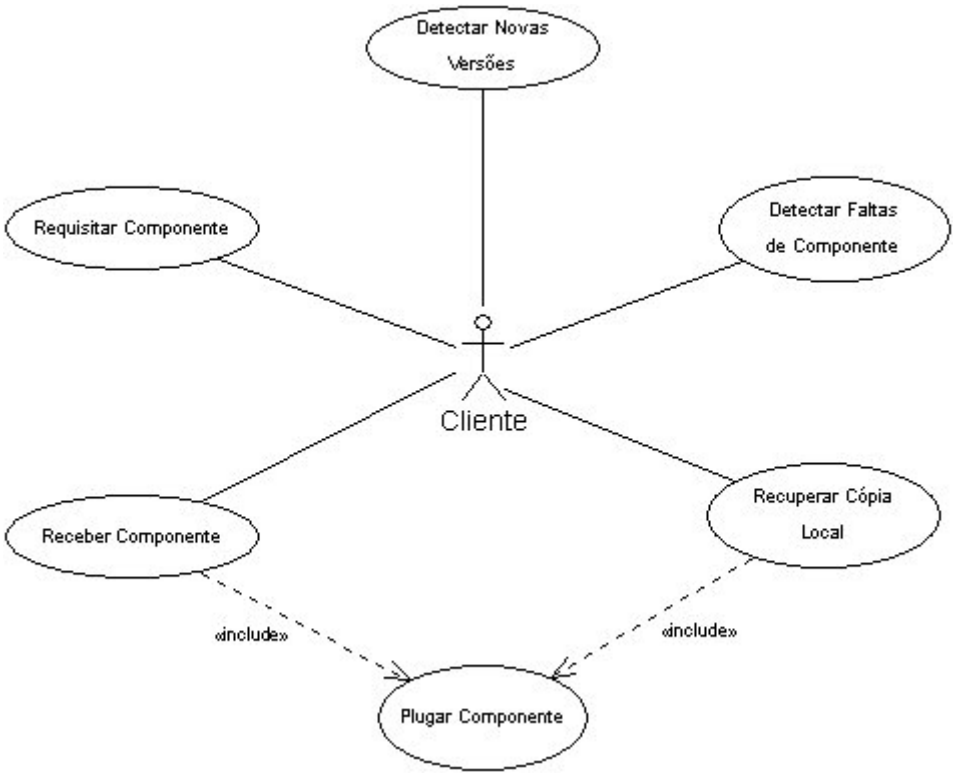

Figura 3.11 – Diagrama de Casos de Uso – Cliente FPA

Assim, as principais tarefas do cliente enquanto ator são:

- Detectar Faltas de Componente;
- Requisitar Componente;
- Receber Componente;

- Plugar Componente.

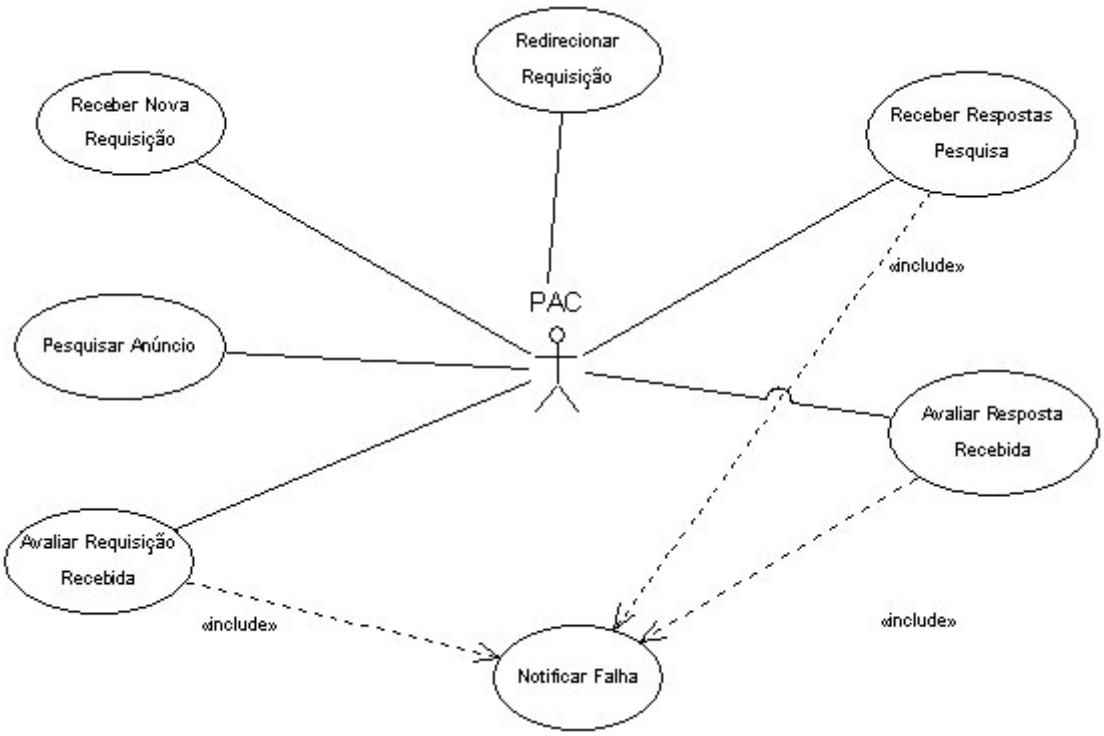

Figura 3.12 – Diagrama de Casos de Uso – PAC

As tarefas básicas do PAC compreendem:

- Receber Nova Requisição;
- Avaliar Cada Requisição Recebida;
- Avaliar Anúncios dos PAs;
- Redirecionar Requisições de Clientes aos PAs.

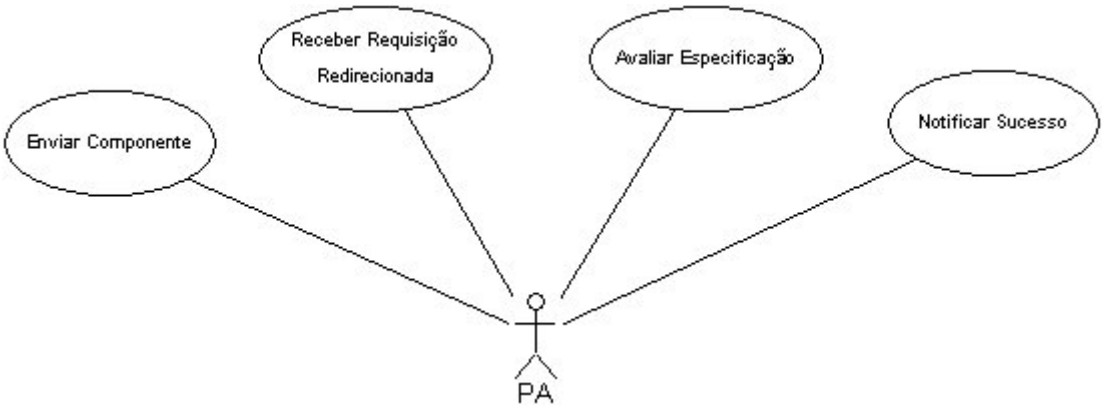

Figura 3.13 – Diagrama de Casos de Uso – PA

Pode-se visualizar na Figura 3.13 as principais tarefas do ator PA:

- Avaliar Especificação;
- Notificar o PAC da Existência do Anúncio Localmente;
- Enviar o Componente para o Cliente FPA;

Na Seção 4.2.6 serão vistas as sequências em que ocorrem os relacionamentos entre esses atores e suas tarefas.

3.4 Manutenção de Sistemas FPA

As principais questões relacionadas à manutenção de sistemas FPA dizem respeito ao ciclo de vida do Cliente FPA, às alterações de versão dos componentes e às alterações de versão do *kernel*.

É importante ressaltar que o que irá determinar o ciclo de vida de um Cliente FPA é o modelo de negócios definido entre o cliente e a FPA (Seção 3.5). Ao término do contrato ou no caso de um rompimento antes do prazo, será necessário se estabelecer mecanismos de controle, de maneira que o ex-Cliente FPA não tenha mais acesso à composição dinâmica. Uma saída para isto está no cabeçalho que cada requisição de componentes inclui. Ele contém um identificador exclusivo para aquele *peer* no ambiente P2P, e assim pode ser facilmente detectada a origem da mensagem. O PAC, então, poderia verificar esta

identificação com alguma espécie de registro pré-cadastrado por cada um dos clientes, no início da utilização do serviço.

As alterações de versão são operações praticamente idênticas às plugagens convencionais, a única diferença está no fato de que se irá plugar (ou não) um componente que já está presente localmente, após se verificar que uma nova versão sua foi disponibilizada pela FPA. Por isso, será necessária uma confirmação por parte do usuário final da aplicação, de que pretende ou não atualizar o módulo. Este usuário também poderá recusar a atualização, não acarretando assim, em nenhum prejuízo ao seu sistema.

A atualização do *kernel* é um pouco mais trabalhosa. Isso porque ela não se dará dinamicamente, mas manualmente pelo usuário da aplicação, que irá comprar uma nova versão e deverá instalá-la na sua máquina. Desta forma, é necessário que as novas versões sejam absolutamente compatíveis com os componentes já plugados nas versões antigas. O novo *kernel*, ao ser inicializado, irá aos poucos recompondo dinamicamente a estrutura da antiga aplicação. Isso porque os componentes que já haviam sido plugados ainda estarão presentes no Repositório Local de Componentes, não necessitando assim, de serem requisitados novamente à FPA.

Um aspecto de extrema relevância para a confiabilidade geral do modelo é a segurança. É ela quem garantirá que os componentes, transações, PAC, PAs, etc., estejam protegidos ao longo do ciclo de vida da plugagem, assim como garantirá a autenticidade dos componentes e das aplicações-cliente. No entanto, esse aspecto não foi coberto nesta dissertação pois está fora do seu escopo.

3.5 Modelo de Negócios FPA

Embora não esteja no âmbito desta dissertação os aspectos não técnicos-científicos associados ao modelo proposto, verificou-se que dele resultou um modelo comercial associado interessante e, porque não dizer, bastante promissor. Desta forma, optou-se por desenvolver superficialmente aspectos de um modelo de negócios que poderão vir a ser úteis para quem desejar levar o conceito adiante em termos comerciais, na ótica de *software* como serviço.

O modelo de negócios da Federação de Provedores de Aplicação (FPA) toma como base o modelo de negócios dos provedores de serviços de aplicação (PSAs / *Application Service Providers - ASPs*). Por isso, assim como aquele, comporta várias configurações e aceita diversas variações de contratos de utilização. No modelo PSA clássico, o cliente paga uma taxa fixa para usufruir dos serviços oferecidos pelo provedor. Esta taxa habilita licensas de utilização, e geralmente libera o cliente para usar o PSA livremente durante a vigência do contrato. É o mesmo caso dos provedores de Internet de banda larga, que por uma taxa mensal, disponibilizam o serviço a seus clientes durantes 24 horas por dia, podendo serem acessados irrestritamente por tantas vezes quantas o cliente desejar. O Anexo 1 mostra um exemplo de um típico contrato PSA.

No caso da Federação de Provedores de Aplicação, o serviço negociado envolve toda a transação, desde a busca pela especificação até a plugagem efetiva do componente. Mas apesar disso, vários itens do contrato PSA podem ser adaptados para o contrato FPA. Principalmente aqueles que envolvem garantias para o cliente, negociação das taxas personalizada, confiabilidade do serviço que será oferecido, possibilidade de alteração da licensa de utilização, confidencialidade, entre outros.

O modelo de negócios da FPA envolve dois contratos: um negociado entre cliente e provedor coordenador e outro acordado entre provedor coordenador e demais provedores.

Cada PA estará livre para oferecer componentes dos fabricantes que desejar, podendo inclusive ser ele próprio um desenvolvedor. O seu contrato com o PAC irá levar em conta a relevância desses componentes, a quantidade disponível, e a frequência com a qual novas versões dos mesmos serão disponibilizadas. Cada PA será remunerado por cada transação que participar como fornecedor. Isto significa que será importante a ele incrementar o seu repositório tanto quanto puder, para que o mesmo seja o responsável por um grande número de especificações anunciadas na federação. Além da busca padrão, uma outra forma de renda de um provedor de aplicação virá através das novas versões dos seus componentes. Cada cliente que possuir um componente seu plugado irá procurar, frequentemente, por uma nova versão do mesmo dentro da FPA. Cada nova transferência resultará num ganho extra para este PA. A política de procura por atualizações estará devidamente formalizada no contrato do cliente. O PA só se relaciona com o PAC. Não é de seu conhecimento nenhum detalhe específico sobre outros PAs ou sobre clientes da federação. Em relação aos clientes, tudo o que os PAs conhecem é o endereço da porta de

entrada por onde deverão ser recebidos os componentes. Desta forma, pode-se dizer que os PAs podem competir entre si e que o resultado desta possível competição será sempre uma maior oferta e atratividade da própria FPA.

O contrato entre o PAC e o cliente é um pouco diferente. Ele formaliza as opções de utilização de componentes pelo cliente, porém sempre considerando a ubiquidade da fonte provedora dos mesmos. Isto quer dizer que cada cliente irá pagar para se relacionar com a FPA, e não com determinado provedor de aplicação em especial. Seu contrato de utilização é bastante flexível, indo desde a pura compra do módulo (o que não seria uma opção interessante, pois denota uma subutilização da federação), passando pela plugagem simples, pela plugagem com direito a atualizações automáticas e chegando ao pagamento por tempo de utilização (com direito ou não a atualizações posteriores). O tempo de utilização poderá ser concebido de duas maneiras:

- Através de um intervalo de tempo pré-determinado: O componente poderia ser utilizado durante um prazo fixo (por exemplo, um determinado número de dias);
- Dentro de um prazo com um limite máximo: Neste caso a utilização continua tendo um prazo delimitador, mas poderia ser finalizada antes do tempo final, reduzindo-se assim a quantia paga à federação.

Dois itens caracterizam um Cliente FPA: uma aplicação desenhada para para utilizar a composição dinâmica e o módulo gerenciador de componentes, conforme foi visto na Seção 3.2.3. Entre os subitens do módulo gerenciador de componentes encontra-se o *Tratador de Faltas*. Ele funciona como o representante do PAC dentro do cliente, é o grande coordenador dentro do módulo. Este tratador é acionado sempre que uma função dinâmica é acionada. Sua função é verificar se o componente existe localmente, e, se este está apto a ser utilizado pela aplicação cliente. O tratador de faltas será o validador do módulo dentro do intervalo de tempo previamente estabelecido. O componente deverá trazer junto com a sua especificação o seu prazo de validade. Desta forma, o cliente poderá utilizar este componente sem problemas até que o tempo de utilização expire.

O outro ítem característico de um Cliente FPA é a sua aplicação. A aplicação núcleo precisa ser prévia e especificamente projetada para suportar a composição dinâmica. Pode-se imaginar essa aplicação como uma “árvore”, que possui apenas o tronco e seus galhos, sem as folhas. O “tronco” representa o *kernel* da aplicação, os “galhos sem as folhas”

significam que a árvore é passível de tê-las ali, ou seja, que a aplicação ainda não possui, mas está apta a implementar determinada funcionalidade. Os componentes seriam estas “folhas”, projetadas para se encaixar apenas no seu “galho de origem”. Cada “galho” trás consigo a especificação exata da “folha” que deve ser encaixada ali. E este “encaixe” nada mais é do que a plugagem. Para que este cenário torne-se realidade, é preciso que se crie uma nova cultura de desenvolvimento e de utilização de *software*. É necessário que os desenvolvedores passem a criar seus novos produtos utilizando uma análise de requisitos voltada ao desenvolvimento baseado em componentes. Uma vez criadas, estas aplicações *kernel* sairíam dos fabricantes com diversos *plugs*, aptos a receberem o componente que foi propositalmente desenvolvido separadamente. Tanto os *plugs* quanto os componentes conheceriam a mesma especificação, que deve ser criada pelo desenvolvedor. Assim sendo, quando uma aplicação deste tipo for disponibilizada no mercado, ela trará junto todas as especificações que forem necessárias para plugar as funcionalidades que não foram acopladas ao seu módulo principal. Da mesma forma, cada componente que for disponibilizado em um PA deverá vir já do fabricante com a sua especificação pronta, idêntica à existente na aplicação cliente (Figura 3.14).

De qualquer forma, observa-se que isso pode, de fato, se tornar uma tendência quando se verifica as iniciativas de AOS (Seção 2.2.3), que neste momento se constitui numa das mais recentes tecnologias em termos de projeto de sistemas flexíveis e orientados a serviços.

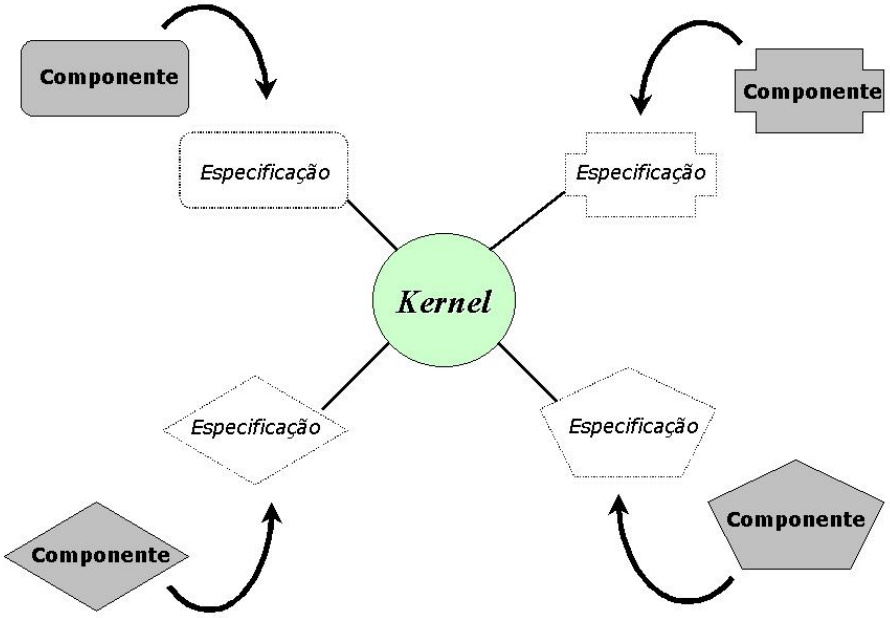

Figura 3.14 – A Aplicação Cliente

Existe também a questão do ClienteAgrupado. Nesta configuração, um ou mais Clientes FPA se juntam e negociam um contrato de utilização de componentes onde todos podem compartilhar dos componentes plugados anteriormente por qualquer um, dentro do prazo de validade. Neste caso, após cada plugagem, uma cópia do componente é transferida para o RLC, que irá ser verificado periodicamente (Seção 3.2.3) pelos tratadores de falta. Esta configuração pode ser interessante quando existirem clientes semelhantes (mesma aplicação ou aplicações na mesma área de atuação) e que requisitem os mesmos tipos de componentes. Do ponto de vista do cliente, é uma maneira de compartilhar a taxa de utilização com outros clientes, e do ponto de vista do PAC, uma maneira de negociar uma taxa mais elevada para a utilização da FPA.

O PAC deverá também se manter atualizado em relação às aplicações que estiverem sendo desenvolvidas para a arquitetura FPA. Esta verificação deverá ser feita periodicamente, de maneira que o mesmo possa avaliar os PAs segundo o critério “relevância de componentes”. Isso porque os PAs que possuírem componentes desenvolvidos para as aplicações mais utilizadas (ou mais passíveis de serem utilizadas) acabarão sendo priorizados na composição dinâmica. Na verdade é um grande desafio à capacidade do PAC (seu corpo gerencial) de identificar novas oportunidades de negócios.

De maneira geral, o modelo de negócios FPA é dinâmico e adaptável ao tipo de cliente. E seu objetivo final reside na idéia básica da FPA: que se utilize apenas o que se necessita, se pague apenas pelo que se utilize, e somente pelo tempo em que se utilizar.

Capítulo 4

4. Protótipo

Este capítulo tem como objetivo descrever e mostrar o protótipo desenvolvido para validar a proposta conceitual de composição dinâmica de sistemas propostos no Capítulo 3.

