ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
DEPARTAMENTO DE ENGENHARIA MECATRÔNICA E DE SISTEMAS MECÂNICOS

Nota final
8,5 (oito e cinco)

Comunicação entre Programas para a Simulação Distribuída de Sistemas Produtivos

Caio Klasing Pandolfi

São Paulo
2005

ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
DEPARTAMENTO DE ENGENHARIA MECATRÔNICA E DE SISTEMAS MECÂNICOS

Comunicação entre Programas para a Simulação Distribuída de Sistemas Produtivos

Monografia apresentada à Escola Politécnica da Universidade de São Paulo para relacionado à disciplina PMR2550 – Projeto de Conclusão do Curso II.

Caio Klasing Pandolfi

Orientador:
Prof. Dr. Paulo Eigi Miyagi

São Paulo
2005

RESUMO

Em função da crescente complexidade e aumento de dispersão entre as partes que compõem um sistema produtivo, existe a necessidade de novas ferramentas para apoiar o projeto e avaliação destes sistemas. Neste contexto, tem sido introduzido o conceito de simulação distribuída, mas, para sua efetiva implementação é necessário especificar como se realiza a troca de informações entre entidades que estão simulando partes diferentes de um sistema produtivo. Desta forma, este trabalho analisa técnicas para implementar mecanismos de comunicação entre programas de simulação, e tem por objetivos identificar as características necessárias às conexões que irão realizar a troca de informações entre estas partes e avaliar dois métodos para implementar tais conexões. Foram assim, desenvolvidos programas baseados em *Web-Services* e *CORBA* para integrar aplicativos distribuídos e sugeridas arquiteturas para uma *plataforma de simulação* baseada nestas tecnologias. Através dos estudos e programas implementados foram levantadas as vantagens e desvantagens de cada um dos métodos para a utilização na plataforma de simulação.

Palavras-Chave: Sistemas distribuídos, comunicação entre programas, sistemas produtivos, edes de Petri.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS E ABREVIAÇÕES

BOA	Basic Object Adapter
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
DII	Dynamic Invocation Interface
DNS	Domain Name Service
DSI	Dynamic Skeleton Interface
ESIOP	Environment Specific Inter-ORB Protocol
FMS	Ferramenta de Modelagem e Simulação
FGD	Ferramenta de Gerenciamento de Domínio
GIOP	Generic Inter-ORB Protocol
HTTP	Hyper Text Transfer Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IIS	Internet Information Server
IOR	Interoperable Object Reference
IP	Internet Protocol
IR	Interface Repository
OMA	Object Architecture Model
ISO	International Standarization Organization
MEP	Message Exchange Pattern
OA	Object Adapter
OAD	Object Activation Domain
OMG	Object Management Group
ORB	Object Request Broker

RdP	Redes de Petri
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
TI	Tecnologia de Informação
UDDI	Universal Description, Discovery and Intergration
W3C	World Wide Web Consortium
WS	Web-Service
WSDL	Web-Service Description Language
XML	Extensible Markup Language

1. Introdução

As atuais ferramentas de modelagem e análise de sistemas produtivos apresentam limitações como, por exemplo, quanto: (1) ao grau de detalhamento que permitem e (2) à forma com que os modelos são estruturados (destacando-se a falta de uma abordagem hierárquica de modelagem) (Junqueira et. al. 2005).

Segundo Ho e Cao (1991), é possível considerar sistemas produtivos como sistemas dinâmicos a eventos discretos. Para criar modelos de sistemas deste tipo, as *Redes de Petri* (RdP) são uma ferramenta especialmente bem adaptada para esta tarefa, pois permitem que características como concorrência, sincronismo e paralelismo, típicos em sistemas produtivos, sejam representadas com uma sintaxe e semântica clara e uma representação gráfica intuitiva. Além disso, os modelos em RdP, podem ser objeto de simulação para a análise detalhada da dinâmica do sistema.

Neste projeto, entende-se por *Plataforma de Simulação* um sistema computacional modular e disperso fisicamente, onde as partes dispersas são interligadas através de uma rede de comunicação e operam de forma combinada para realizar a simulação de um modelo de um sistema baseado no conceito de sistemas a eventos discretos, utilizando como método de modelagem as RdP.