Este capítulo divide-se em três partes: na primeira é descrito o cenário de aplicação que serviu de base para o protótipo. Depois, são apresentados os detalhes relacionados à sua implementação, tais como os requisitos do sistema e o modelo utilizado para construir o protótipo. A última parte se concentra nos testes efetuados, incluindo os critérios utilizados e uma avaliação dos resultados obtidos.

4.1 Cenário Base

Um contexto de Empresas Virtuais (EVs) e *Business-to-Business* (B2B) foi tomado como base para o desenvolvimento do protótipo. Uma EV pode ser definida basicamente como um conjunto lógico de empresas que, coordenada e colaborativamente, através de transações pela rede, atenderão a uma oportunidade de negócios, produzindo o bem pedido por um cliente (Rabelo *et al.*, 2002). O SC^2 (Rabelo *et al.*, 2002) é um sistema multiagente de suporte à gestão de EVs e cadeias de suprimento e foi tomado como base para a construção de um cenário que suportasse a composição dinâmica. Sendo um sistema complexo e composto de funcionalidades de variadas naturezas e requisitos, a idéia é a de avaliar a situação de o usuário do sistema (gestor da EV) não necessitar de todo o conjunto daquelas funcionalidades a todo tempo, mas apenas parte delas e quando, efetivamente, as necessitar. Portanto, compondo dinamicamente o sistema, sob demanda.

Para isso, utilizou-se um dos módulos do SC^2, o de geração e formatação de relatórios dirigidos, chamado *Adhoc Reports*. Ele é responsável por consultar na base de dados itens como ordens de venda, informações sobre membros, processos de negócios, entre outros. Uma vez carregados, estes dados são usados na criação de um relatório, segundo o tipo escolhido pelo usuário, numa filosofia de *Business Intelligence*. Depois de formatado, este relatório pode então ser visualizado. Existem vários tipos de consultas pré-definidas no *Adhoc Reports*, cada uma gerando um tipo diferente de relatório. Assim sendo, devido à natureza especializável deste módulo, optou-se por retirá-lo do sistema principal e adaptá-lo num processo de "enxugamento", do qual só restaram agregadas a ele as suas funcionalidades essenciais. Todo o resto foi transformado em componentes, de maneira que o *Adhoc Reports* se tornasse um Cliente FPA. A este novo módulo enxuto deu-se o nome de *AdHocReportsDinamico.*

Por se tratar de um módulo genérico, o *AdHocReportsDinamico* necessita ser especializado para funcionar (Figura 4.1). Neste cenário, três tipos de relatório foram escolhidos para serem componentizados: o relatório das ordens de venda (*SalesOrderReport*), o relatório das EVs-membro do sistema (*SCMemberReport*) e o relatório das ordens de entrega do SC^2 (*ShippingOrderReport*). Cada um desses transformou-se em um componente especialmente projetado para plugar-se ao kernel do *AdHocReportsDinamico*.

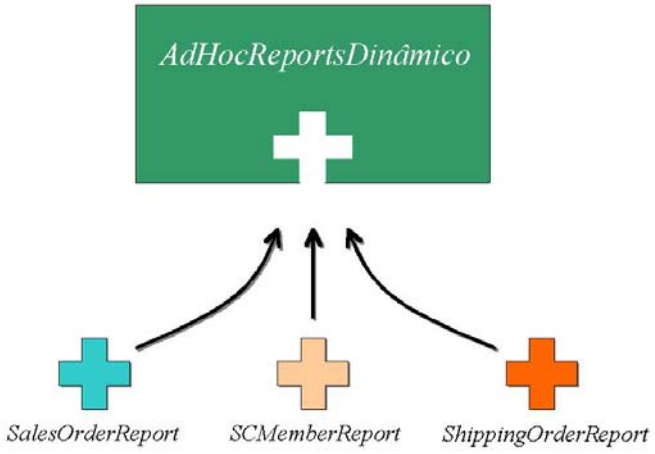

Figura 4.1 – Cenário Base

A tela inicial do protótipo é mostrada na Figura 4.2.

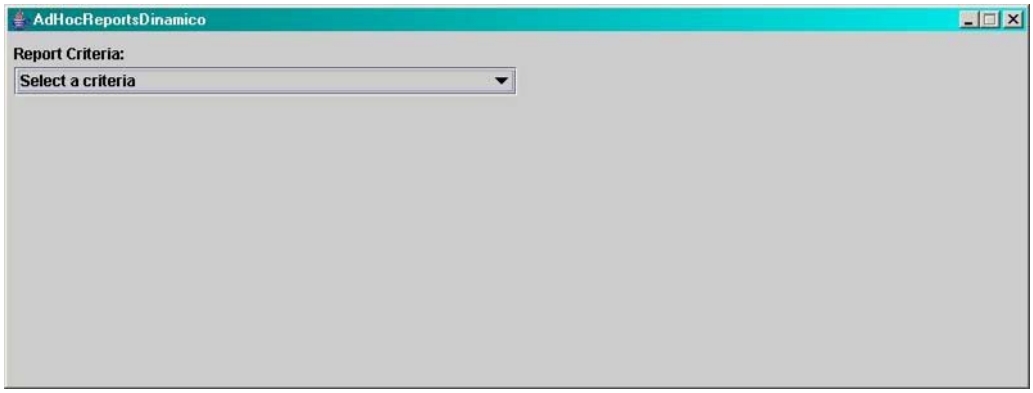

Figura 4.2 – Tela Inicial do Protótipo

A operação para o usuário consiste primeiramente da seleção de um critério para o tipo de relatório que se pretende gerar (Figura 4.3). Neste momento, o sistema detecta que não dispõe localmente do código associado à funcionalidade desejada (*component fault*) e então procura o componente na FPA e, uma vez encontrado, ele é plugado dinamicamente e de maneira transparente ao sistema (Figura 4.4). O sistema então executa a funcionalidade invocada e efetua a consulta na base dados para verificar os itens resultantes desta pesquisa (Figura 4.5). Finalmente, o usuário irá selecionar um item, que será formatado e visualizado (Figura 4.6), finalizando a operação. Neste caso, apenas um componente foi plugado, o associado à funcionalidade de cada tipo de *Adhoc Report*. Contudo, dependendo da implementação e granularidade desejados, cada uma dessas interfaces poderia estar sendo representada por um outro componente, criando-se uma cascata de invocações dinâmicas de componentes, onde um chamaria outro.

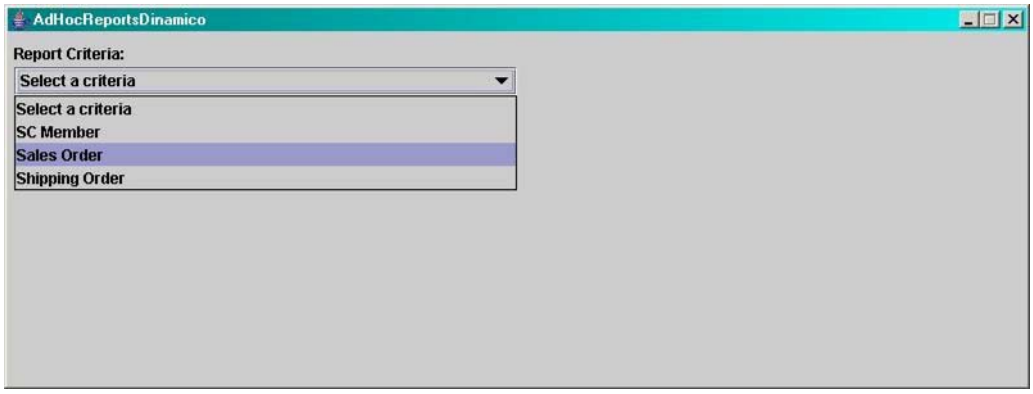

Figura 4.3 – Seleção do Tipo de Relatório

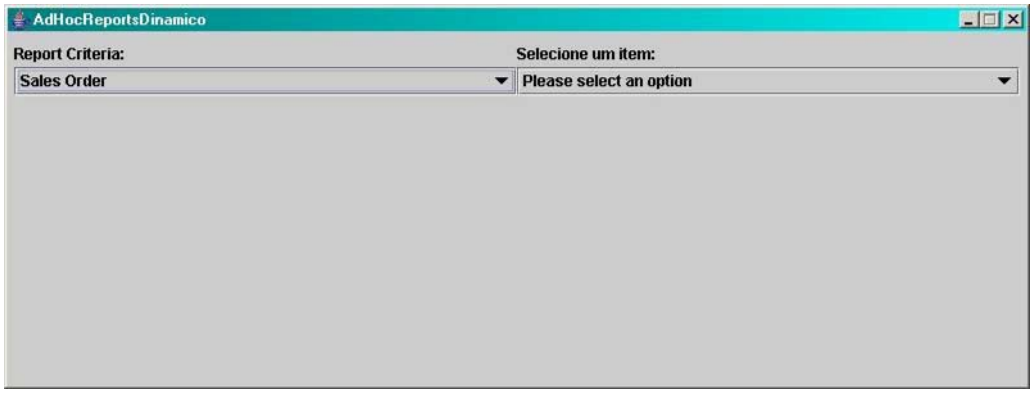

Figura 4.4 – Componente Dinâmico Plugado

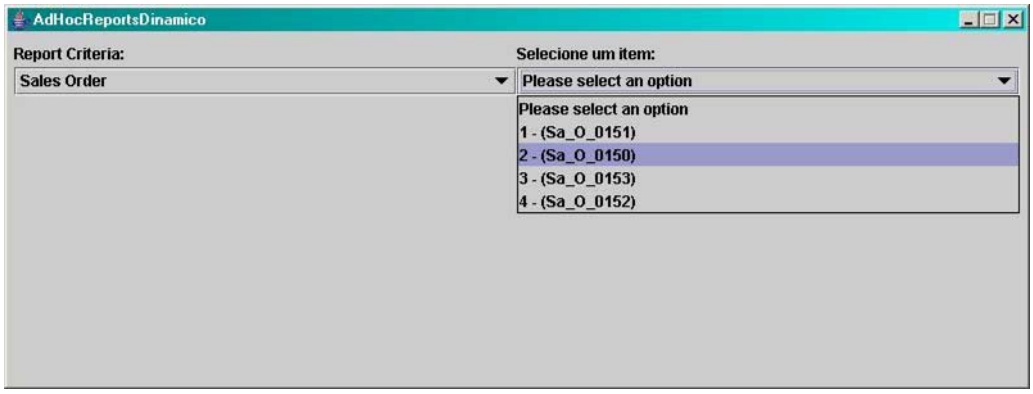

Figura 4.5 – Seleção de Item para Visualização

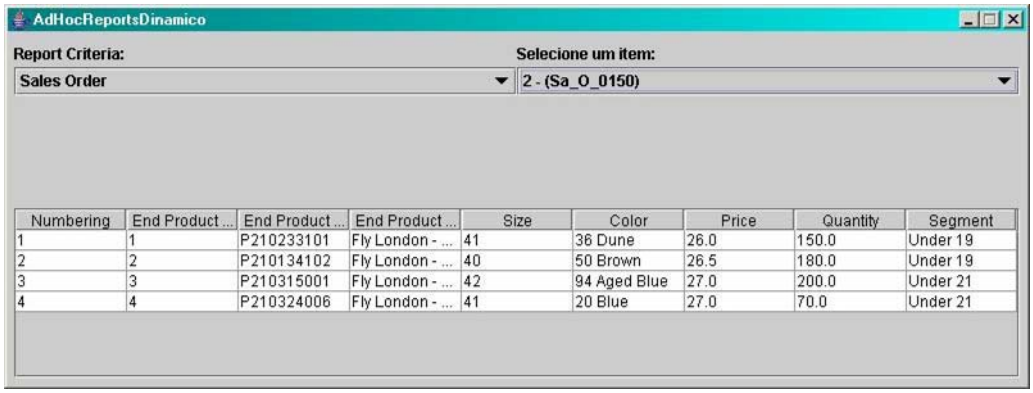

Figura 4.6 – Visualização do Relatório

Para este protótipo, foi considerado um modelo de negócios em que cada componente permanece plugado ao cliente enquanto este estiver em execução. Neste caso, a transação com a FPA é passível de ocorrer apenas três vezes, uma para cada tipo de relatório selecionado. O componente plugado será reutilizado em caso de nova seleção, ou

seja, está no RLC. Quando a aplicação é finalizada, os componentes eventualmente plugados são descartados. Apesar de simbolizar um cenário de relativamente baixa complexidade, este protótipo possibilitou a validação dos critérios selecionados para os testes (Seção 4.3).

4.2 Implementação

O primeiro passo a ser dado antes de se construir o protótipo foi a definição das metodologias e das ferramentas envolvidas neste processo. Depois disso pôde-se, então, instanciar um caso de teste. A seguir, são detalhados esses pontos.

4.2.1 Ferramentas Utilizadas

A base de dados utilizada foi a mesma do SC^2, Borland Interbase 6.0 (Borland, 2004). Foi utilizado também o driver JDBC FireBirdSQL 1.0.0 (Firebird, 2004) para comunicar-se com ela (arquivo *MyFashionDM.gdb*). Aproveitou-se desta estrutura (base de dados) já existente pelo fato de já existirem registros nela associados que serviriam como itens de pesquisa para o *AdHocReportsDinamico*. A plataforma de programação escolhida foi o Java SDK 1.4.2.02 (Java, 2004), por duas razões: primeiramente porque é a linguagem original do SC^2, o que facilitou as adaptações; e em segundo lugar, porque é a linguagem para o qual vêm sendo desenvolvidas as versões mais estáveis da plataforma JXTA (JXTA, 2004a), a plataforma P2P escolhida para o desenvolvimento (Seção 2.3.1). Para este protótipo, usou-se diversas versões do JXTA (a última utilizada foi a 2.1.1) para implementar a Plataforma de Suporte à Plugagem (Seção 3.2.3). O Windows XP (Microsoft, 2004) foi escolhido como sistema operacional para o protótipo, devido principalmente, ao fato da maioria das máquinas de testes o utilizarem. Como ambiente de programação foi utilizado o JCreator LE (JCreator, 2004), versão 2.5.

4.2.2 Modelo de Implementação

Definiu-se que o protótipo iria ser composto por três grandes objetos: o cliente, o PAC (Figura 4.7) e outro representando cada PA.

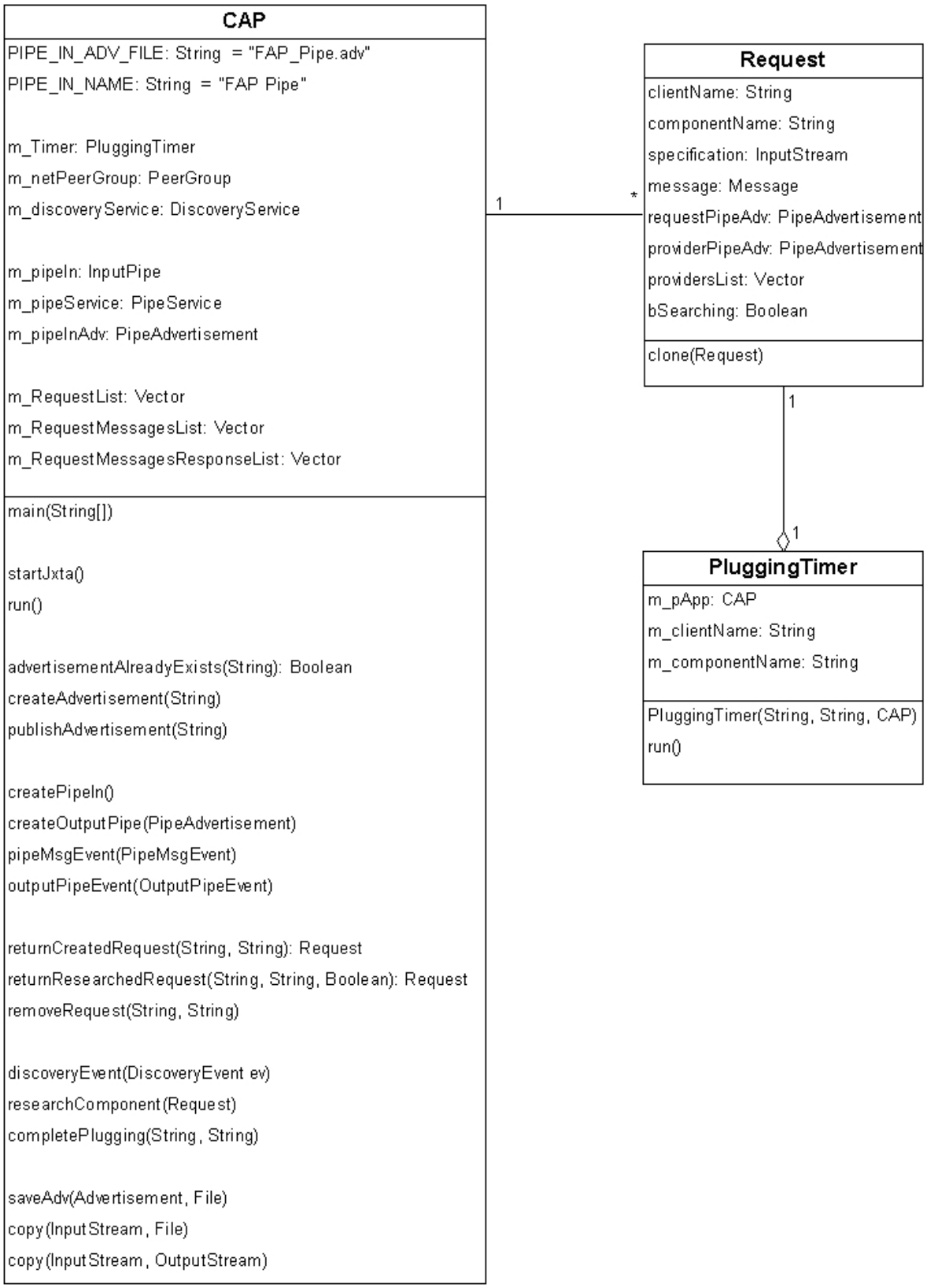

Figura 4.7 – Diagrama de Classes – PAC

Toda a interação inicial do cliente durante o processo de composição dinâmica se dá com o PAC, o *peer* com o qual ele tem acesso direto. Na Figura 4.7 pode ser visto o diagrama de classes do PAC. Este diagrama é composto por três classes, a que representa o PA Coordenador (*CAP*) e mais duas classes associadas.

A classe *Request* instancia um objeto sempre que uma nova requisição é recebida. Cada objeto criado desta classe será armazenado em uma lista do PAC (*m_RequestList*) de maneira a facilitar a manipulação de múltiplas requisições simultâneas. O objeto *Request* armazena inicialmente informações como nome do componente requisitado, nome do cliente que requisitou, o anúncio (Seção 4.2.3) do seu canal de comunicação e a especificação a ser pesquisada. Após a execução da pesquisa pelo PAC, este objeto irá guardar também uma lista dos provedores que responderam positivamente à busca pelo anúncio do componente. Existe uma classe agregada à classe *Request*, a classe *PluggingTimer*. Esta será responsável por definir um *timeout* para a pesquisa do componente. Este *timeout* é importante para evitar que a requisição e o cliente que a efetuou permaneçam inativos indefinidamente.

O PAC também possui atributos relativos à plataforma JXTA (*m_netPeerGroup*, *m_discoveryService*, *m_pipeService*) e relativos a seu canal de comunicação (*PIPE_IN_ADV_FILE*, *PIPE_IN_NAME*, *m_pipeIn*, *m_pipeInAdv)*. Seus métodos são operações que inicializam a plataforma JXTA; gerenciam o seu canal de comunicação; procuram pelo anúncio do componente dentro da FPA; gerenciam as mensagens trocadas com os PAs e com os clientes; salvam em arquivos as especificações recebidas nas mensagens dos clientes.

A Figura 4.8 mostra o diagrama de classes dos PAs. Aqui também existe uma classe *Request* associada. Isso porque um PA interage com o PAC em dois momentos. No primeiro, ele notifica ou não o PAC de que possui o componente pesquisado. Se possuir e se for escolhido pelo PAC como a melhor opção, ele deverá então efetuar a transferência do componente diretamente para o cliente. Neste caso, recebe a requisição do cliente redirecionada pelo PAC. Baseado nos dados desta requisição, o PA então irá criar uma mensagem para o cliente, que levará em anexo o arquivo contendo o módulo de *software* propriamente dito. Quando a transferência for finalizada, a tarefa do PA no processo de composição dinâmica estará encerrada. Assim com o PAC, os PAs também possuem uma lista de requisições para casos de transações simultâneas.

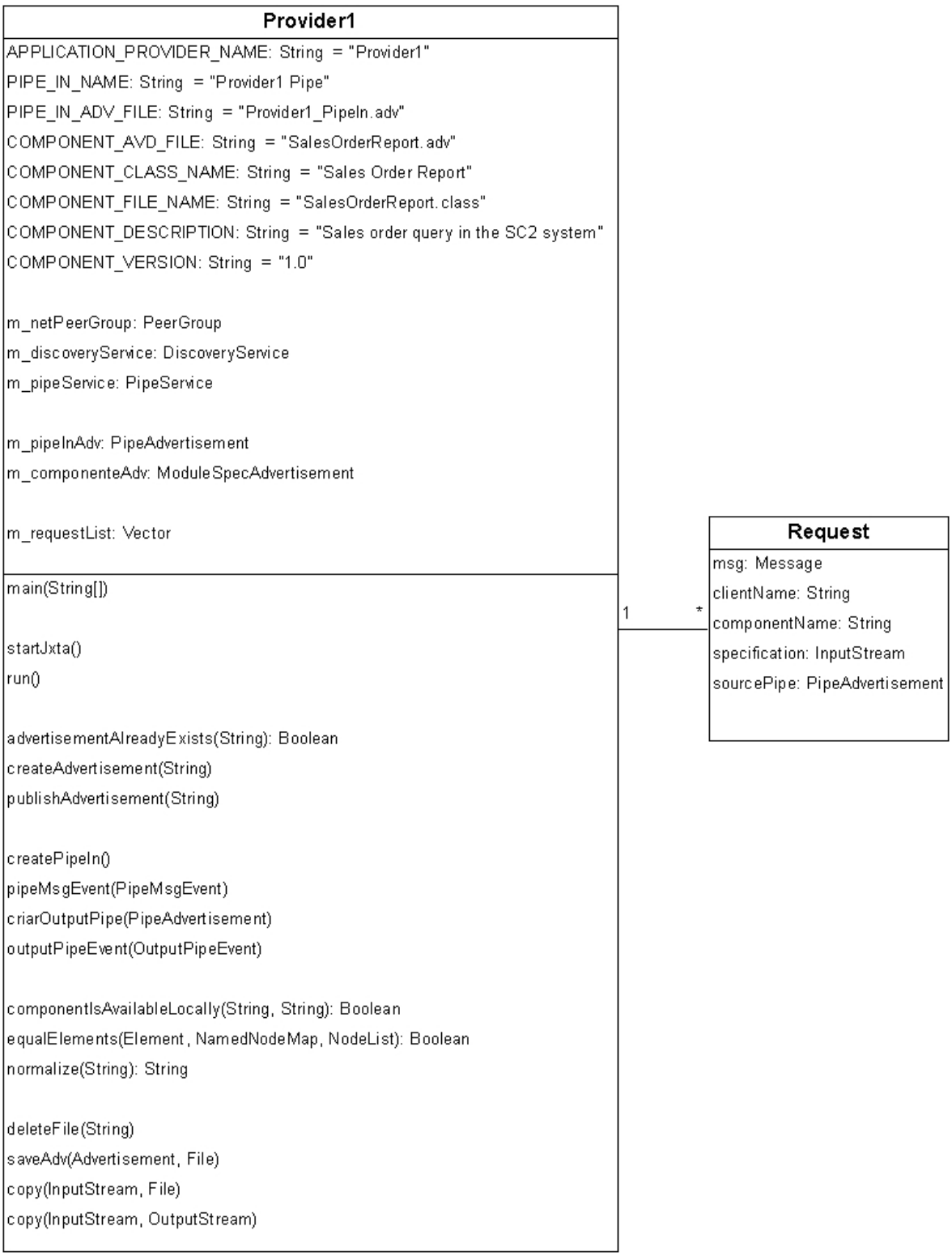

Figura 4.8 – Diagrama de Classes – PA

A Figura 4.9 mostra o diagrama de classes do Cliente FPA.

Figura 4.9 – Diagrama de Classes – AdHocReportsDinamico

O diagrama de classes do cliente é composto por três classes e por uma interface. A maior delas (*AdHocReportsDinamico*) representa a aplicação cliente. As outras duas dizem respeito ao componente e a seus elementos internos, e são representadas neste caso pelo componente *SalesOrderReport*. A interface *AdHocReportsInterface* é o elo que une o *kernel* do cliente ao componente dinâmico. A interface é pensada na modelagem da aplicação cliente e é incluída na sua implementação. Para que a conversão de tipos (*type casting*) entre ela e o módulo a ser plugado seja perfeita, é necessário que suas assinaturas (cabeçalhos de métodos e tipos de parâmetros) estejam perfeitamente de acordo. É para definir esta compatibilidade que existe a especificação do componente, que deverá ser criada pelo desenvolvedor da aplicação e estar de acordo com a do desenvolvedor do componente no caso de não serem o mesmo.

A classe que representa o componente (*SalesOrderReport*) é composta por dois métodos: *ListItems* e *QueryItem*. O primeiro apresenta ao usuário da aplicação as opções de itens existentes para a seleção efetuada por ele (Figura 4.5). O outro método, *QueryItem*, irá pesquisar na base de dados o registro referente ao item selecionado pelo usuário e mostrá-lo na tela do protótipo (Figura 4.6). Toda a aplicação cliente é construída com referências explícitas a esses dois métodos que, contudo, só são de fato instanciados após a composição dinâmica. O acesso a esses dois métodos pela classe *AdHocReportsDinamico* está restrito à condição de haver componente plugado. Antes de se executar a chamada *ListItems* é feita uma verificação para se certificar de que o arquivo do componente está presente no disco e que seu conteúdo está carregado em memória. Quando algum problema ocorre e o módulo não é carregado na memória do programa, convencionou-se dizer que ocorreu uma falta de componente (*component fault*). Portanto, o que determina ou dispara o processo de composição dinâmica é a ocorrência de *component faults*.

Pelo fato de os três componentes construídos (*SalesOrderReport*, *SCMemberReport* e *ShippingOrderReport*) possuírem formatações diferentes para a visualização de dados, foi necessário definir-se um mecanismo de adaptação para que não houvesse incompatibilidade após diferentes plugagens. A saída encontrada foi deixar que cada componente indicasse que formato de dados utilizaria. Desta forma, o método *ListItems* retorna um parâmetro que indica quantas são as colunas da tabela a ser montada e qual é a

disposição dos seus campos. Através desta informação o *AdHocReportsDinamico* pode, então montar a correta visualização do relatório.

Como pode ser visto na Figura 4.9, a classe *AdHocReportsDinamico* possui diversos atributos relacionados à interface do protótipo (*m_Table*, *m_panelCombos*, *m_labelItem*, *m_StatusText*, *m_scrollpane*, entre outros). E da mesma forma que ocorre no PAC e nos PAs, existem também atributos relacionados à plataforma JXTA. É importante ressaltar, porém, a existência do atributo *m_AdHocReportsDinamicoInterface*. Ele representa a interface genérica do componente e é quem será especializado durante a composição. Entre os métodos desta classe pode-se destacar o que procura pelo anúncio da FPA (PAC), *findOutFPA_Advertisement*, e os dois que interagem com o componente plugado, *generateReport* e *itemStateChanged*.

Existe também um método chamado *removeComponent*. Ele seria o responsável por remover o arquivo físico do componente e, desta forma, gerar um *component fault* durante a próxima tentativa de utilização. Infelizmente esta tentativa não obteve sucesso porque, nem a técnica utilizada para carregar o componente de Liang *et al.* (1998) e nem a linguagem Java oferecem uma maneira de retirar da memória uma classe já alocada na Máquina Virtual Java, mesmo que se remova o arquivo do disco. Esta é a razão pela qual foi definido para este protótipo um modelo de negócios onde os componentes são plugados apenas uma vez. Para que uma nova composição com o mesmo componente possa ocorrer é necessário reinicializar-se a aplicação cliente. Entretanto, o conceito de “desplugagem” foi elaborado, apesar das restrições tecnológicas quanto à sua implementação.

Os outros métodos do cliente são similares aos existentes no PAC e nos PAs, métodos para o gerenciamento da comunicação entre os *peers*, gerenciamento do canal de comunicação, pesquisa de anúncios JXTA, etc.

4.2.3 Anúncios

A plataforma JXTA baseia-se fortemente na utilização de anúncios (*advertisements*). Cada item existente no mundo JXTA deve possuir um anúncio associado de maneira que possa ser localizado. Os *peers*, os grupos de *peers*, os canais de comunicação (*pipes*), os serviços, os dados, entre outros, podem ser localizados no ambiente JXTA através de

anúncios. É desta forma que esta plataforma *peer-to-peer* implementa a sua pesquisa por conteúdo.

Devido a este fato, os anúncios JXTA foram o meio escolhido para especificar os componentes dinâmicos. Um anúncio é, primordialmente, um arquivo XML. Este padrão é usado para especificar as características daquilo que se quer anunciar. No caso do canal de comunicação de entrada (*pipe in*), o anúncio irá conter o seu identificador no mundo JXTA (que é exclusivo), o tipo de *pipe* utilizado (*unicast*, por exemplo) e o nome do *pipe*. Estas informações serão utilizadas como referência de pesquisa (*tags*) para quem desejar localizar este *pipe*. Estes arquivos XML devem ser publicados dentro do ambiente JXTA para que as suas informações tornem-se conhecidas. A publicação pode ser de duas maneiras: local ou remota.

A publicação local consiste em se enviar uma cópia do arquivo para um determinado diretório local, que deverá ser o primeiro a ser consultado em caso de uma busca, procurando-se assim diminuir o *overhead* de processamento desta transação. No caso da publicação remota, deverão ser enviadas cópias dos anúncios para os *peers* JXTA responsáveis pelo armazenamento de informações, os *rendezvous*. Os *rendezvous* são pontos de referência JXTA que outros *peers* utilizam quando efetuam as suas buscas por conteúdo. As pesquisas JXTA são feitas através de saltos entre os *rendezvous*, até que se alcance um limite máximo de saltos ou de tempo. Assim, um *peer* que pretende anunciar uma informação deverá fazê-lo tanto localmente quanto nos *peers rendezvous* que forem de seu conhecimento.

No caso deste protótipo, isto se tornou uma tarefa relativamente simples. Por se tratarem de apenas cinco *peers* JXTA (PAC, PA, Provedor1, Provedor2, Provedor3), definiu-se que o PAC é o único *rendezvous* do sistema. Assim, todos os outros *peers* devem publicar no PAC seus canais de comunicação e, no caso dos PAs, também as especificações de seus componentes. Um anúncio JXTA de um componente dinâmico neste protótipo é composto de duas partes (Seção 3.1.1):

- Um arquivo XML contendo o nome do componente, o provedor onde ele está localizado e o identificador do seu canal de comunicação (Figura 4.10);

- Outro arquivo XML que deve conter todas as informações que o caracterizem enquanto pacote de *software*, dispostas de maneira que possam ser interpretadas e comparadas pelo PAC (Figura 4.11).

Todavia, diferentemente da linguagem humana, a especificação utilizada no protótipo é bem mais simples. Detalhes como restrições e tipagem de dados (encontradas nos diagramas UML) foram deixados de lado. Concentrou-se exclusivamente no nome do pacote, versão, arquivo, linguagem humana, linguagem de programação, processador e tipo de sistema operacional. Esta decisão foi tomada para que se pudesse utilizar o arquivo de definições de pacotes padrão da OMG (*softpkg.dtd*) (SPD, 2004), que é utilizado no *CORBA Component Model*. Trata-se portanto, de um arquivo público, conhecido e bem acessível.

```
<?xml version="1.0"?>
<!DOCTYPE jxta:MSA>
<jxta:MSA xmlns:jxta="http://jxta.org">
      <MSID>
            urn:jxta:uuid-66F43FAF2945407A83CC9EA0DA5C0CAE0A3498A20FBC47449C4F8FEEB559C3E606
      </MSID>
      <Name>
            SalesOrderReport.class
      </Name>
      <Crtr>
            Provider1
      </Crtr>
      <SURI>
      </SURI>
      <Vers>
            1.0
      </Vers>
      <jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
            <Id>
                  urn:jxta:uuid-59616261646162614E5047205032503317AAFCDCABE64D4C94EF8780C671F4D004
            </Id>
            <Type>
                  JxtaUnicast
            </Type>
            <Name>
                  Provider1 Pipe
            </Name>
      </jxta:PipeAdvertisement>
</jxta:MSA>
```

Figura 4.10 – Anúncio do Componente SalesOrderReport

```
<?xml version="1.0"?>
<!DOCTYPE softpkg SYSTEM "softpkg.dtd">

<!— Software package descriptor for dynamic component to be published in the FAP —>

<softpkg name = "SalesOrderReportPKG" version = "1,0">
  <title>SalesOrderReport</title>
  <pkgtype>Dynamic Component</pkgtype>

  <author>
      <company>G-Sigma - DAS - UFSC</company>
      <webpage href="http://www.gsigma-grucon.ufsc.br/english"/>
  </author>
  <description>Component used to generate a report of the SC2 sales orders</description>

  <implementation>
      <os name = "WinNT" version = "4,0,0,0"/>
      <os name = "Win95"/>
      <processor name = "x86"/>
      <compiler name = "javac"/>
      <programminglanguage name = "Java"/>
      <language name = "EN"/>
      <code type = "Java Class">
            <fileinarchive name = "SalesOrderReport.class"/>
      </code>
  </implementation>


</softpkg>
```

Figura 4.11 – Especificação XML do Componente SalesOrderReport

Ao receber estes dois arquivos, o PAC os compara com a requisição feita pelo Cliente FPA. O componente precisa preencher todos os requisitos especificados pelo cliente para poder ser transportado até ele. Quando uma ou mais especificações estão de acordo, o PAC precisa escolher aquela que será utilizada. Para isso, ele verifica a versão de cada um dos pacotes e analisa também os dados relativos à acessibilidade do cliente pelo PA (*ping*) que cada PA efetuou antes de responder à pesquisa. Com estes dados, o PAC pode então decidir que especificação será utilizada.

A utilização de anúncios se adequa perfeitamente ao modelo da FPA. O grande diferencial das arquiteturas *peer-to-peer* está justamente na maneira como as mesmas efetuam as suas buscas por conteúdo, algumas mais eficientemente do que outras. Através de anúncios, o JXTA implementa esta busca de uma forma bastante eficaz, o que no caso da FPA contribui para o processo de pesquisa por componentes dinâmicos.

4.2.4 Requisitos

Depois de definidas as ferramentas de construção do protótipo era preciso que se definisse um ambiente de utilização que pudesse ter suas características adaptadas para a composição dinâmica. O módulo *AdHocReports*, do sistema SC^2, com sua natureza especializável, se mostrou bastante apropriado. Decidiu-se então que seria construída uma nova aplicação, que iria agrupar apenas este módulo, exportado do sistema principal. Assim, podería-se testar apenas as tarefas que o envolvessem, sem sofrer interferências do resto do sistema. Entretanto, uma vez testado e aprovado, nada impede que após a sua validação o *AdHocReportsDinamico* possa ser introduzido no SC^2 no lugar de seu correspondente tradicional.

A granularidade dos componentes do *AdHocReportsDinamico* foi definida depois que se separou a funcionalidade básica da funcionalidade agregável. A cada novo relatório gerado, o usuário do *AdHocReports* (SC^2) precisa selecionar primeiramente o tipo de relatório e depois um dos itens que representam este tipo. Ou seja, as tarefas de escolha de tipo e de item são pertinentes a qualquer geração de relatório em que se utilize o *AdHocReports*. Cada uma destas escolhas significa uma consulta à base de dados (BD) do SC^2. Por exemplo, quando o usuário seleciona o critério *Sales Order*, uma pesquisa é feita na base para identificar todas as ordens de venda do sistema. Feito isso, estes registros são adicionados à lista de itens selecionáveis e o usuário, então, decide qual registro deve ser visualizado. No momento em que se seleciona um item, uma nova consulta à BD é feita para retornar os detalhes daquela ordem de venda que serão mostrados ao usuário. Estes dados são mostrados através de uma tabela, que é montada dinamicamente após a última consulta à BD (Figura 4.6). Os campos e o formato desta tabela dependem exclusivamente do tipo de relatório que se estiver gerando.

Pôde-se identificar claramente, após esta etapa inicial, como funcionalidade básica, a interface do protótipo e, como funcionalidades agregáveis, a consulta na base de dados e a formatação da tabela de visualização. Desta forma, concluíu-se que estas duas últimas deveriam estar dispostas na forma de componentes. E pôde-se também visualizar um cenário de utilização em que a aplicação *kernel* (*AdHocReportsDinamico*) utilizaria componentes dinâmicos para interagir com o SC^2 (Figura 4.12).

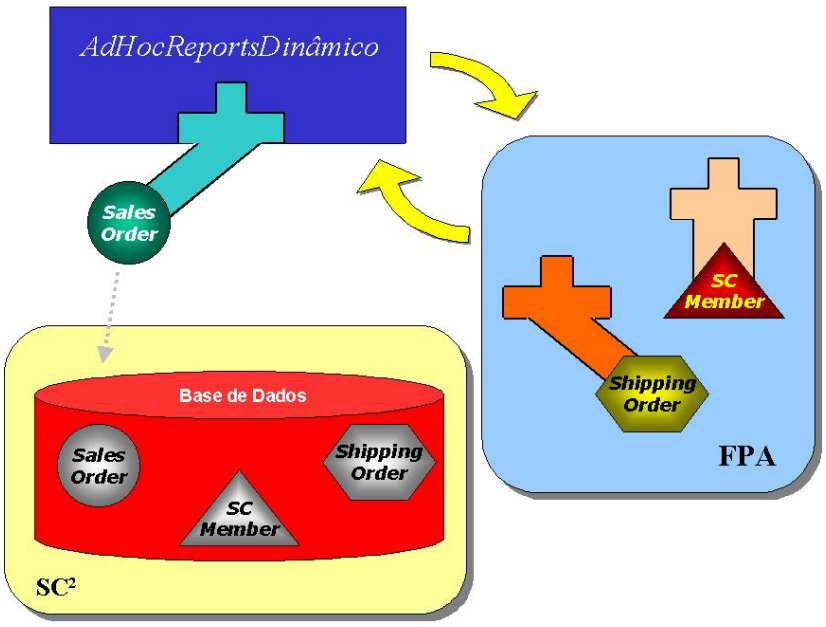

Figura 4.12 – Interação entre Protótipo e SC²

Portanto, cada componente criado passou a ser composto de duas interfaces, uma que lista os itens disponíveis no tipo de relatório escolhido, e outra que gera a visualização dos detalhes deste item (tabela dinâmica). Feito isto, a próxima etapa foi dotar o *AdHocReportsDinamico* de uma estrutura que possibilitasse a plugagem dinâmica destes componentes, pois assim ele poderia reportar-se a um componente genérico, através de uma linguagem genérica, que seria reconhecida por cada um dos três componentes dinâmicos construídos. Assim, definiu-se a granularidade destes componentes em duas funções, que deveriam possuir as mesmas declarações e parâmetros, de maneira a se tornarem genéricas (exemplo em Java):

- *ListItems(Vector columnsVector)*;
- *QueryItem(String scCode, String tableCode).*

Todavia, apenas as assinaturas das funções são mesmas (Figura 4.13); suas implementações são bem diferentes dentro de cada componente (Figura 4.14).

```
import java.util.Vector;

class TableElement
{
    String scCode;
    String itemName;
    String tableCode;
    String originalCode;
}

public interface AdhocReportsInterface
{
    public Vector ListItems(Vector columnsVector);
    public Vector QueryItem(String scCode, String tableCode);
}
```

Figura 4.13 – Interface Genérica AdhocReportsInterface

```
import java.util.Vector;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import javax.swing.JOptionPane;
import br.ufsc.gsigma.util.DBConnectionHolder;

class SalesOrderReport implements AdhocReportsInterface
{
    int scCode = 73;

    public Vector ListItems(Vector columnsVector)
    {
        Vector pVector = new Vector();

        try
        {
            Statement ts = DBConnectionHolder.getConnection().createStatement();

            String str;
            pVector.removeAllElements();
            pVector.addElement(""); //to synchronize with the combobox indexes

            ResultSet rSet = ts.executeQuery("SELECT CD_SALES, CD_ORIGINALCODE"
                        + " FROM TABLESALES "
                        + "WHERE CD_SUPPLYCHAIN = "
                        + scCode);

            while (rSet.next())
            {
                TableElement sale = new TableElement();
                sale.scCode = String.valueOf(scCode);
                sale.tableCode = rSet.getString("CD_SALES");
                sale.originalCode = rSet.getString("CD_ORIGINALCODE");
                sale.itemName = (rSet.getString("CD_SALES") + " - (" + rSet.getString("CD_ORIGINALCODE") + ")");
                pVector.addElement(sale);
            }

            columnsVector.addElement("Numbering");
            columnsVector.addElement("End Product Code");
            columnsVector.addElement("End Product Original Code");
            columnsVector.addElement("End Product Description");
            columnsVector.addElement("Size");
            columnsVector.addElement("Color");
            columnsVector.addElement("Price");
            columnsVector.addElement("Quantity");
            columnsVector.addElement("Segment");
        }
        catch (SQLException e2)
        {
            e2.printStackTrace();
        }
```

Figura 4.14 – Componente SalesOrderReport – Função *ListItems*

Uma vez que estavam definidas as funcionalidades “componentizáveis”, suas granularidades e as ferramentas a serem utilizadas, o projeto estava pronto para começar a ser implementado.