Na literatura são encontradas duas abordagens de como gerenciar uma simulação distribuída utilizando-se RdP, a *conservativa* e a *otimista*. (Junqueira et. al.) Uma vez que a *Plataforma de Simulação* estudada neste projeto utiliza uma rede de computadores para processar a simulação, o que implica na não existência de uma memória compartilhada entre os processadores envolvidos na simulação, o que é necessário ao se adotar o método *otimista*,

Observação: As palavras marcadas em itálico no texto, servem para representar termos com um significado específico para esta monografia. O significado de cada uma destas palavras encontra-se no próprio texto. Palavras com a fonte `Courier New` representam termos associados a código de programação ou documentos XML.

a *Plataforma de Simulação* é baseada no método *conservativo*. Além do problema do comgpartilhamento de memória, existe no método *otimista* a necessidade de se guardar informações sobre estados anteriores do modelo, pois existe a possibilidade da ocorrência de eventos inconsistentes durante a simulação. Quando isso ocorre, o simulador precisa voltar atrás na ocorrência do evento e recuperar o estado anterior. Esta volta poderia sobrecarregar demasiadamente o tráfego de dados na rede de comunicação, o que tornaria inviável a adoção do método *otimista* para a estrutura da *Plataforma de Simulação* (Beraldi, Nigro, 1999). Por fim, uma outra característica do método *conservativo*, é a necessidade da existência de sincronismo de operação entre as partes distribuídas da *Plataforma de Simulação*. Este sincronismo garante a consistência na ocorrência de eventos do modelo que está sendo simulado. Na revisão bibliográfica são apresentados os conceitos básicos de RdP e é dada uma explicação sobre o mecanismo de ocorrência de eventos.

Para que a modelagem utilizando a *Plataforma de Simulação* possa ser feita de forma modular, ela mesma deve ser um sistema distribuído, o que gera a necessidade de trocar grandes volumes de informações entre as partes que compõe a plataforma. A interação entre as partes, se dá, por exemplo, pela fusão de eventos. Esta fusão é realizada através da troca de mensagens entre duas partes, o que requer o estabelecimento de uma conexão entre os computadores que estiverem executando as partes. Além disso, pelo fato da *Plataforma de Simulação* basear-se na abordagem *conservativa* para realizar a simulação dos módulos de RdP, torna-se necessário criar um mecanismo que sincronize a evolução do tempo de simulação entre todos as partes envolvidos na simulação. Isto pode ser feito pela circulação de uma mensagem entre eles, contendo a informações correspondentes ao avanço do tempo de simulação. Por fim, para que a estrutura hierárquica da *Plataforma de Simulação* seja mantida e suas partes possam ser gerenciadas, uma conexão entre os computadores e o *gerenciador do domínio* precisa ser estabelecida.

1.1. Motivação e Justificativa

Nas últimas duas décadas e ainda hoje, podem-se identificar as seguintes transformações e fatos que vem ocorrendo na sociedade que motivam e justificam o desenvolvimento da *Plataforma de Simulação* em questão:

- crescimento dos sistemas produtivos de manufatura, acarretando uma maior complexidade de sua estrutura e funcionamento, sendo que muitas vezes estes sistemas são estabelecidos de forma distribuída através do planeta;
- grandes avanços na área de TI que reduzem as barreiras geográficas, aumentaram a velocidade de transmissão, disponibilidade e facilidade de acesso às informações;
- necessidade de ferramentas que auxiliem a tarefa de projetar sistemas de manufatura com as características exigidas atualmente e modificar os já existentes, de forma a reduzir os riscos, custos e o tempo demandado por estas atividades.

Tais fatos sugerem o desenvolvimento de uma nova geração de ferramentas com a capacidade de manipular estes sistemas produtivos. A ferramenta a ser desenvolvida pode ser uma *Plataforma de Simulação* conforme anteriormente definida que tem por objetivo auxiliar a tomada de decisões, fornece análises qualitativas e previsões de comportamento de um dado sistema produtivo através de simulação.