4.2.5 Configuração de um Peer JXTA

Cada *peer* JXTA precisa ser configurado para suportar os protocolos JXTA. Esta configuração é feita apenas na primeira vez em que ele é executado, pois armazena as informações localmente para uso futuro. A Figura 4.15 mostra a primeira tela configuração, onde deve ser definido o nome do *peer* e as configurações de *firewall*, se estas existirem.

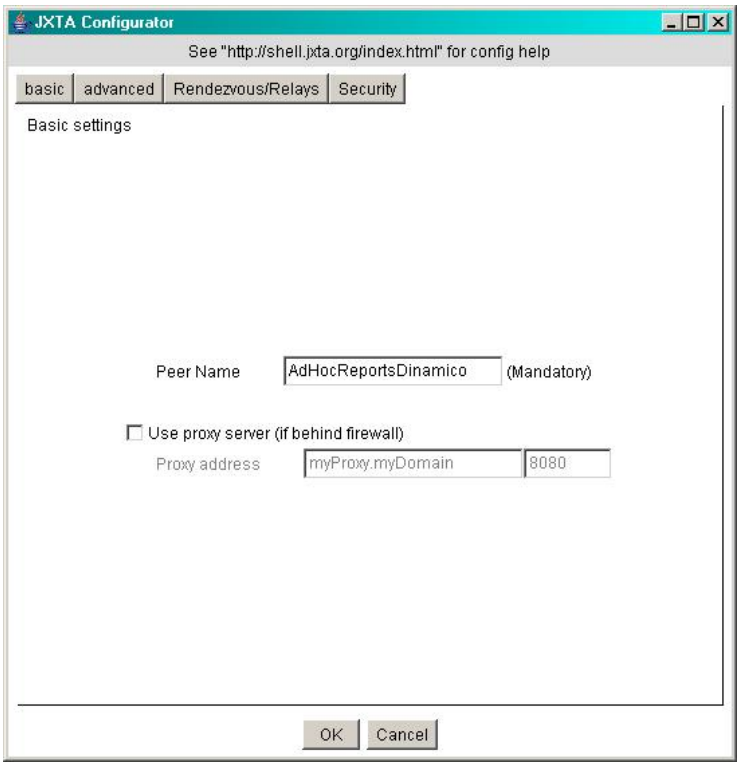

Figura 4.15 – Configuração *Peer* JXTA – *Basic*

A Figura 4.16 detalha as configurações dos canais de comunicação, incluindo o tipo de protocolo a ser utilizado (TCP e / ou HTTP), o endereço IP (*Internet Protocol*) aonde o *peer* se encontra e a sua porta de comunicação, que deverá ser exclusiva.

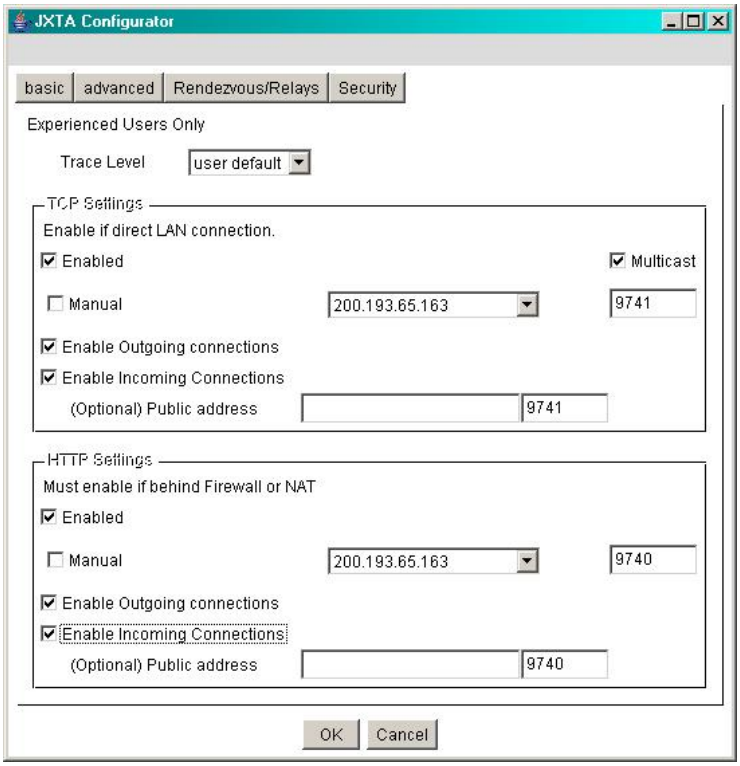

Figura 4.16 – Configuração *Peer* JXTA – *Advanced*

As configurações de *rendezvous* (*peers* que armazenam informações no sistema) e *relays* (*peers* de redirecionamento de mensagens) estão na próxima tela. Para o protótipo *AdHocReportsDinamico*, definiu-se que o *peer* representante do PAC ficaria sendo o *rendezvous* do sistema. A Figura 4.17 mostra a configuração para o *peer* cliente, portanto nele foi inserido o endereço e a porta do *peer* do PAC.

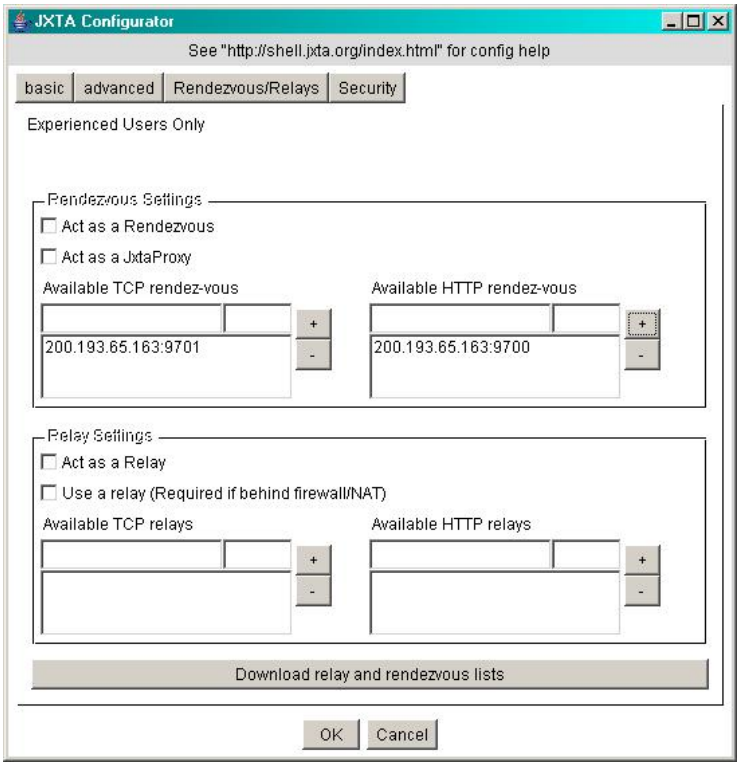

Figura 4.17 – Configuração *Peer* JXTA – *Rendezvous / Relays*

A última tela de configuração cadastra um nome e uma senha de usuário (Figura 4.18).

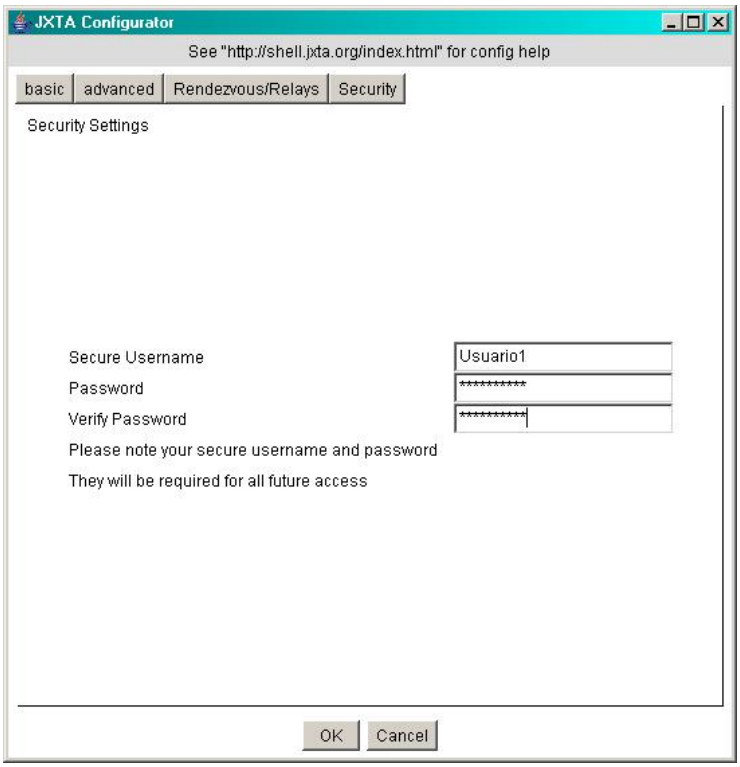

Figura 4.18 – Configuração *Peer* JXTA – *Security*

Após a primeira execução dos peers, não é mais necessário se entrar novamente com todas essas informações. O sistema apenas pede a confirmação do usuário e senhas cadastrados para cada um deles (Figura 4.19).

Figura 4.19 – *Login* no *Peer* JXTA

4.2.6 Fluxo de Execução

Nesta seção será abordado o fluxo de execução do *AdHocReportsDinamico*. O fluxo de execução diz respeito à sequência de operações que devem ser efetuadas para que uma transação ocorra no sistema.

Os testes com o protótipo foram realizados em duas configurações diferentes: em uma única máquina e em várias máquinas. Foram definidos cinco *peers* JXTA: três representando os PAs, um para o PAC e outro para o Cliente FPA. Cada *peer* é um processo independente, que é referenciado através de um endereço IP (*Internet Protocol*) e de uma porta de comunicação (*socket*). Os *peers* podem compartilhar o mesmo IP, porém nunca a mesma porta. Para funcionarem corretamente em uma mesma máquina, eles devem estar configurados para utilizar portas diferentes. A Figura 4.20 mostra o cenário do fluxo de execução do protótipo.

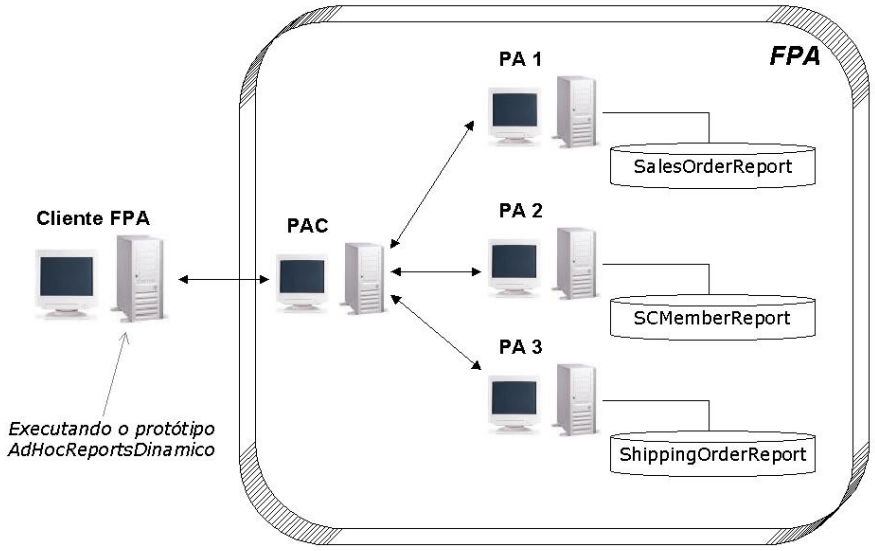

Figura 4.20 – Cenário do Fluxo de Execução

Cada entidade da FPA possui duas sequências de execução, uma para a inicialização do *peer* JXTA e a outra para comunicação com os outros *peers* durante a composição dinâmica. O fluxo de execução durante a inicialização é o seguinte:

- No PAC (Figura 4.21):
  - Inicialização do ambiente JXTA;

- Se não existe ainda o anúncio do *pipe in*, este é criado;
- Anúncio do *pipe in* é publicado local e remotamente;
- *Pipe in* é criado utilizando-se o identificador do anúncio publicado;
- *Receptor Cliente* é inicializado e o PAC irá aguardar por novas requisições.

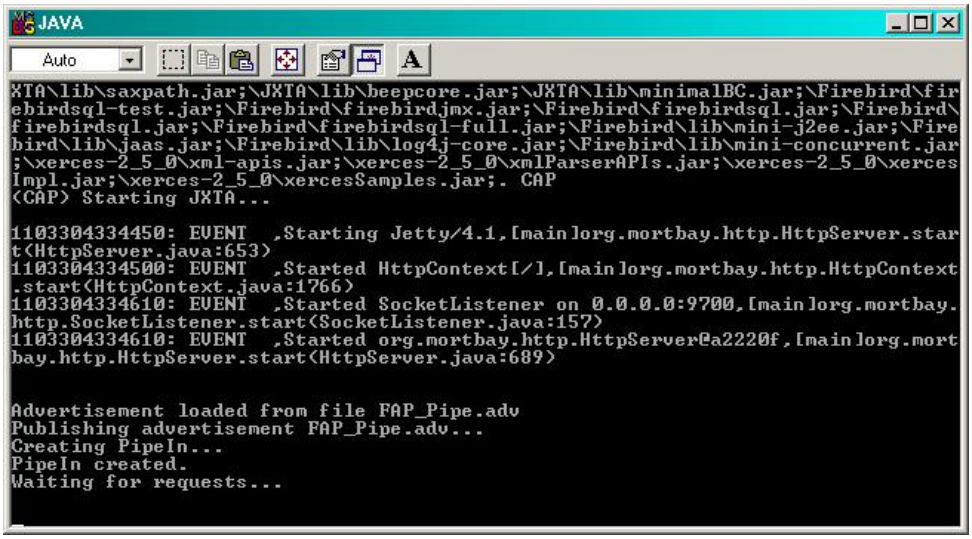

Figura 4.21 – Inicialização do PAC

- No PA (Figura 4.22):

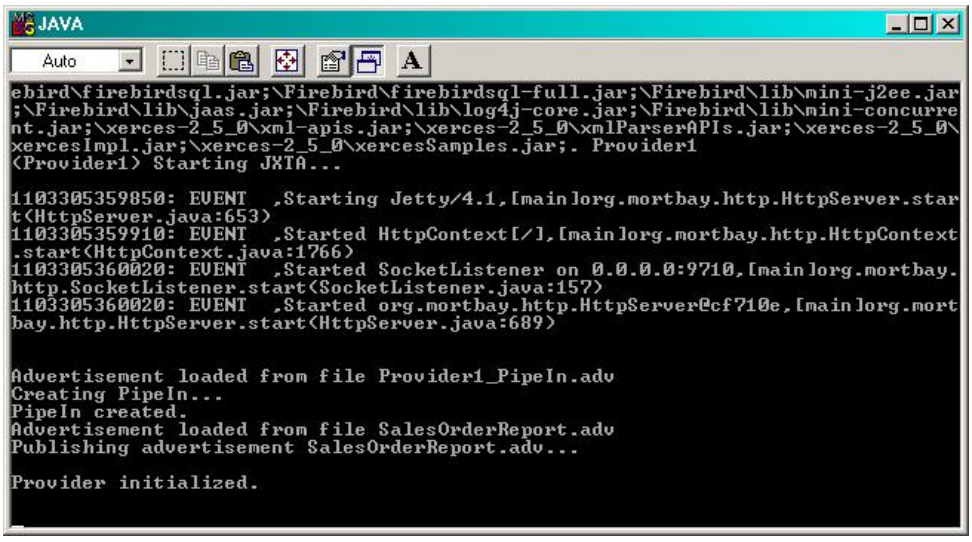

Figura 4.22 – Inicialização do PA

- Inicialização do ambiente JXTA;
- Se não existe ainda o anúncio do *pipe in*, este é criado;

- *Pipe in* é criado utilizando-se o identificador do anúncio criado;
- Se não existe ainda o anúncio do componente, este é criado;
- Anúncio do componente é publicado local e remotamente;
- *Receptor PAC* é inicializado e o PA irá aguardar por novos redirecionamentos de requisições.

- No Cliente FPA (Figura 4.23):
  - Inicialização do ambiente JXTA;
  - Se não existe ainda o anúncio do *pipe in*, este é criado;
  - *Pipe in* é criado utilizando-se o identificador do anúncio criado;
  - Interface do *AdHocReportsDinamico* é inicializada, e o sistema está pronto para ser usado.

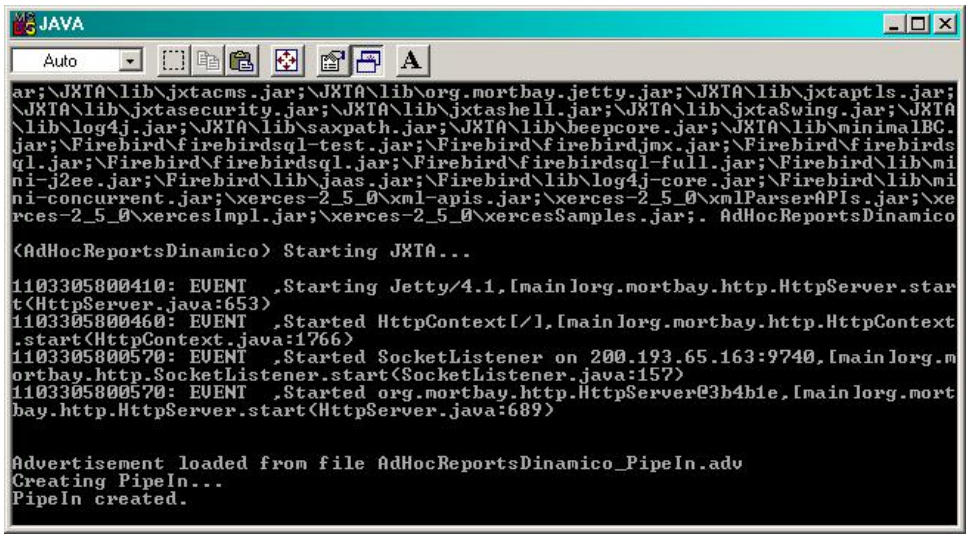

Figura 4.23 – Inicialização do Cliente FPA

A seguir, a Figura 4.24 apresenta o diagrama de sequência da arquitetura FPA durante uma composição dinâmica, baseado na UML.

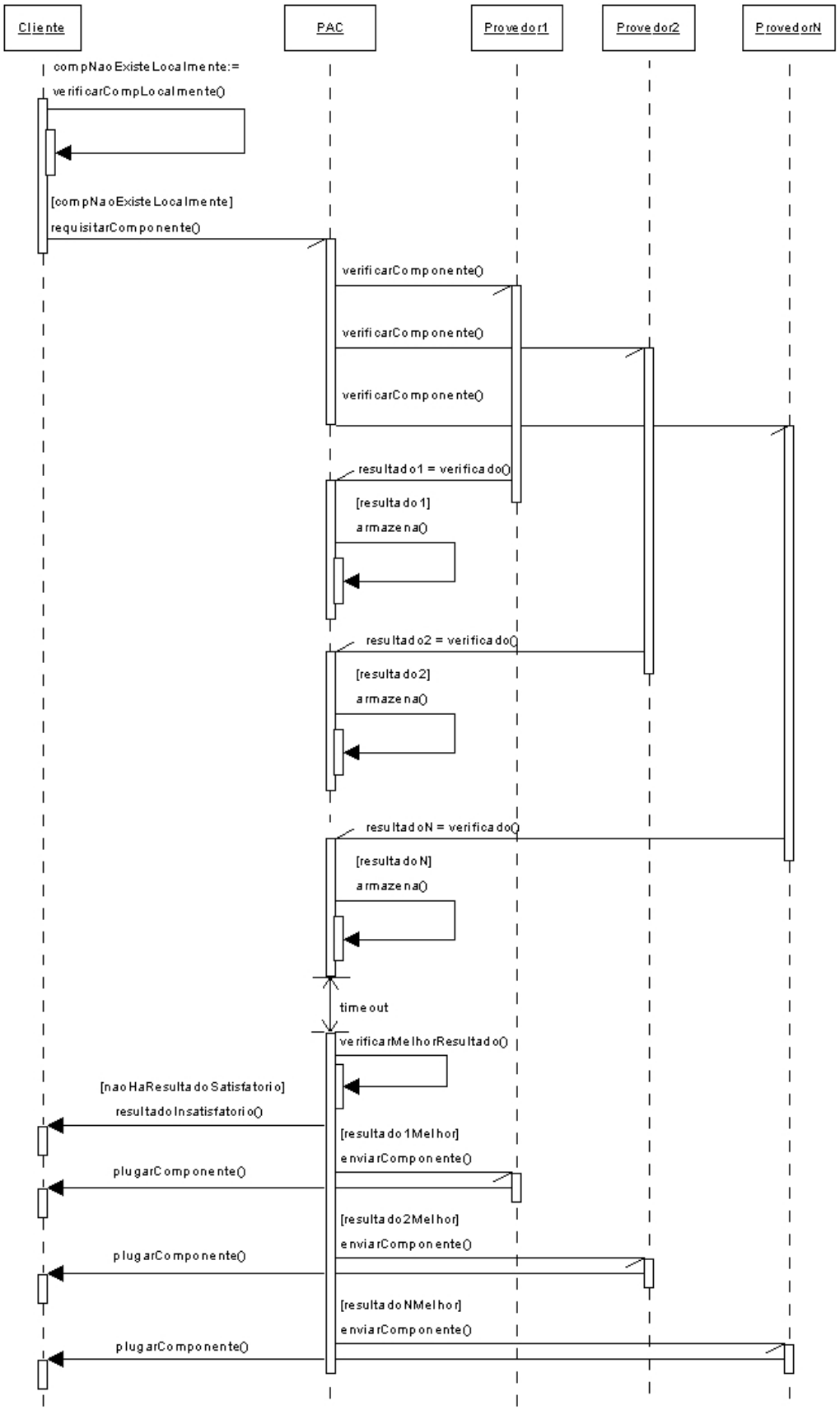

Figura 4.24 – Diagrama de Sequência – Plugagem Dinâmica

A Figura 4.24 detalha a seguinte sequência de operações (executada após a detecção de um *component fault*):

- Uma falta de componente (*component fault*) é detectada;
- O *Cliente* requisita um componente ao *PAC*;
- O *PAC* pesquisa em todos os *PAs* da federação pelo anúncio do componente;
- Se nenhuma resposta for obtida, o *PAC* irá notificar uma falha ao cliente;
- Caso contrário, todas as respostas recebidas serão armazenadas;
- Após um determinado período de tempo (*timeout*), o *PAC* finaliza o recebimento de novas respostas e analisa as já recebidas;
- Se não houver nenhuma resposta satisfatória, notificará isto ao cliente;
- Caso contrário, enviar ao *PA* que publicou o anúncio uma ordem para que este envie o componente ao cliente;
- O *PA* que recebeu a ordem de envio transporta o componente até o cliente.

Este é o fluxo geral de execução do protótipo. Internamente, porém, existem outras relações entre objetos menores. Cada entidade possui o seu próprio fluxo interno de execução.

A Figura 4.25 mostra o diagrama de sequência do Cliente FPA.

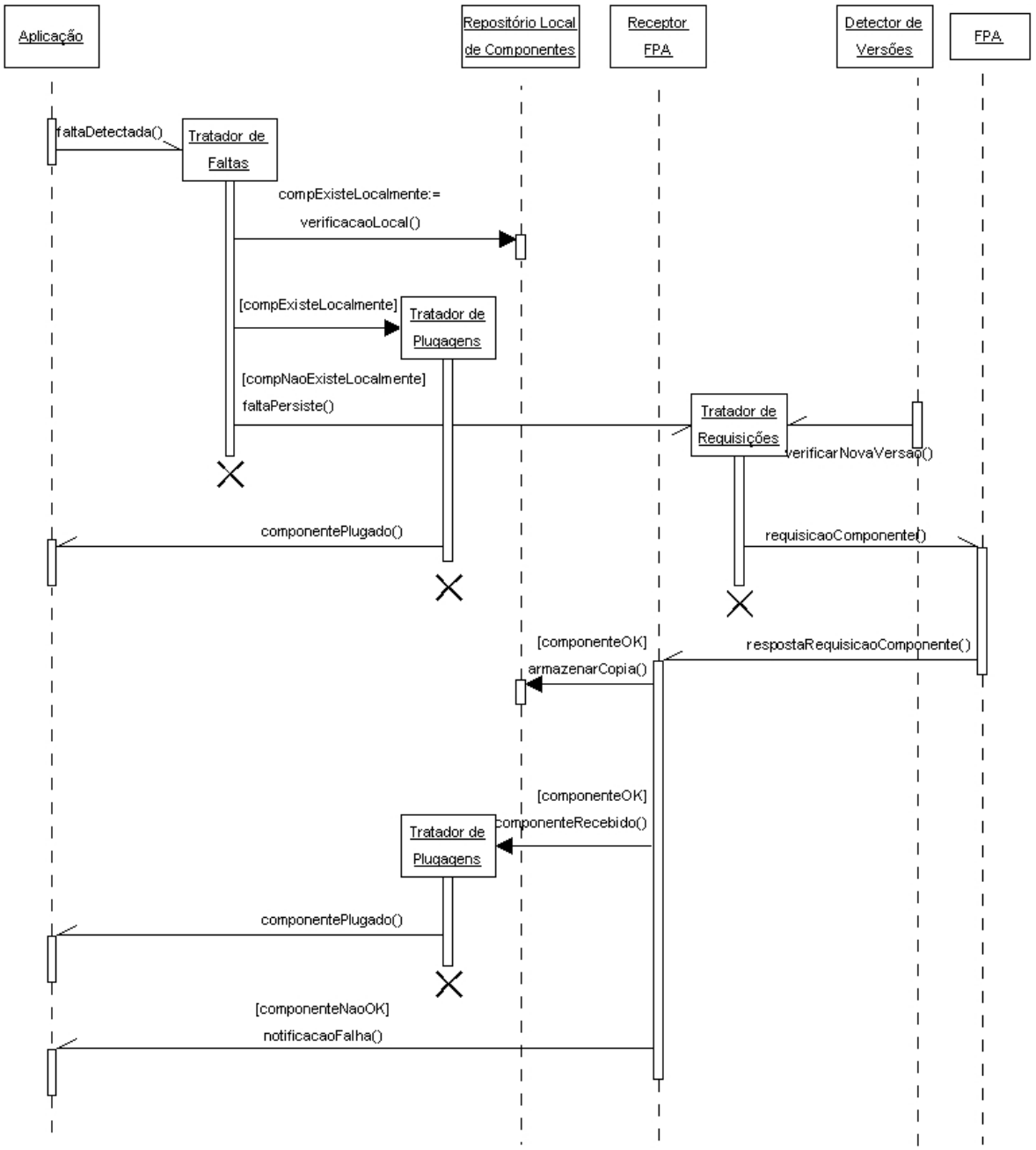

Figura 4.25 – Diagrama de Sequência – Cliente FPA

No Cliente FPA, o fluxo de execução é o seguinte (Figura 4.26):

- *Aplicação* tenta acessar uma funcionalidade inexistente localmente;
- É gerada uma falta de componente (*component fault*) e o cliente fica bloqueado;
- O *Tratador de Faltas* procura por uma cópia do componente no *Repositório Local de Componentes (RLC)*;

- Se a cópia local existir, o *Tratador de Plugagens* é chamado e esta é plugada ao cliente, desbloqueado-o e finalizando a transação;
- Caso contrário, o *Tratador de Requisições* é acionado;
- A especificação do componente, distribuída pelo desenvolvedor junto com a aplicação é então encapsulada em uma mensagem que será enviada à *FPA*;
- O *Tratador de Requisições* (*TR*) requisita o componente à *FPA*. Esta operação se dá da seguinte maneira:
  - O *TR* procura pelo anúncio do *pipe in* do PAC;
  - O *TR* cria um *pipe out* e o conecta ao *pipe in* do PAC;
  - Através deste canal de comunicação estabelecido, a requisição do componente faltante é enviada à *FPA*;
- A seguir, o *Receptor FPA* aguarda pela resposta da *FPA*;
- Se o componente não existir na *FPA*, o *Receptor FPA* notificará a Aplicação;
- Se o componente tiver sido transferido, o *Receptor FPA* irá encaminhar uma cópia para o *RLC*, e acionará o *Tratador de Plugagens*, que irá finalizar a transação.

Obs: O *Tratador de Requisições* também é acionado pelo *Detector de Versões*, a cada intervalo de tempo pré-determinado.

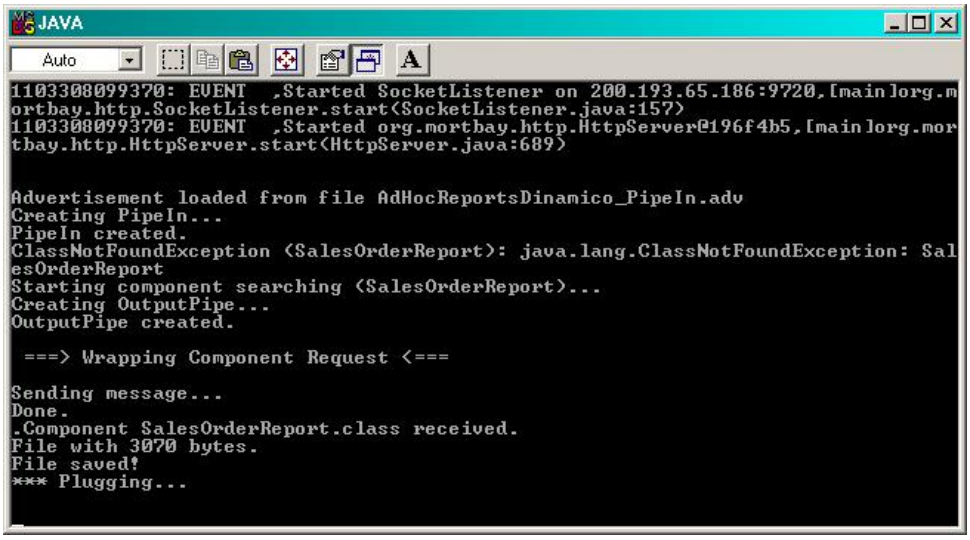

Figura 4.26 – Plugagem Dinâmica no Cliente FPA

A Figura 4.27 mostra o fluxo de execução do PAC.

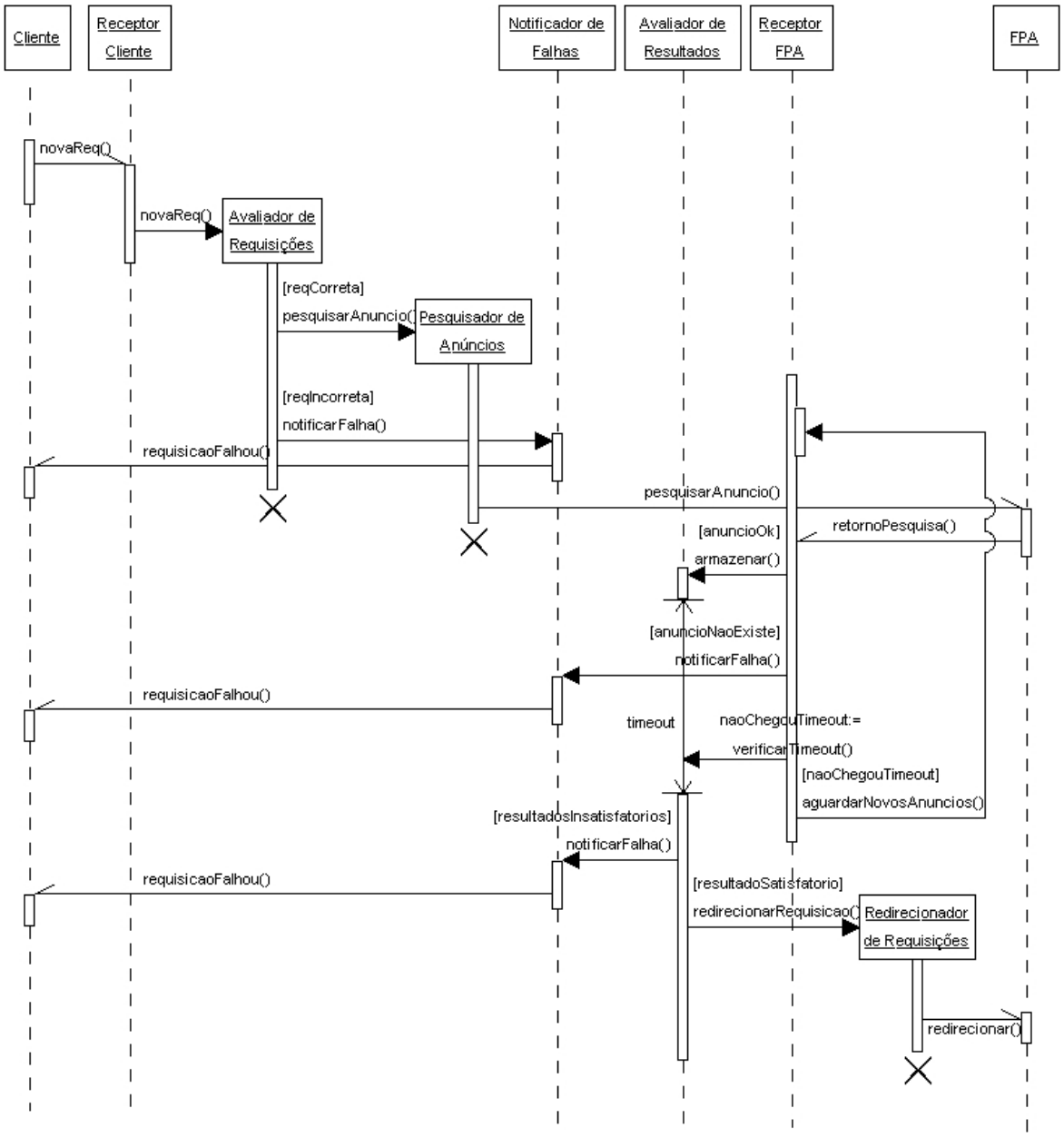

Figura 4.27 – Diagrama de Sequência – PAC

A sequência de operações é a seguinte (Figura 4.28):

- O *Receptor Cliente* recebe uma nova requisição;
- O *Avaliador de Requisições* é chamado para validar ou não a requisição;

- Se for verificado que a requisição é incorreta (isto é, não preenche o formato previamente estabelecido ou possui algum erro de transmissão), será acionado o *Notificador de Falhas*, encarregado de enviar uma mensagem de erro ao *Cliente*;
- Se for uma requisição válida, o *Pesquisador de Anúncios* será chamado para enviar uma mensagem *broadcast* a todos os PAs da federação averiguando sobre o anúncio do componente requisitado;
- Neste ponto entra em cena o *Avaliador de Resultados*, que irá armazenar cada resposta recebida, durante um determinado período de tempo (*timeout*);
- Se nenhum resultado chegar, o *Notificador de Falhas* será chamado e a transação será finalizada;
- Se após o *timeout*, nenhum resultado se mostrar satisfatório, o *Notificador de Falhas* também será chamado;
- Entretanto, se o *Avaliador de Resultados* considerar algum resultado como sendo satisfatório, ele acionará o *Redirecionador de Requisições*, que irá enviar a requisição do *Cliente* para o PA que publicou o anúncio escolhido;
- Neste ponto acaba a intervenção do PAC. Todo o resto do processo de composição dinâmica será feito diretamente pelo PA e pelo cliente;

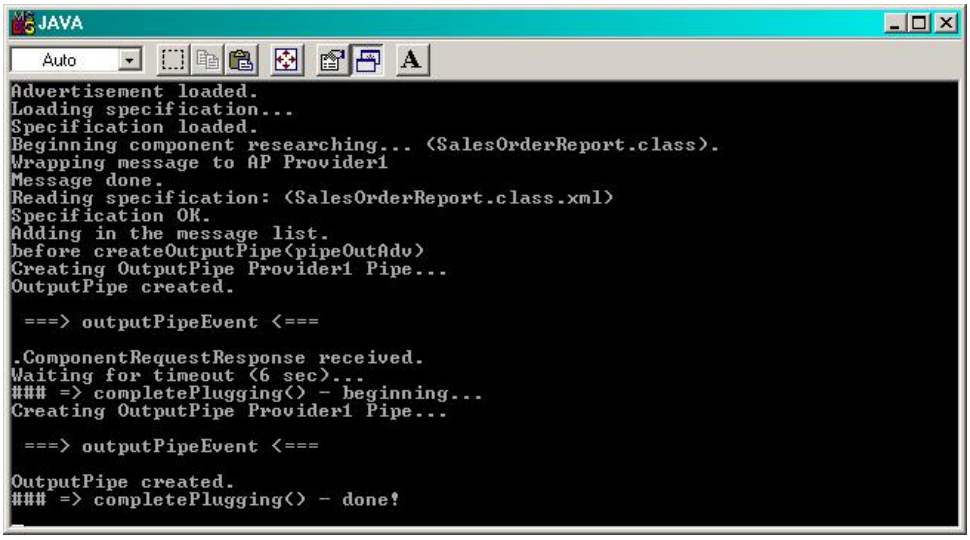

Figura 4.28 – Plugagem Dinâmica no PAC

Cada PA possui dois fluxos internos de execução, um que é executado durante a pesquisa do componente e outro durante a transferência do componente. Desta forma, quando recebe uma mensagem de pesquisa do PAC, as seguintes operações são feitas no PA (Figura 4.29):

- O *Receptor PAC* processa esta mensagem de pesquisa;
- O *Avaliador de Especificações* é chamado e efetua uma verificação pelo componente no *Repositório de Componentes*;
- Se o componente existir no *Repositório de Componentes*, o *Notificador de Sucesso* é acionado para enviar uma resposta positiva ao PAC;
- Caso contrário, nada é feito e a transação é finalizada pelo *Finalizador da Transação*.

Quando ocorre redirecionamento de requisição, a seguinte sequência de execução é disparada:

- *Receptor PAC* rebe o redirecionamento;
- O *Transportador de Componentes* é chamado. Ele irá criar um canal de comunicação de saída (*pipe out*) e irá conectá-lo ao canal de comunicação de entrada (*pipe in*) do cliente, cujo anúncio encontra-se na mensagem redirecionada. Através deste canal, o componente será transportado ao cliente;
- O papel do PA no processo se encerra após o transporte do componente (Figura 4.30).