1.2. Objetivos

Este trabalho trata do estudo de mecanismos para implementar a comunicação entre as partes que compõem uma *Plataforma de Simulação* de sistemas produtivos. Os objetivos que se pretendem alcançar através deste estudo são:

- Identificar formas de efetuar e trocar de informações entre programas pertencentes à *Plataforma de Simulação* que estiverem sendo executados em um ambiente distribuído e interligados através de uma rede de comunicação;
- Investigar tecnologias que possam ser utilizadas para implementar as conexões e relacionar seu uso à *Plataforma de Simulação*, de forma a torná-la um sistema flexível e escalável. Neste estudo, as tecnologias investigadas são *Web-services* e CORBA, dois padrões desenvolvidos para serem utilizados em aplicativos distribuídos;
- Desenvolver programas de teste que implementem conexões entre computadores através das tecnologias investigadas;
- Comparar os dois métodos testados quanto a implementação das conexões e avaliação de qual se adapta melhor ao uso na *Plataforma de Simulação*.

2. Conceitos Fundamentais

2.1. Características dos Sistemas Produtivos Dispersos

Desde 1980 tem-se observado uma profunda reestruturação da indústria e da economia. Esta reestruturação se refere às mudanças na forma como bens e serviços são desenvolvidos, projetados e produzidos. Os principais fatores que impulsionaram estas mudanças foram; a evolução da tecnologia de comunicação e transporte, o acelerado ritmo de transferência de tecnologia, as reformas sociais e econômicas e as modificações nos hábitos de consumo dos países desenvolvidos e em desenvolvimento, gerando uma intensa competição global entre empresas (Shi; Gregory, 1998; Lee; Lau, 1999). Com os mercados se tornando mais globais e independentes de barreiras geográficas, um maior número de indústrias de manufatura tem se estabelecido de forma distribuída, aproveitando o crescimento das redes de comunicação e *Tecnologia da Informação* (TI), permitindo, entre outras vantagens, uma maior cooperação transnacional. De forma a aproveitar as oportunidades num mercado global e lidar com novos padrões de competição, os sistemas produtivos devem apresentar novas características permitindo cobrir a dispersão geográfica e a coordenação interdependente, além do tradicional foco em diversas plantas produtivas. A incorporação da TI nos processos industriais está permitindo a integração dos diversos sistemas da empresa, em seus diferentes níveis (Sanz; Alonso, 2001). Além disso, os sistemas de supervisão e controle (responsáveis pelas operações do chão de fábrica), típicos em uma instalação moderna de médio e grande porte são compostos por uma coleção heterogênea de dispositivos de *hardware* e *software* que se encontram distribuídos, compondo sistemas e subsistemas

(estações de operação, unidades remotas, processos computacionais, controladores programáveis, etc.) interligados através de sistemas de comunicação (*cabeamentos analógicos, linhas seriais, fieldbuses, LANs* ou mesmo comunicação via satélite) heterogêneos.

Em um sistema produtivo disperso a troca de informações e materiais entre as empresas requer uma alta flexibilidade o que aumenta a complexidade dos sistemas.(Junqueira, Miyagi, Villani, 2005). Portanto, torna-se imprescindível utilizar métodos de validação do sistema, para evitar erros e falhas. Esta validação não é um processo trivial, mas pode ser realizado aproveitando-se os avanços na área de TI e da expansão do uso das redes de comunicação como as *Intranets* e a própria *Internet*, que permitem criar uma nova geração de *Plataformas de Simulação* de sistemas. Esta plataforma pode ser utilizada para realizar a validação de um sistema de forma distribuída através de simulações feitas em computadores, onde o processamento é dividido entre diversos computadores conectados através de uma rede de comunicação. Esta abordagem torna possível tratar modelos de sistemas relativamente mais complexos do que as ferramentas de simulação da atualidade são capazes de manipular.

2.2. Simulação em computadores

O termo "simulação" tem vários significados atribuídos a ele. Usualmente, refere-se à representação do comportamento de algo maior, ou de uma atividade mais complexa. Todos os simuladores usam um *modelo* que é uma representação do comportamento de um *sistema* que pode ou não existir e que geralmente é maior e mais complexo que o modelo (Seila; 1995).

O modelo pode ser físico, ou pode ser representado por um programa de computador, como no exemplo dos processos de um sistema produtivo. Em todos os casos, a idéia principal

é que a simulação é uma realização alternativa do comportamento de um sistema, que pode ser testado e avaliado em diversas circunstancias.

A Simulação feita em computadores já vem sendo usada desde a década de 1950 e a idéia por trás dessas simulações é testar estratégias de ação através de modelos simplificados da realidade que podem ajudar na tomada de decisão no mundo real. Nem todos os sistemas reais são apropriados para uma modelagem computacional, porém em muitos casos, modelos feitos em computadores são a melhor forma de se desvendar o comportamento de um sistema real.

É preciso deixar claro que existem diferentes formas de se fazer uma simulação, sendo a utilização de computadores apenas uma delas. É possível, por exemplo, fazer testes diretamente no mundo real de forma restrita, porém muitas vezes isto se torna impraticável, seja pelo custo ou pelo risco de se realizar tais testes. Uma outra forma de realizar simulações é modelar um sistema matematicamente, caracterizando equações que se adaptam ao comportamento sistêmico. Tal abordagem pode se tornar impraticável para sistemas muito complexos onde a obtenção do modelo matemático e sua solução seriam impossíveis.

A terceira opção é simular um sistema em um modelo computacional e então testá-lo para várias estratégias e ver qual se adapta melhor à realidade.

A simulação feita por computadores melhor se aplica e tem um melhor desempenho quando:

- os sistemas são dinâmicos e variam de comportamento ao longo do tempo. Esta variação no tempo ocorre devido a fatores desconhecidos que precisam ser tratados estatisticamente, ou então podem ser representados por equações que os computadores conseguem resolver;

- os sistemas são interativos, ou seja, o sistema é formado de vários elementos que interagem entre si e com isso resultam em um comportamento diferenciado. A interação de muitos fatores pode ser processada por computadores;
- os sistemas são complexos e existem muitos fatores interagindo entre si e cada fator funciona de maneira complexa e precisa ser analisado com cuidado;
- as simulações têm maior credibilidade junto aos tomadores de decisão do que outros métodos disponíveis.

Apesar de poder ser aplicado a sistemas com certa complexidade, os resultados obtidos através de simulações não são exatos, indicando apenas um intervalo de confiança para os parâmetros de saída, que não são exatamente o comportamento do sistema real e, portanto estão sujeitos a variações. Além disso, construir simulações pode ser relativamente caro, porque os modelos precisam ser verificados e validados minuciosamente para que forneçam dados confiáveis.

2.2.1 Como criar Modelos para Simulação

O modelo de um sistema é o coração de qualquer simulação feita em computadores. Portanto, a primeira etapa a ser realizada, é a obtenção deste modelo (Pidd; 1994).

Para criar um modelo, é necessário observar e coletar dados do sistema que se deseja analisar. É importante notar que pessoas diferentes podem ter interpretações diferentes de um sistema e seu funcionamento, portanto o mesmo sistema pode ser modelado de diferentes formas (Centeno; 1996).

Através da interpretação da realidade e da análise dos dados colhidos, chega-se a um modelo conceitual do sistema. É a partir do modelo conceitual que um programa de computador é criado. Este modelo funciona de forma mais simples do que a realidade. Portanto modelagem em computador é um trabalho de simplificação e abstração no qual o

modelador tenta identificar os fatores cruciais do sistema do qual se deseja estudar o comportamento através de simulação. Este processo depende da quantidade de informações que se tem do sistema, assim como o propósito pelo qual o modelo está sendo criado (Pidd; 1994).

Em muitos casos, o modelo a ser elaborado não está baseando em um sistema que existe no mundo real, mas sim em uma idealização abstrata de um sistema que se deseja criar. Nestes casos, o modelo é utilizado para analisar se as necessidades do sistema idealizado serão atendidas.