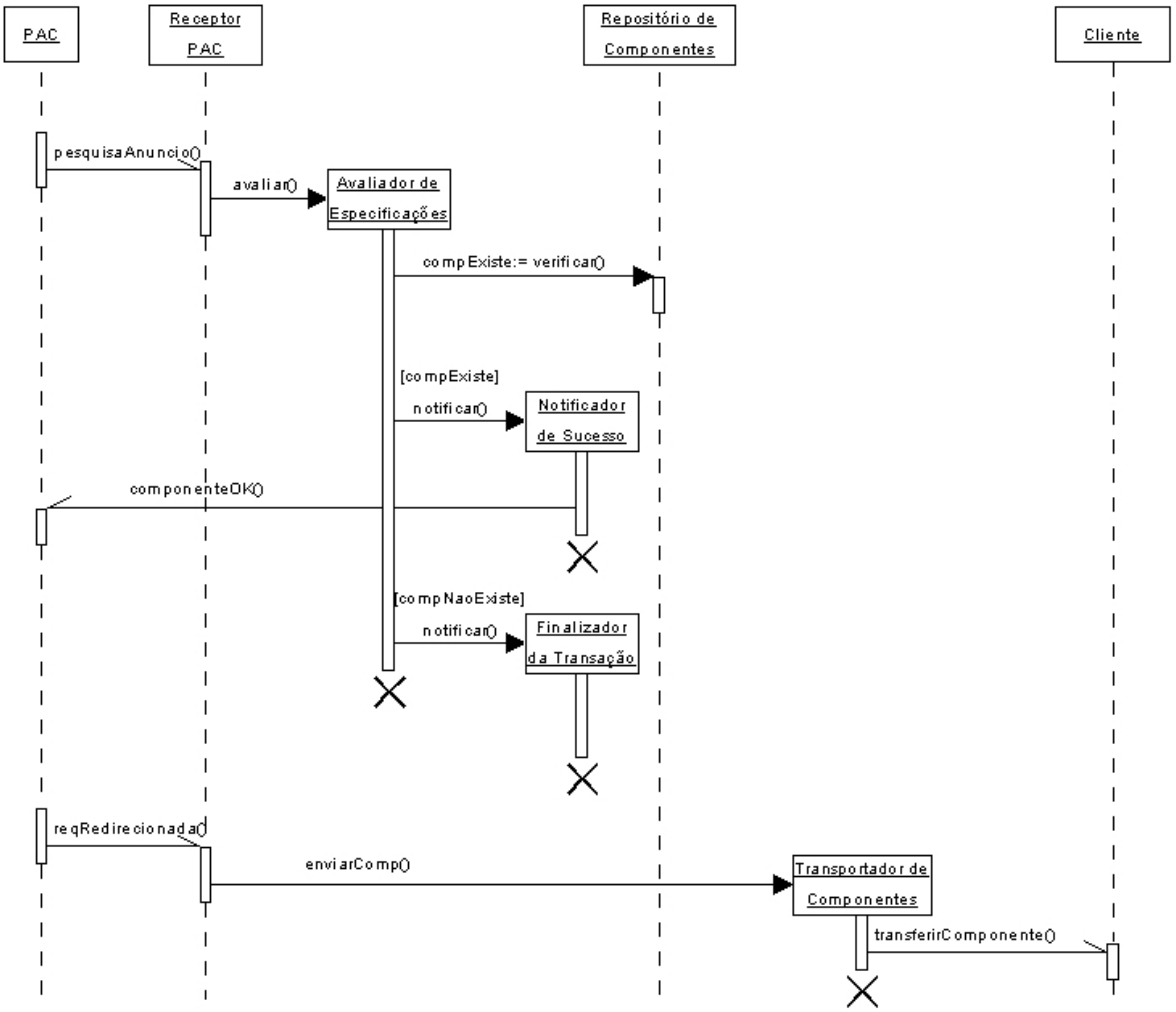

Figura 4.29 – Diagrama de Sequência – PA

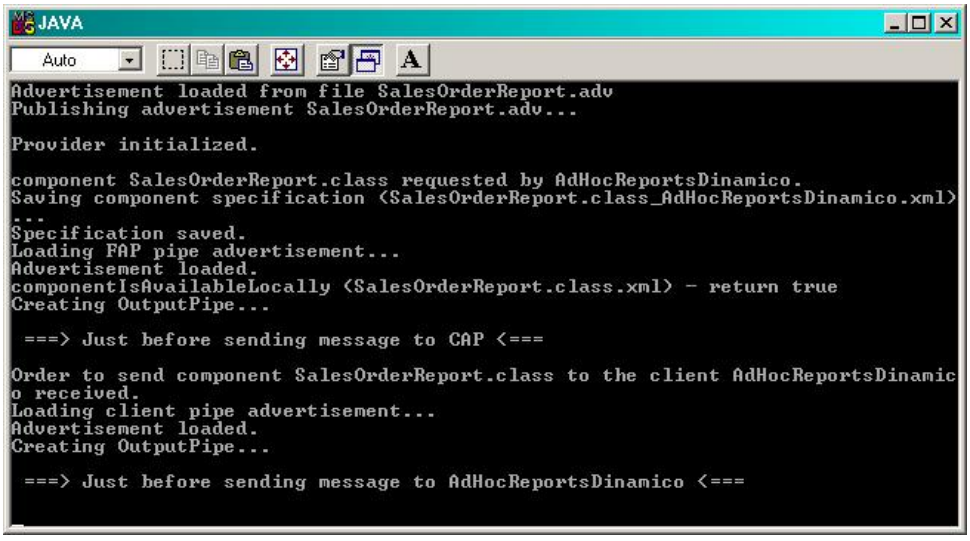

Figura 4.30 – Plugagem Dinâmica no PA

4.3 Avaliação dos Resultados

Segundo (Pressman, 2002), a medição qualitativa é fundamental para qualquer atividade de engenharia e a Engenharia de Software não é exceção porque se pode medir o processo de *software* com o objetivo de melhorá-lo de forma contínua. A medição também pode ser usada ao longo de um projeto para auxiliar na estimativa, no controle de qualidade, na avaliação de produtividade e no controle deste projeto. E, finalmente, pode ser usada pelos engenheiros de *software* para ajudar a avaliar a qualidade dos produtos do trabalho técnico e para auxiliar a tomada de decisões táticas, à medida que o projeto evolui.

Medidas diretas do processo de engenharia de *software* incluem custo e esforço aplicados. Medidas diretas do produto incluem linhas de código produzidas, velocidade de execução, tamanho de memória e defeitos relatados durante um certo período. Medidas indiretas incluem funcionalidade, qualidade, complexidade, eficiência, confiabilidade, manutenção, etc. O custo e o esforço necessários para construir um *software*, o número de linhas de código produzidas e outras medidas diretas são relativamente fáceis de se coletar, desde que convenções específicas para a medição sejam estabelecidas antecipadamente. Todavia, a qualidade e a funcionalidade do *software*, ou a sua eficiência, ou manutenção são mais difíceis de se avaliar e podem ser medidas apenas indiretamente (Pressman, 2002). No protótipo-exemplo implementado, apesar de ser uma aplicação pequena e bastante simples, utilizou-se *métricas orientadas a função* na tentiva de se avaliar o código construído.

Métricas Orientadas a Função

(Pressman, 2002) sugere a utilização de pontos por função. Segundo esta técnica, alguns parâmetros devem ser verificados e a estes deve ser associado um peso. A Tabela 2 mostra quais são estes parâmetros e as suas respectivas contagens no *AdHocReportsDinamico*.

A quantidade de entradas e saídas do usuário diz respeito à interação deste com o sistema, de maneira a fornecer ou visualizar dados do mesmo. As consultas são perguntas feitas em tempo de execução. O número de arquivos deve contabilizar todos os arquivos envolvidos no processo, como os códigos binários das classes, arquivos de base de dados

ou, especificamente para este caso, os arquivos relacionados com os componentes, com suas especificações e os arquivos de suporte da plataforma JXTA. São atribuídos fatores de peso, que dependem da complexidade do *software* em questão. Para este caso, considerou-se subjetivamente que o fator de peso simples era o mais adequado, pois a aplicação-protótipo em questão possui baixa complexidade.

Parâmetro de Medição	**Contagem**	**Fator de Peso Simples**	*Fator de Peso Médio*	*Fator de Peso Complexo*	**Resultado Multiplicação**
Quantidade de entradas do usuário	3	x 3	*x 4*	*x 6*	9
Quantidade de saídas do usuário	4	x 4	*x 5*	*x 7*	16
Quantidade de consultas do usuário	1	x 3	*x 4*	*x 6*	3
Número de arquivos	235	x 7	*x 10*	*x 15*	1645
Quantidade de interfaces externas	59	x 5	*x 7*	*x 10*	295
Resultado Total da Contagem (somatório das multiplicações)					1968

Tabela 2 – Métricas Orientadas a Função

Feito isto, (Pressman, 2002) define a seguinte relação para se contar pontos por função (*function points*, *FP*) em uma aplicação:

$$\mathbf{FP = Resultado\ Total\ da\ Contagem \times [0{,}65 + 0{,}01 \times \Sigma\ (Fi)]}$$

Os *Fi* (i = 1 a 14) são respostas baseadas nas perguntas abaixo. Cada resposta deve obedecer a uma escala de 0 a 5, onde 0 significa não-importante ou não-aplicável e 5 absolutamente essencial.

A seguir, serão mostradas as perguntas, conforme definidas por Arthur (1985), com suas respectivas respostas para o caso analisado:

1. O sistema requer salvamento (*backup*) e recuperação (*recovery*)? 0
2. Comunicações de dados são necessárias? 5
3. Há funções de processamentos distribuídos? 5
4. O desempenho é crítico? 1
5. O sistema vai ser executado em um ambiente operacional existente, intensamente utilizado? 0
6. O sistema requer entrada de dados *on-line*? 1
7. A entrada de dados *on-line* exige que a transação de entrada seja construída através de várias telas ou operações? 1
8. Os arquivos-mestre são atualizados *on-line*? 3
9. As entradas, saídas, arquivos ou consultas são complexas? 2
10. O processamento interno é complexo? 2
11. O código é projetado para ser reusado? 5
12. A conversão e a instalação estão incluídas no projeto? 2
13. O sistema está projetado para instalações múltiplas em diferentes organizações? 4
14. A aplicação está projetada para facilitar modificações e para a facilidade de uso pelo usuário? 4

Neste caso, FP é igual a 1968. O valor de FP é então usado de maneira análoga à contagem das linhas de código (*lines of code, LOC*). Sendo assim, pode-se definir alguma referência (um outro *software*, por exemplo) e se verificar a ocorrência de (Pressman, 2002):

- Erros por FP / LOC;
- Defeitos por FP / LOC;
- Dinheiro gasto por FP / LOC;
- Páginas de documentação por FP / LOC;

- FP / LOC por pessoa / mês.

Medição Qualitativa

Medir a eficiência de um *software* é uma tarefa um pouco mais complicada. Neste caso entram em cena parâmetros subjetivos de avaliação, tais como eficácia, facilidade de funcionamento, especialização, adaptabilidade, etc. Para este protótipo, definiu-se que os seguintes critérios deveriam ser levados em conta:

- Transparência: Foi estabelecido que toda a operação do protótipo decoresse com a mais absoluta transparência em relação ao usuário. O programa simplesmente disparou o processo de composição dinâmica enquanto permanecia em estado de espera (*stand-by*). A aplicação-cliente foi capaz de se comunicar com a FPA para receber o componente desejado, plugar o componente e liberar a sua funcionalidade ao usuário-final, sem que deste fosse exigida qualquer tipo de interação;

- Especialização de Tarefas: Cada um dos componentes realizou a sua tarefa de maneira bem específica. As consultas SQL que traziam embutidas definiram as tabelas da base de dados do SC^2 com as quais cada um iria se comunicar;

- Escalabilidade: As plugagens sucessivas não afetaram o desempenho dos componentes plugados anteriormente, nem a execução do *AdHocReportsDinamico* como um todo;

- Portabilidade: A plataformas de programação utilizadas (Java e JXTA) permitem que o código seja automaticamente portado para outro ambiente que implemente a Máquina Virtual Java;

- Viabilidade de Modelo de Negócios: Não foi possível testar a fundo diferentes tipos de modelo de negócios porque, como foi dito na Seção 4.2.2, a técnica de plugagem utilizada não permitiu o desacoplamento dos componentes dinâmicos;

- Viabilidade do ClienteAgrupado (Seção 3.2.3): O Repositório Local de Componentes (RLC) mostrou ser importante em sistemas que possuam muitos clientes, como por exemplo, funcionários de uma empresa conectados a uma rede

local. O processo de composição se torna muito mais rápido porque se elimina a comunicação com a FPA;

- Eficiência da Remoção de Defeitos: Por se tratar se uma aplicação simples, os testes e remoções de defeitos nunca foram muito trabalhosos. A ferramenta de desenvolvimento utilizada (JCreator, 2004) não possui tantas funcionalidades como outras equivalentes, mas foi suficiente para que se pudesse “debugar” o código;

- Tempo de Desenvolvimento: O protótipo foi desenvolvido ao longo de 2 anos e meio, em paralelo a este projeto de Mestrado. Durante este tempo, em 3 períodos ocorreram mudanças significativas, gerando versões diferentes das anteriores. Cada um destes períodos, entretanto, não levou mais do que 20 dias entre a modelagem e os testes. No meio termo entre estes períodos apenas se fez pequenos ajustes e testes visando o aprimoramento do que já estava funcionando, principalmente em relação às especificações dos componentes dinâmicos.

Pontos Fracos

Alguns problemas foram vivenciados com relação ao processamento do *AdHocReportsDinamico*. A maioria deles, porém, deveu-se ao ambiente de execução Java. A inicialização dos *peers* JXTA também não foi tão rápida quanto se esperava, e em várias ocasiões, conflitos gerados entre as configurações dos *peers* (especialmente em relação às portas de comunicação) obrigaram a que se reinicializasse toda a(s) máquina(s) que hospedava(m) o protótipo.

Um outro ponto a se ressaltar também foi o alto tempo de espera durante uma pesquisa ou descoberta JXTA, um problema típico de um sistema P2P que efetua busca por conteúdo. Mas apesar disso, pôde-se notar uma certa evolução nesta plataforma ao longo deste trabalho: as versões mais novas utilizadas se mostraram mais estáveis que as suas antecessoras.

Capítulo 5

5. Conclusões

No mundo moderno, percebe-se uma tendência à modularização de programas. Isso pode ser comprovado pela utilização de bibliotecas de *software* no desenvolvimento dos mais diversos tipos de programas, pela existência de tecnologias e ferramentas de integração de aplicações e pelo desenvolvimento baseado em componentes. Em um cenário de grande complexidade, poder se lidar com os problemas de forma localizada é sempre uma estratégia positiva. Apesar disso, obter-se um ambiente de implementação e utilização de *software* que disponibilize efetiva flexibilidade funcional não é tão simples.

O significativo avanço verificado nos últimos anos nas áreas de tecnologias de informação e infra-estruturas de comunicação e redes de computadores gerou grandes expectativas quanto à possibilidade concreta de se desenvolver tais ambientes, que consigam lidar com a crescente demanda por sistemas mais leves, ágeis, baratos, que possam ser executados em qualquer plataforma de *hardware* e de *software* e que estejam situados fisicamente em qualquer lugar. As Aplicações Compostas (*Composite Applications*) (Krass, 2004) são uma aposta neste sentido. O desenvolvimento de módulos de *software* dinâmicos e multi-plataforma (disponibilizados na forma de serviços ou componentes), que seguem especificações rigorosas, também tem sido visto como uma proeminente abordagem para maximizar a reusabilidade de código e tornar as aplicações mais flexíveis funcionalmente.

Ao se propor a Federação de Provedores de Aplicação (FPA) (Dutra e Rabelo, 2003) (Dutra e Rabelo, 2004), o que se pretendeu foi oferecer mais uma possibilidade de arquitetura, que fosse flexível, adaptável ao ambiente de implementação e utilização,

customizável, leve e escalável, e que utilizasse a composição de aplicações através de plugagem dinâmica de componentes. A FPA não pretendeu ser uma arquitetura substitutiva às já existentes mas, pelo contrário, sua concepção está assentada na idéia de integração com outras arquiteturas, tecnologias e modelos já existentes. O seu ponto de partida, inclusive, se deu com a análise de outras tecnologias, como por exemplo, os modelos-padrão de componentes, os Provedores de Serviços de Aplicação – PSAs (*Application Service Providers – ASPs*) (Dewire, 2002) – e as Aplicações Compostas.

Percebeu-se, então, a necessidade de se conceber uma arquitetura que:

- Possibilitasse, de fato, ao usuário "usar apenas o necessário, no momento necessário e no local necessário", ou seja, *flexibilidade funcional*.
- Oferecesse a flexibilidade e as garantias contratuais dos PSAs, porém com níveis mais variáveis de granularidade e não-dependente de um único servidor central;
- Possuísse a flexibilidade do uso de componentes, mas que não estivesse tão atrelada a estruturas de suporte, como os *containers* dos modelos tradicionais;
- Utilizasse a dinâmica das Aplicações Compostas, especialmente da Arquitetura Orientada a Serviços – AOS (*Service-Oriented Architecture – SOA*) (SOA, 2004a), mas que não fosse dependente de serviços (invocação remota de componentes);
- Possuíse uma lógica interna de busca por conteúdo, como a utilizada pela Federação de Serviços (Camarinha-Matos *et al.*, 2001), mas que diferentemente desta última, possuísse um grande nível de transparência de suas transações em relação aos seus usuários (humanos ou aplicações);
- Fosse passível de oferecer diferentes tipos de modelos de negócios, adaptados a diferentes tipos de clientes;
- Atuasse como um intermediário entre as empresas desenvolvedoras de *software* e o cliente final, que passaria a interagir especialmente com a FPA;
- Possibilitasse a imediata adaptação das aplicações-cliente aos seus respectivos ambientes de execução, ou seja, multi-plaforma;

- Reduzisse os custos da aquisição de novos módulos, aplicações ou de novas versões destes;
- Possuísse fácil instalação e utilização;
- Utilizasse grandes repositórios de componentes dinâmicos independentes, geograficamente distribuídos e expansíveis;

Mesmo com alguns pontos ainda em aberto – a maior parte decorrente do próprio aspecto emergente da abordagem e das tecnologias utilizadas – o protótipo-exemplo implementado para a área de gestão de empresas virtuais dá uma idéia concreta da potencialidade da arquitetura proposta. Através do protótipo pôde-se visualizar diversos aspectos que não haviam sido previstos no modelo inicial, como por exemplo, as interações entre os usuários e as aplicações-cliente da FPA. Detalhes como granularidade dos componentes utilizados e de projetos de aplicações previamente desenhadas para suportar a composição dinâmica somente se tornaram mais claros quando se instanciou um caso de uso. Este protótipo, desenvolvido paralelamente desde o início do trabalho, serviu para aperfeiçoar certos aspectos funcionais do modelo. Assim, com a sua constante evolução e aprimoramento ao longo do tempo, pôde-se sempre avançar na direção de se obter uma arquitetura que fosse a mais simples possível na sua concepção, porém prática de se usar e eficaz na obtenção de resultados.

A FPA mostrou ser uma arquitetura adaptável a diferentes configurações. Qualquer módulo de *software* que seja passível de ser composto dinamicamente e que possua especificadas as suas funcionalidades inexistentes pode ser considerado um Cliente FPA. Isto é válido para um componente que se plugue a outro componente, para uma aplicação especialmente desenhada ou para um sistema legado que incorporou um componente que suporte a composição dinâmica. Do ponto de vista conceitual é perfeitamente viável que um serviço seja um Cliente FPA, por exemplo; ou que um componente qualquer, plugado em um Cliente FPA ou não, também o seja.

Por causa desta flexibilidade, a FPA pode ser utilizada tanto por usuários domésticos como por usuários-empresa (sistemas industriais). As vantagens para a utilização doméstica estaria principalmente de se arcar com os custos exclusivos daquilo que se utilizou e durante o tempo em que esse uso foi feito. Uma outra vantagem, “herdada” dos PSAs, diz respeito aos usuários leigos em informática. Estes teriam a seu dispor uma

estrutura de suporte para ajudá-los na resolução de problemas relacionados a seus programas e também não precisariam mais enfrentar tantos problemas com relação à instalação ou à atualização de versões de suas aplicações dinamicamente compostas.

Para as aplicações industriais, além das vantagens citadas acima, pode-se destacar os seguintes fatores:

- Customização das Ferramentas: Seriam utilizadas apenas versões “enxutas” dos programas utilizados internamente. Toda a funcionalidade adicional seria dinamicamente plugada quando necessitada;
- Redução de Custos: Esse “enxugamento” das aplicações geraria uma boa economia financeira já que as licenças dos *software* seriam negociadas em outras condições e segundo um modelo de negócios previamente acordado com a FPA;
- Adaptação das Aplicações a Diferentes Ambientes de Utilização: Um funcionário que estivesse em viagem de negócios poderia facilmente acessar a sua área de trabalho onde quer que estivesse, bastando, para isso, que a sua empresa a tivesse disponibilizado em forma de componentes para serem executados em diferentes plataformas de *hardware* e *software*;
- Padronização de Dados: Esta característica, existente também na AOS, tornaria mais fácil a integração com sistemas de outras empresas, facilitando o tráfego de informações e serviços (cooperação) entre as mesmas.