2.3. Conceitos básicos de Redes de Petri (RdP)

O texto que segue, trata-se de trechos adaptados de Moore e Brennan (1996).

Uma das formas de se representar sistemas a eventos discretos, é por RdP. As RdP são uma técnica de modelagem matemática e gráfica, desenvolvida por C. A. Petri no início da década de 1960, para caracterizar operações concorrentes em sistemas de computadores. As RdP evoluíram de forma a contemplarem muitos aspectos importantes de sistemas de grande porte, incluindo atributos, relações de tempo, e eventos estocásticos. A grande vantagem da utilização de RdP é sua simplicidade conceitual e seu caráter genérico, que permite que sejam utilizadas para modelar sistemas bastante variados.

As RdP consistem de quatro elementos primitivos (*marcas*, *lugares*, *transições* e *arcos*) e as regras que definem suas operações. As RdP são baseadas na idéia de *marcas* que se movem através de uma rede. Essas *marcas* são representadas por pontos e representam os *objetos* ou *entidades* em um sistema. (Deve-se tomar cuidado para não confundir os *objetos* mencionados aqui, com os *objetos* definidos na hierarquia dos modelos da *Plataforma de Simulação*, na introdução. Apesar da duplicidade de significado atribuído à palavra "*objeto*", o contexto em que o termo estiver inserido deixará claro qual o significado atribuído a ele.) Os

lugares são representados por círculos e representam as localidades onde *objetos* aguardam um processamento. Os *lugares* podem representar espaços físicos (por exemplo, uma fila onde uma peça aguarda para ser processada), ou um *estado* (recurso ocioso).

Transições são representadas por retângulos e indicam processos ou *eventos* (por exemplo, ou a usinagem de uma peça). Finalmente, os *arcos* representam as ligações ou caminhos entre *objetos* de um sistema. Os *arcos* conectam os *lugares* às *transições* e as *transições* a *lugares*; a direção de um *arco* é indicada por uma seta na sua extremidade Figura 2-1.

Figura 2-1 - (a) marca (b) lugar (c) transição (d) arco

As *regras de disparo* de uma RdP definem o comportamento das *transições*, ou seja, a condição necessária para que um processo ou *evento* possa ocorrer. Três regras regem o disparo de uma *transição*:

Quando todos os *lugares* que antecedem a *transição* estão ocupados com pelo menos uma marca, a *transição* está ativa. Uma vez ativa, a *transição* dispara. Quando a *transição* dispara, uma *marca* é retirada de cada *lugar* exatamente antes da *transição*, e uma *marca* é colocada em cada *lugar* subseqüente à *transição*.

A **Erro! Fonte de referência não encontrada.** ilustra as regras para uma *transição* (agrupamento) que contém dois *lugares* conectados a montante (A e B) e um *lugares* a jusante (montagem).