A FPA não é recomendada, entretanto, para sistemas essencialmente monolíticos ou para aqueles que não necessitam se especializar. Esta arquitetura é melhor aproveitada se utilizada por aplicações que possuem um alto grau de expansão e retração, ou seja, uma escalabilidade dinâmica. Em outras palavras, que possuam grande possibilidade de vir a acoplar e desacoplar novas funcionalidades, ou ainda, para as quais a atualização constante de versões é imprescindível. A FPA é extremamente dependente da comunicação em rede (mensagens P2P, busca por conteúdo, envio de arquivos entre *peers*, publicação de anúncios, entre outros), portanto, também pouco aconselhada para sistemas de tempo-real, já que não se pode prever com precisão a latência de suas operações.

Um outro ponto a ser ressaltado é o fato de se poder visualizar cenários futuros onde surjam empresas cujo foco seja o desenvolvimento de componentes dinâmicos, e de outras,

especializadas exclusivamente em produzir aplicações que suportem a plugagem destes componentes. Talvez algumas empresas procurem se readaptar para englobar este novo tipo de desenvolvimento. Entretanto, é possível que num primeiro momento exista também alguma resistência a esta nova idéia, principalmente das maiores companhias, que podem não se interessar de imediato em vender *software* com menor valor agregado. Se se pensar no modelo de negócios, talvez uma solução para este impasse esteja na transformação ou associação destas grandes companhias em PAs, eliminando-se assim os intermediários.

De maneira geral, portanto, percebe-se uma certa tendência em se obter arquiteturas, modelos e tecnologias que flexibilizem o desenvolvimento e a utilização de *software*. Talvez a próxima década fique marcada como aquela em que os programas de computadores deixaram de ser vistos como produtos de prateleira e passaram a ser encarados como um serviço, ou seja, fáceis de se adquirir, de se usar, de se trocar por outro melhor e de se descartar. Para as empresas o ganho é evidente, com a redução de custos associada e a possibilidade de integração com sistemas de outras empresas. Para o usuário doméstico, aquele que há pouco mais de vinte anos precisava ser um *expert* em informática para utilizar um programa de computador, restará a verificação de que utilizar um *software* terá se tornado tão simples quanto alugar um filme em uma locadora ou assinar o *pay-per-view* na TV a cabo.

5.1 Trabalhos Futuros

O modelo proposto da Federação de Provedores de Aplicação associado à abordagem de Aplicações Compostas é bastante amplo e complexo. Devido a restrições temporais associadas naturalmente a uma dissertação de Mestrado, alguns pontos da proposta não puderam ser desenvolvidos na sua plenitude, ou mesmo não abordados.

O principal ponto em aberto deste trabalho diz respeito à especificação dos componentes dinâmicos. Desde o princípio este foi um item que exigiu um grande esforço no sentido de se buscar uma padronização em relação aos modelos já existentes. Logo ficou claro que o XML (Xml.Com, 2004) seria a tecnologia mais apropriada para especificar as interfaces. Primeiro pelo seu livre trânsito entre diferentes tecnologias, depois pela sua larga utilização pelos sistemas em rede que necessitam formatar e

transportar seus dados, e também por ser de fácil utilização e modelagem. Além disso, o XML também é a tecnologia base utilizada pela plataforma JXTA (JXTA, 2004a).

No momento seguinte verificou-se a necessidade de se ter duas especificações: uma para os dados não-funcionais (*metadados*) e outra para as interfaces propriamente ditas. Essa distinção foi feita de maneira a facilitar a sua definição pelos desenvolvedores na modelagem dos componentes. Para especificar os *metadados*, optou-se então pelo descritor de componentes CORBA (CCM) (OMG, 2002), que utiliza XML e é baseado no *Open Source Description* (OSD) (OSD, 1997), chamado *Software Package Descriptor*. Este descritor está definido em um arquivo público (*softpkg.dtd*), o que facilitou a sua utilização por parte da FPA. Ele define itens como nome do componente, desenvolvedor, sistema operacional para o qual o componente é destinado, linguagem de programação utilizada, linguagem humana do componente, etc.

Entretanto, esta facilidade não foi encontrada na hora de se especificar a interface do componente. Isso porque o CCM utiliza a IDL CORBA (OMG, 2004c) para especificar suas interfaces, o que, atrelaria fortemente a FPA ao modelo CCM. Isto, obviamente, ficaria fora do escopo concebido para esta arquitetura. Da mesma forma, as IDLs EJB (EJB, 2004) e Microsoft COM / DCOM (MIDL, 2004) não se mostraram satisfatórias. A empresa BEA (BEA, 2004) também vivenciou esta questão de padronização durante a concepção da AOS (Natis *et al.*, 2003). Segundo Natis *et al.* (2003), antes de optar pelo WSDL (WSDL, 2001), algumas outros padrões para definição de dados haviam sido tentados, como o CORBA *Interface Definition Language* (IDL) (OMG, 1992), a COM/DCOM Microsoft IDL (MIDL) (MIDL, 2004) e o *Customer Information Control System* (CICS) (CICS, 2001).

Porém, neste caso, a WSDL é de fato interessante por se tratar de uma linguagem de definição de serviços. O mesmo não se aplica à FPA, pois não existe um padrão para especificar componentes dinâmicos (*metadados* e interface). Portanto, um próximo trabalho que tenha como um dos objetivos se aprofundar neste item deverá tentar definir um padrão que seja integrável com as principais tecnologias de componentes e de serviços.

Outro ponto que precisa ser mais explorado é a questão da execução adaptável. Segundo esta idéia, um Cliente FPA será capaz de conectar-se à FPA de qualquer ponto geográfico que se encontre e independente do tipo de dispositivo que esteja usando. Assim,

desde que possua aplicações desenhadas para suportar a composição dinâmica, esse cliente será capaz de acessar dados e funcionalidades onde quer que esteja. A grande dificuldade desta abordagem, porém, se encontra na instalação da Plataforma de Suporte à Plugagem. Será necessário que se defina uma plataforma mínima, que possa ser rapidamente implantada em qualquer dispositivo virgem, seja ele um *laptop*, um quiosque Internet, um telefone celular, etc. Novos modelos de negócios também deverão emergir deste tipo de utilização, haja vista que deverão cobrir o contrato de utilização destes clientes temporários.

Trabalhos futuros deverão prestar bastante atenção à questão da segurança. Existem várias questões envolvidas em uma arquitetura que se pretende fortemente decentralizada e onde os dados irão trafegar livremente entre as partes. Seria muito importante investir no desenvolvimento do suporte à criptografia e à autenticação dos dados, por exemplo. A idéia por trás disto é que pacotes indesejados de dados possam ser descartados ou ignorados, evitando assim a difusão de vírus, componentes não-certificados, informações falsas em geral e ataques externos do tipo “cavalo de tróia”. A plataforma JXTA, inclusive, já possui protocolos prontos para serem utilizados por quem pretende dotar seus sistemas de tais garantias.

A Arquitetura FPA “herda” diversos problemas existentes em sistemas P2P e em sistemas em rede, no geral (Seção 2.3). É necessário, pois, se tomar as devidas precauções para que a utilização da mesma não seja comprometida, principalmente quando se pretende que seja utilizada em larga escala.

Além disso, há pressupostos importantes que são assumidos e que tem fortes implicações no projeto de sistemas. Talvez a grande contribuição que a FPA ofereça aos desenvolvedores de *software* seja a perspectiva de se desenvolver uma nova metodologia de implementação de *software*. Será necessário se repensar a maneira com a qual são feitos os projetos de aplicações. Para que estas sejam compatíveis com o ambiente de composição dinâmica que se pretende criar, é preciso que se reveja os limites do que hoje é considerado como funcionalidade básica. Os analistas de sistemas e programadores em geral precisam ter bem claro a idéia de que haverá um bloco básico com poucas funcionalidades (*kernel*) e composto por *plugs*, nos quais serão agregados os componentes dinâmicos. Um cuidado todo especial com a especificação dos componentes deverá ser tomado pois, após a fase de desenvolvimento, estas mesmas serão distribuídas para os

Provedores de Aplicação que forem armazenar os componentes por elas especificados, assim como também para seus eventuais clientes, pois estas especificações estarão agregadas ao *kernel* de cada aplicação recém adquirida por eles.

É necessário também se pensar em um modelo de integração entre aplicações passíveis de serem compostas dinamicamente e outros tipos de aplicações. Isso sem falar dos sistemas já instalados, ou legados (*legacy systems*), que não foram projetados para suportar a composição dinâmica mas que pretendam utilizá-la. Provavelmente seja necessário se desenvolver algum módulo de adaptação, talvez uma pequena aplicação ou até mesmo um componente que possua os *plugs* necessários para a composição dinâmica. Isto acarretaria em uma arquitetura híbrida, onde aplicações “não enxutas” poderiam ter suas funcionalidades expandidas através de componentes dinâmicos.

É importante ressaltar também novas comparações com outros modelos e propostas. Existem tecnologias recentes com grande potencial de expansão, como por exemplo os Grids (Grid, 2004), que oferecem uma boa perspectiva de interação com a FPA.

6. Referências Bibliográficas

ACE-GIS (2004), *Adaptable and Composable E-commerce and Geographic Information Services* – http://www.acegis.net/, em maio de 2004.

Afsarmanesh, H., Camarinha-Matos, L. M. (2000), *Future Smart-Organizations: A Virtual Tourism Enterprise. In proceedings of the First International Conference on Web Information Systems Engineering (WISE'00)*. Hong Kong, China, 19 a 21 de junho de 2000, págs. 456-461.

AGEDIS (2003), *Automated Generation and Execution of Tests Suites for Distributed Component-based Software* – http://www.agedis.de/, em março de 2004.

Arthur, L. J. (1985), *Measuring Programmer Productivity and Software Quality, Wiley-Interscience, 1985*.

ASP (2004), *Application Service Provider Agreements*, http://www.techagreements.com/T-Application-Service-Provider-Agreements.asp, em dezembro de 2004.

Atkinson, C., Bayer, J., Laitenberger, O., Zettel, J., (2000), *Component-Based Software Engineering: The KobrA Approach - International Workshop on Component-Based Software Engineering, held in conjunction with the 22nd International Conference on Software Engineering (ICSE2000) Limerick, Ireland, June 5-6, 2000.*

Balasubramanian, K. (2002), *Composition In The CORBA Component Model – Department of Computer Science, Washington University in St.Louis, St. Louis, USA, 2002.*

BankSEC (2002), *Secure Banking Application Assembly using a component based approach* – http://www.atc.gr/banksec, em agosto de 2002.

Bartlett, D. (2001), *CORBA Component Model (CCM) – Introducing next-generation CORBA – IBM*, http://www-128.ibm.com/developerworks/webservices/library/co-cjct6/, em dezembro de 2004.

BEA (2004), *SOA Resource Center* – http://www.bea.com, em outubro de 2004.

Beatty, J. (2004), *Introducing SDO – BEA Systems* – http://dev2dev.bea.com/technologies/soa/articles/sdo_beatty.jsp, em outubro de 2004.

Beneken, G., Hammerschall, U., Broy, M., Cengarle, M. V., Jürjens, J., Rumpe, B., Schoenmakers, M. (2003), *Componentware – State of the Art 2003. In Understanding Components Workshop of the CUE Initiative at the Univerità Ca' Foscari di Venezia, Venice – Italy, October 7th-9th 2003*.

BitTorrent (2004), *BitTorrent: Protocol Specification* – http://bittorrent.com/protocol.html, em novembro de 2004.

Booth, M. (2002), *The ABC of ASPs, 28/01/2002* – http://b2b.ebizq.net/asp/booth_1.html, em agosto de 2002.

Borland (2004), Borland Interbase – Cross-platform embedded database – http://www.borland.com/interbase/, em novembro de 2004.

Brain, M. (2002), *How ASPs Work* – http://www.howstuffworks.com/asp.htm/printable – em agosto de 2002.

Calim (2001) – *Corba Architecture for Legacy Integration and Migration* – http://dbs.cordis.lu/fep-cgi/srchidadb?ACTION=D&SESSION=92122003-8-7&DOC=30&TBL=EN_PROJ&RCN=EP_RCN_A:53593&CALLER=PROJ_IST, em março de 2004.

Camarinha-Matos, L. M., Afsarmanesh, H., Kaletas, E., Cardoso, T. F. (2001), *Service Federation in Virtual Organizations, in Proceedings of IFIP TC5 / WG5.2 & WG5.3 Eleventh Int. PROLAMAT Conf. On Digital Enterprise – New Challenges, Kluwer Academic Publishers, pp 305-324, Hungary, 2001*.

CARTS (2002), *Computer Aided Architectural Analysis of Real Time Systems* – http://www.tcpsi.es/carts/, em agosto de 2002.

Chastain, S. (2004), *Graphics Software – Plug-ins, Filters, and Extensions* – http://graphicssoft.about.com/od/pluginsfilterseffects/a/aboutplugins.htm, em dezembro de 2004.

Cheesman, J., Daniels, J. (2001), *UML Components: A Simple Process for Specifying Component-Based Software*, Addison Wesley, 2001.

CICS (2001) – *Customer Information Control System (CICS) – WhatIs.com* – http://search390.techtarget.com/tip/1,289483,sid10_gci561264,00.html, em novembro de 2004.

ClienteServidor (2004) – *Client / Server – A WhatIs.com Definition* – http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci211796,00.html, em novembro de 2004.

Coach (2004) – *Component Based Open Source Architecture for Distributed Telecom Applications* – http://dbs.cordis.lu/fep-

cgi/srchidadb?ACTION=D&SESSION=92122003-8-7&DOC=39&TBL=EN_PROJ&RCN=EP_RCN_A:61829&CALLER=PROJ_IST, em março de 2004.

COM (2004), *COM: Component Object Model Technologies – Microsoft* – http://www.microsoft.com/com/default.mspx, em dezembro de 2004.

Combine (2002), *Combine Project* – http://www.opengroup.org/combine/overview.htm, em março de 2004.

COTS (2004) – *COTS-Based Systems (CBS) Initiative* – http://www.sei.cmu.edu/cbs/, em maio de 2004.

Cylex (2001), CyLex i-DOX – http://www.dmi-inc.com/products/cylex/cylex_idox.html, em dezembro de 2004.

DAS (2004), Departamento de Automação e Sistemas – Webmail – https://webmail.das.ufsc.br, em dezembro de 2004.

Dewire, D. T. (2002), *Application Service Providers - Enterprise Systems Integration, 2nd Edition, pag.449-457. Auerbach Publications, 2002.*

Dietzen, S. (2004), *Standards for Service-Oriented Architecture – BEA Systems* – http://dev2dev.bea.com/technologies/soa/articles/soa_dietzen.jsp, em outubro de 2004.

D'Souza, D. F., Wills, A. C. (1999), *Objects, Components and Frameworks with UML – The Catalysis Approach, Addison-Wesley Publishing Company, 1999.*

Dutra, M. L., Rabelo, R. J. (2003), Instanciação Funcional e Dinâmica de Sistemas Industriais, Anais do VI Simpósio Brasileiro de Automação Inteligente (SBAI), Bauru (SP), Setembro de 2003.

Dutra, M. L., Rabelo, R. J. (2004), *Intelligent and Dynamic Plugging of Components - An Example for Networked Enterprises, in Proceedings of IFIP TC 5 / WG 5.5 Sixth IFIP International Conference on Information Technology for Balanced Automation Systems in Manufacturing and Services (BASYS) - Emerging Solutions for Future Manufacturing Systems, pg. 219-230, Vienna, Austria, 27-29 September 2004.*

E-Business Strategies (2004), *Composite applications – Frequently Asked Questions* – http://www.ebstrategy.com/services/composite/composite_apps_faq.htm, em abril de 2004.

ECC (2001), *Efficient Collaboration in the Prefabricated Components Construction Network* – http://dbs.cordis.lu/fep-cgi/srchidadb?ACTION=D&SESSION=253052004-12-13&DOC=21&TBL=EN_PROJ&RCN=EP_RPG:IST-1999-55013&CALLER=PROJ_IST, em dezembro de 2004.

EJB (2004), *J2EE – Enterprise JavaBeans Technology* – http://java.sun.com/products/ejb/, em dezembro de 2004.

EJBFactory (2003), Java2 Enterprise Edition (J2EE) – http://www.ejbfactory.com/welcome/documentation/java/j2ee/j2ee.htm, dezembro de 2004.

EJBTutorial (2004), *EJB Tutorial – An Overview of EJBs* – http://www.ejbtut.com/Overview.jsp, em dezembro de 2004.

Fabiunke, M. (2001), *Component-based Product Line Engineering - The KobrA Project (Position Paper) - Proceedings of the Dagstuhl-Seminar on Product Family Development, Wadern, Germany, 12-20 April 2001*.

FETISH (2003), *Federated European Tourism Infrastructure System Harmonization – FETISH* – http://www.fetish.t-6.it/, em abril de 2003.

Firebird (2004), Firebird – Relational Database for the New Millennium – http://www.firebirdsql.org/, em dezembro de 2004.

Fraga, J., Siqueira, F., Favarim, F. (2003), *An Adaptive Fault-Tolerant Component Model, IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003), January 2003, Guadalajara, Mexico*.

Fowler, M., Scott, K. (2000), UML Essencial – Um breve guia para a linguagem-padrão de modelagem de objetos, Bookman, 2000.

Garita, C., Kaletas, E. C., Afsarmanesh, H., Hertzberger, L. O. (2002), *A Service Interface Definitions Catalogue for Virtual Enterprises in Tourism, 5th IEEE/IFIP International Conference on Information Technology for Balance Automation Systems in Production and Transportation – BASYS '02, Cancun, Mexico*.

Garfinkel, S. (2004), *Peer-to-Peer Comes Clean – They're not just for file-sharing anymore: P2P networks are transmitting phone calls, blocking spam, backing up hard drives, and spreading scholarship*. – http://www.technologyreview.com/articles/04/10/wo_garfinkel100704.asp?p=0, em novembro de 2004.

Gimenes, I. M. S., Barroca, L., Huzita, E. H. M., Carniello, A. (2000), O Processo de Desenvolvimento Baseado em Componentes Através de Exemplos, VIII Escola de Informatica da SBC Sul, Raul Ceretta Nunes (ed), Editora da UFRGS, 147-178, 2000.

Giotto (1999), *Giotto ASP White Paper – How ASPs Deliver Value: Next Generation Portals for Business Applications, May 3, 1999* – http://www.trginternational.com/HTML/giotto.htm, em agosto de 2002.

Golden Orb (2003), *Questions & Answers – What are Composite Applications?* – Golden Orb Technologies Australia Ltd, http://websites.golden-orb.com/GoldOrb/100202.php, em abril de 2004.

Gong, L. (2002), *Project JXTA: A Technology Overview – Sun Microsystems, Inc., October 29, 2002.*

Grid (2004), *Grid Computing Projects – Grid.org* – http://www.grid.org/home.htm, em novembro de 2004.

Han, J. (1999), *An Approach to Software Component Specification - Papers: 1999 International Workshop on Component-Based Software Engineering ICSE 1999: May 16-22, 1999 Los Angeles, CA, USA.*

Havenstein, H. (2003), *Startup to create composite applications, InfoWorld* – http://www.infoworld.com/article/03/06/23/HNcomposite_1.html, em abril de 2004.

HARP (2000), *HArmonization for the secuRity of the web technologies and aPplications* – – http://www.telecom.ntua.gr/~HARP/HARP/HARP.htm, em dezembro de 2004.

Hocker, R. (2004a), *Why SOA Now: A Q&A With BEA CIO Rhonda Hocker – BEA Systems* – http://www.bea.com/framework.jsp?CNT=fea00018.htm&FP=/content/news_events/features_news/features, em outubro de 2004.

Hocker, R. (2004b), *A Roadmap for SOA: A Q&A With BEA CIO Rhonda Hocker – BEA Systems* – http://www.bea.com/framework.jsp?CNT=fea00020.htm&FP=/content/news_events/features_news/features, em outubro de 2004.

Hocker, R. (2004c), *Making the Case for SOA: A Q&A With BEA CIO Rhonda Hocker – BEA Systems* – http://www.bea.com/framework.jsp?CNT=fea00021.htm&FP=/content/news_events/features_news/features, em outubro de 2004.

Horstmann, M., Kirtland, M. (1997), *DCOM Architecture – MSDN Library, July 23, 1997* – http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomarch.asp, em dezembro de 2004.

Hritz, D. (2004), *Business Processes Made Simple with SOA – WebLogicPro* – http://www.ftpon-line.com/weblogicpro/2004_05/magazine/features/dhritz/, em outubro de 2004.

Hurley, O. (2004), *Web Services or Peer-to-Peer? – Web Services Journal* – http://www.sys-con.com/webservices/article.cfm?id=57, em novembro de 2004.

HyperDictionary (2003), *Meaning of Granularity* – http://www.hyperdictionary.com/dictionary/granularity, em dezembro de 2004.

IBM (2004a), *WebSphere Software* – http://www-306.ibm.com/*software*/websphere/, em dezembro de 2004.

IBM (2004b), *WebSphere Technology – What is WebSphere?* – http://www-306.ibm.com/*software*/info1/websphere/wstechnology.jsp?S_TACT=103BGW01&S_CMP=campaign, em dezembro de 2004.

IBM (2004c), WebSphere Software – WebSphere Studio – http://www-306.ibm.com/*software*/info1/websphere/index.jsp?tab=products/studio&S_TACT=103BGW01&S_CMP=campaign, em dezembro de 2004.

iCMG (2004), Interworking between EJBs and CORBA Components (CCM) – Developer Resource, iCMG – http://www.icmgworld.com/corp/Developer/dev.ejb_ccm.asp, em dezembro de 2004.

Java (2004), Java 2 Platform Standard Edition (J2SE) – Java Technology – http://java.sun.com/j2se/, em dezembro de 2004.

J2EE (2004), *Java 2 Platform, Enterprise Edition (J2EE) – Java Technology* - http://java.sun.com/j2ee/, em agosto de 2004.

JavaRMI (2004), *Java Remote Method Invocation (Java RMI)* – http://java.sun.com/products/jdk/rmi/, em agosto de 2004.

JavaPlugIn (2004), The JavaTM Plug-in Component – http://java.sun.com/j2se/1.4.2/docs/guide/plugin/, em dezembro de 2004.

JCreator (2004), Jcreator – Xinox Software – http://www.jcreator.com/, em dezembro de 2004.

Jini (2004), *Jini Network Technology* – http://wwws.sun.com/*software*/jini/, em maio de 2004.

Juric, M. B. (2002), *Integrate EJBs with CORBA – Access EJBs from non-Java-based applications, JavaWorld March 2002* – http://www.javaworld.com/javaworld/jw-03-2002/jw-0329-corba.html?, em dezembro de 2004.

JXTA (2004a), *JXTA Project* – http://www.jxta.org/, em novembro de 2004.

JXTA (2004b), *JXTA Technology: Creating Connected Communities – JXTA Technology Documents, January 2004.*

JXTA (2004c), *Project JXTA Overview Scenarios* – http://platform.jxta.org/TutorialGraphics.html, em dezembro de 2004.

JXTA (2004d), *JXTA Solutions Catalog* – http://www.jxta.org/Catalog/index-catalog.html, em dezembro de 2004.

JXTA (2004e), *Company Spotlight Archive* – http://www.jxta.org/project/www/companies/companyarchive.html, em dezembro de 2004.

Kazaa (2004), *Kazaa* – http://www.kazaa.com/us/index.htm, em novembro de 2004.

Krass, P. (2004), *Composite applications – Easy-to-build software that works the way you do. – CFO.com* – http://www.cfo.com/printarticle/0,5317,11199/M,00.html?f=options, em abril de 2004.

KWeb (2004), *KNOWLEDGE WEB – Realizing the semantic web* – http://dbs.cordis.lu/fep-cgi/srchidadb?ACTION=D&SESSION=244802004-12-13&DOC=175&TBL=EN_PROJ&RCN=EP_RPG:507482&CALLER=PROJ_IST, em dezembro de 2004.

LaMonica (2003), *SeeBeyond looks for an edge – SeeBeyond, hoping to find an edge in the competitive integration software market, has retooled its products with Web services in mind – CNET News* – http://news.zdnet.com/2100-3513_22-992017.html, em dezembro de 2004.

Lauder, A. (1999), *Pluggable factory in Pratice, C++ Report Magazine, pp. 27-32, v. 11, n. 9, Oct 1999*.

Liang, S., Bracha, G. (1998), *Dynamic Class Loading in the Java Virtual Machine, Proceedings of OOPSLA'98, Vancouver, Canada, October, 1998*.

LionShare (2004a), *LionShare – Project Information*, http://lionshare.its.psu.edu/main/, em novembro de 2004.

LionShare (2004b), *LionShare – Project Description*, http://lionshare.its.psu.edu/main/info/descript, em novembro de 2004.

Liu, Y., Cunningham, H. C. (2002), *Software component specification using design by contract - Proceedings of the SoutEast Software Engineering Conference, Tennessee Valley Chapter, National Defense Industry Association, Huntsville, AL, April 2002.*

Liu Y., Cunningham, H. C. (2004), *Mapping component specifications to Enterprise JavaBeans implementations - Proceedings of the ACM Southeast Conference, pp. 177-182, April 2004.*.

LIVE@WEB.COM (2000) – *LIVE@WEB.COM – Context-based Video Retrieval on the Web* – http://a7www.igd.fhg.de/projects/liveweb/liveweb.html, em dezembro de 2004.

Macdonald, M. (2002), *ASP.NET: The Complete Reference – McGraw-Hill, Berkley, California, USA, 2002*.

McIlroy, D. (1968), *Mass-produced software components. In Software Engineering, NATO Science Committe report, págs. 138-155*.

MECASP (2001), *Maintenance and Improvement of component-based applications diffused in ASP mode* – http://dbs.cordis.lu/fep-cgi/srchidadb?ACTION=D&SESSION=253052004-12-13&DOC=27&TBL=EN_PROJ&RCN=EP_RPG:IST-2000-30165&CALLER=PROJ_IST, em dezembro de 2004.

Meyer, B. (1992), *Applying Design by Contract, Computer, IEEE, Ocotober 1992*, pags. 40 - 51.

Meyer, B. (2003), *Design by Contract - A Conversation with Bertrand Meyer, Part II by Bill Venners, December 8, 2003*, http://www.artima.com/intv/contracts.html, em janeiro de 2004.

Microsoft (2002), *COM+* – http://www.microsoft.com/com/tech/COMPlus.asp, em maio de 2004.

Microsoft (2004), Windows XP Home Page – http://www.microsoft.com/windowsxp/default.mspx, em dezembro de 2004.

MIDL (2004), *Microsoft Interface Definition Language (MIDL) – Microsoft Platform SDK* - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/midl/midl/midl_start_page.asp, em novembro de 2004.

Milojicic, D., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., Xu, Z. (2002), *Peer-to-Peer Computing , Technical Report HPL-2002-57, HP Labs. 2002.*

Minar, N., Hedlund, M. (2001), *A Network of Peers – Peer-to-Peer Models Through the History of the Internet – Peer-to-Peer: Harnessing the Power of Disruptive Technologies, Chapter 1, Edited by Andy Oram, March 2001.*

MMB (2004), *Magic Mirror Backup – Pensamos Digital* – http://www.pensamos.com/mmb/, em novembro de 2004.

MobileInfo (2001), Current *Topics in Mobiling Computing – IBM WebSphere SoftWare Platform* – http://www.mobileinfo.com/Current_Topics/websphere.htm, em dezembro de 2004.

MOM (1997), *Message-Oriented Middleware – Carneggie Mellon – Software Engineering Institute* – http://www.sei.cmu.edu/str/descriptions/momt_body.html, em outubro de 2004.

Morpheus (2004), *Morpheus: peer-to-peer file sharing software* – http://www.morpheus.com/, em novembro de 2004.

Mullender, M., Burner, M. (2002), *Services – Application Architecture: Conceptual View – Microsoft Corporation* – http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnea/html/eaappconservices.aspl, em dezembro de 2004.

Napster (2004), *Napster* – http://www.napster.com/, em novembro de 2004.

Natis, Y., Schulte, R. (2003), *Introduction to Service-Oriented Architecture – Gartner Inc.* – http://mediaproducts.gartner.com/reprints/bea_systems/114295.html, em outubro de 2004.

Nelson, M. (1998), *Using Distributed COM with Firewalls – MSDN Library* – http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomfirewall.asp, em dezembro de 2004.

NETDev (2004), Microsoft .NET Development – http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/netdevanchor.asp, em dezembro de 2004.

NETRemoting (2004), *.NET Remoting Overview – .NET Framework Developer's Guide* – http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconnetremotingoverview.asp, em agosto de 2004.

NETWeb (2003), What Are Web Services? – Microsoft .NET – http://www.microsoft.com/net/basics/webservices.asp, em dezembro de 2004.

Network (2000), A Chave que leva ao ASP, Revista Network Computing Brasil, pp. 26-34, n. 16, junho 2000.

OMG (1992), *OMG CORBA IDL* – http://www.objs.com/x3h7/corbaidl.htm, em abril de 2004.

OMG (2002), *Object Management Group – CORBA Componentes – formal/02-06-65.2002*.

OMG (2004a), *OMG's CORBA Website* – http://www.corba.org/, em maio de 2004.

OMG (2004b), *Unified Modelling Language* – http://www.uml.org/, em maio de 2004.

OMG (2004c), *OMG IDL: Details* – http://www.omg.org/gettingstarted/omg_idl.htm, em outubro de 2004.

Orchard, D. (2004), *Versioning XML Vocabularies – SOA Overview – BEA Systems* – http://dev2dev.bea.com/technologies/soa/businesslogic/articles/bizlogic_wilkes.jsp, em outubro de 2004.

OSD (1997), *The Open Software Description Format (OSD)* –W3C – http://www.w3.org/TR/NOTE-OSD, em novembro de 2004.

Osório, A. L., Barata, M. M., Abrantes, Abrantes, A. J., Gomes, J. S., Jacquet, G. C. (2003), *Underlying ITS Business Processes with Flexible and Plugged Peer Systems: The Open ITS-IBUS Approach, In proceedings of the Fourth Working Conference on Virtual Enterprises (PRO-VE '03), IFIP TC5 / WG5.5, October 29-31, 2003, Lugano, Switzerland*.

P2P (2004), *Peer-to-Peer – A WhatIs.com Definition* – http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci212769,00.html, em novembro de 2004.

Parameswaran, M., Susarla, A., Whinston, A. B. (2001), *P2P Networking: An Information-Sharing Alternative – IEEE Computer Society's – Computing Practices, pag. 31, July 2001*.

Peer-to-Peer Working Group (2002), *What is peer-to-peer?* – http://www.peer-to-peerwg.org/whatis/index.html – abril de 2003.

Plug-Ins (2004), *BBCi – Ask Bruce What are 'plug-ins'?* – http://www.bbc.co.uk/webwise/askbruce/articles/download/print/whatareplugins.shtml, em dezembro de 2004.

Pratschner, S. (2001), *Simplifying Deployment and Solving DLL Hell with the .NET Framework – Microsoft Corporation*, November 2001 – http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/dplywithnet.asp, em dezembro de 2004.

Pressman, R. S. (2002), Engenharia de Software – 5ª Edição. McGraw-Hill, Rio de Janeiro, 2002.

Rabelo, R. J.; Klen, A. P.; Klen, E. R., *A Multi-agent System for Smart Coordination of Dynamic Supply Chains, in Proceedings of PRO-VE'2002, 2002*.

RadioJXTA (2004), *Project: radioJXTA* – http://radiojxta.jxta.org/servlets/ProjectHome, em maio de 2004.

Raj, G. S. (1999), *The CORBA Component Model (CCM)* – http://my.execpc.com/~gopalan/corba/ccm.html, em dezembro de 2004.

Razor (2004), *Vipul's Razor: Project Info – SourceForge.net* – http://sourceforge.net/projects/razor/, em novembro de 2004.

Sebrae (2004), Serviço Brasileiro de Apoio às Micro e Pequenas Empresas – www.sebrae.com.br/, em dezembro de 2004.

SeeBeyond (2004a), *Composite Applications and Service-Oriented Architectures – SeeBeyond* – http://www.seebeyond.com/resource/information.asp, em novembro de 2004.

SeeBeyond (2004b), *The SeeBeyond Integrated Composite Application Network Suite 5.0 – SeeBeyond* – http://www.seebeyond.com/*software*/ican.asp, em novembro de 2004.

Seiter, L., Mezini, M., Lieberherr, K. (1999), *Dynamic component gluing. In Ulrich Eisenegger, editor, First International Symposium on Generative and ComponentBased Software Engineering, Springer , 1999.*

Sengupta, S. (2004) – *A Technique for Creating Self Described Data to Enable SOA – BEA Systems* – http://dev2dev.bea.com/technologies/soa/article/som_soa.jsp, em outubro de 2004.

SharedDiary (2004), *Project: shareddiary* – http://shareddiary.jxta.org/servlets/ProjectHome, em maio de 2004.

Skype (2004), *Skype – Free Internet telephony that just works* – http://www.skype.com/, em novembro de 2004.

SOA (2004a), *SOA Resource Center – BEA Systems* – http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/solutions/soa/, em outubro de 2004.

SOA (2004b), *Service-Oriented Architecture – BEA Systems* – http://dev2dev.bea.com/technologies/soa/index.jsp, em outubro de 2004.

SOAP (2003), *SOAP Specifications – W3C* – http://www.w3.org/TR/soap/, em agosto de 2004.

Sparling, M. C. (2001), *Web Services – A Higher Level of Abstraction – CBD-HQ* – http://www.cbd-hq.com/articles/2001/010806ms_webservices.asp, em agosto de 2002.

SPD (2004), Software Package Descriptor – http://openccm.objectweb.org/dtd/ccm/softpkg.dtd, em dezembro de 2004.

Stardock Corporation (2000), *ASPs – A Primer – April/2000* – http://www.stardock.net/media/asp_primer.html, em fevereiro de 2004.

Sun (2004), *Enterprise JavaBeans Technology* – http://java.sun.com/products/ejb/, em maio de 2004.

Sundsted, T. (2001) – *The practice of peer-to-peer computing: Introduction and history – A new-fangled name, but an old and useful approach to computing* – http://www-106.ibm.com/developerworks/java/library/j-p2p/, em novembro de 2004.

Sweeney, P. (1999), *CRM Application Service Providers – Risks and Rewards – November 11, 1999* – http://www.sweeneygroup.com/whitepaper, em agosto de 2002.

Szyperski, C. (1997), *Component Software: Beyond Object-oriented Programming*, Addison-Wesley, 1997.

Thakkar, Y. (2001), *COM, COM+ and .NET : The Difference – .NET Extreme* – http://www.dotnetextreme.com/articles/ComDotNetDiff.asp, em agosto de 2004.

UDDI (2004), *UDDI.org* – http://www.uddi.org/, em agosto de 2004.

Vanmechelen, K. (2003), *A performance and feature-driven comparison of Jini and JXTA frameworks – Universiteit Antwerpen Departement Wiskunde-Informatica, Academiejaar 2002-2003*.

Vasudevan, V. (2001), *A Web Services Primer – WebServices.XML.com* – http://webservices.xml.com/pub/a/ws/2001/04/04/webservices/, em dezembro de 2004.

Waloszek, G. (2003), *Crossing Boundaries with Composite Applications, SAP Design Guild* – http://www.sapdesignguild.org/editions/edition7/print_intro_article.asp, em abril de 2004.

WebServices (2004), *Web Services Activity – W3C* – http://www.w3.org/2002/ws/, em outubro de 2004.

WebOS (1998), *WebOS: Operating System Services for Wide Area Applications* – http://www.cs.duke.edu/ari/issg/webos/, em dezembro de 2004.

Wilkes, S. (2004a), *Loosen Up – Business Logic & Data Services – BEA Systems* – http://dev2dev.bea.com/technologies/soa/businesslogic/articles/bizlogic_wilkes.jsp, em outubro de 2004.

Wilkes, S. (2004b), *Do the Service Salsa – Composite Applications – BEA Systems* – http://dev2dev.bea.com/technologies/soa/compapps/articles/compapps_wilkes.jsp, em outubro de 2004.

Wilkes, S. (2004c), *SOA Much More Than Web Services – XML & Messaging – BEA Systems* – http://dev2dev.bea.com/technologies/soa/xmlmessaging/articles/soa_wilkes.jsp, em outubro de 2004.

Woods, D. (2003), *Packaged Composite Applications: A Liberating Force for the User Interface, SAP Design Guild* – http://www.sapdesignguild.org/editions/edition7/print_composite_applications.asp, em abril de 2004.

WSDL (2001), *Web Services Description Language (WSDL) 1.1 – W3C Note 15 March 2001* – http://www.w3.org/TR/wsdl, em agosto de 2004.

Xml.Com (2004), http://www.xml.com/, em abril de 2004.

Anexo 1 – Exemplo de Contrato PSA

Um típico contrato entre um cliente (por exemplo, uma empresa qualquer) e um PSA detalharia os seguintes itens (ASP, 2004):

1) Jargão: Definição dos termos utilizados entre as partes (“serviços”, “dados”, “aplicações”, “servidores”, “desenvolvedor”, etc.);

2) Acessos: Especificações gerais do que pode ser acessado (sistema, *software*s, processamento e armazenamento de dados), definições sobre como as partes (cliente e PSA) devem proceder no caso de acesso a sistemas do outro contratante (autorizações e responsabilidades);

3) Serviços e melhorias: O PSA deve manter os serviços funcionando corretamente, fazer as atualizações devidas sempre que for necessário, disponibilizar manuais e / ou documentação *on-line* explicando como acessar o sistema e as funcionalidades do *software*, além do suporte técnico e da segurança dos dados armazenados. Se alguma nova funcionalidade for adicionada, o PSA poderá elevar o valor da taxa de utilização. Havendo necessidade de alguma conversão no *software* do cliente, esta deverá ser feita gratuitamente. Além disso, o cliente tem o direito de rejeitar qualquer nova funcionalidade;

4) Desempenho e garantia de disponibilidade: O PSA garante que o tempo máximo em que eventualmente permanecerá desconectado durante o tempo de uso (24 horas por dia, 7 dias por semana) não ultrapassará duas horas;

5) Alteração da licença: O cliente tem o direito de possuir uma cópia local da aplicação, se assim desejar. Esta licença pode ser temporária, e pode ter o seu *status* alterado a qualquer momento, em comum acordo com o PSA;

6) Segurança e recuperação de dados: O PSA deve garantir a integridade dos dados e a sua recuperação em caso de problemas. O PSA deve informar o cliente em caso de alguma mudança (por exemplo, alteração do endereço do *host*) com pelo menos 60 dias de antecedência;

7) Independência: O cliente e o PSA são contratantes independentes. Nenhum dos dois tem o direito de interferir na maneira de atuar do outro e nem na de seus funcionários, desde que o contrato esteja sendo respeitado;

8) Confidencialidade dos dados: Toda informação armazenada no PSA não será repassada a terceiros e só poderá ser usada estritamente na execução dos serviços do cliente;

9) Confidencialidade do cliente: O PSA não irá revelar detalhes particulares do cliente a quem quer que seja;

10) Direitos de propriedade: Todos os dados dos clientes armazenados ou resultantes de algum processamento permanecem como propriedade exclusiva destes;

11) Garantias em geral: O PSA deve garantir ao cliente que é realmente capaz de oferecer os serviços a que se propõe. Em troca, o cliente deve autorizar a manipulação de suas informações pelo PSA. As duas partes se comprometem a não proliferar códigos de qualidade duvidosa, assim como vírus em geral ou “cavalos de tróia”. Nenhum *site* deverá violar nenhuma lei ou regulamentação governamental, ser difamatório, obsceno, racista, pornográfico ou indecente;

12) Infrações: O PSA tem o direito de interferir no serviço do cliente se este estiver infringindo a lei;

13) Publicidade: O nome do cliente não poderá ser utilizado como anúncio de publicidade do PSA;

14) Término do contrato: O cliente pode terminar o contrato a qualquer momento e por qualquer razão, desde que notifique o PSA com 30 dias de antecedência. O PSA pode fazer o mesmo se notificar o cliente por escrito com pelo menos 180 dias de antecedência;

15) Taxas: As taxas são periódicas e devem ser acordadas entre as partes. Devem também variar de acordo com o número de licenças envolvidas e tipo de *software* utilizado.