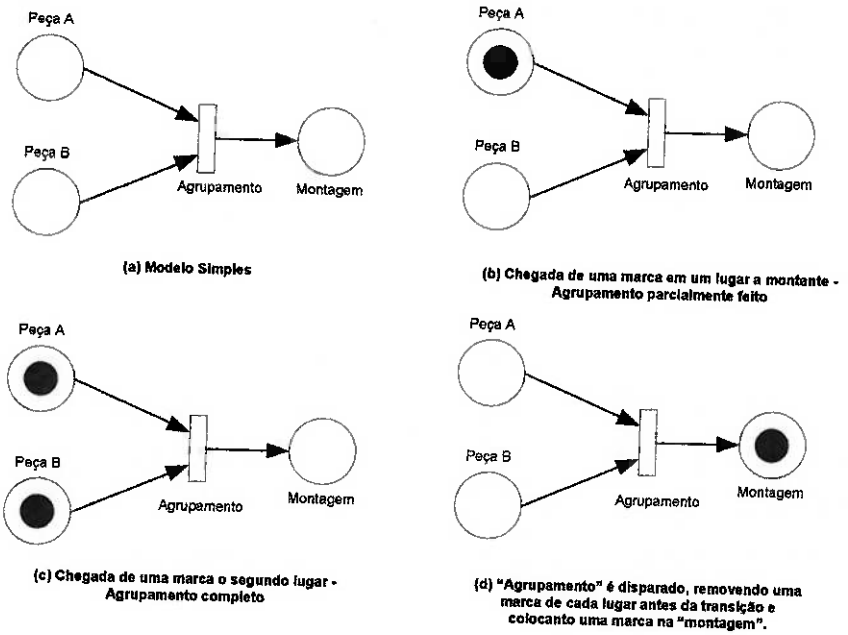


Figura 2-2 - Disparo de transição

Regras de tempo podem ser associadas às *transições* e representam o tempo necessário para se completar uma atividade. Uma regra de tempo pode ser estocástica, baseada em uma função probabilística, um valor computado, ou uma constante. Regras de decisões são associadas aos *lugares* e decidem sobre problemas onde uma ou mais *marcas* ativam mais do que uma *transição* ao mesmo tempo. Existem três tipos de regras de decisão: *prioridade*, *probabilidade* e *estruturada*. A regra de decisão de *prioridade* estabelece que, se todas as outras regras de decisão coincidem, as *marcas* irão seguir o caminho com a mais alta prioridade. A regra de decisão de *probabilidade* incita que se todas as outras regras de decisão coincidem, as marcas tomarão um caminho baseado em uma probabilidade. A regra de decisão *estruturada* permite que o usuário especifique a condição sob a qual uma *marca* selecionará um caminho, no caso das regras de decisão coincidirem. Exemplos de regras são mostrados na Figura 2-3.

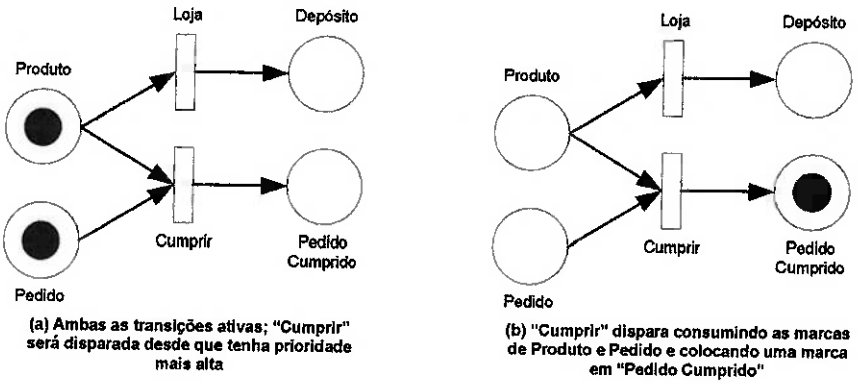


Figura 2-3 - Efeito da regra de Decisão de Prioridade

Atributos são utilizados para especificar um conjunto de características associadas a uma *marca* (tamanho, tipo, prioridade, identidade, etc.). O valor destes *atributos* pode ser modificado nas *transições*. Eles podem ser utilizados também para determinar regras de tempo e decisão. O valor dos *atributos* pode ser passado para algoritmos externos e os resultados incorporados ao modelo de RdP.

Existem ainda outros dois tipos de *arcos* além do *arco* padrão, estabelecem outras condições para o disparo da *transição*, porém não serão abordadas neste tópico introdutório ao assunto. Maiores informações sobre o assunto podem ser encontrados em Morata (1989).

2.4. Estrutura dos modelos da Plataforma de Simulação

Para alcançar a modularidade necessária aos modelos do sistema a serem simulados na *Plataforma de Simulação*, foi incorpora-se o *paradigma de orientação a objeto* ao formalismo das RdP. O *paradigma de orientação a objetos*, permite que um modelo seja composto de um conjunto de sub-modelos que interagem entre si. A interação entre os sub-modelos é feita através da troca de mensagens entre eles. Os dados que devem ser trocados assim como sua correta interpretação, dependem da definição de uma interface que descreva quais as informações que devem ser trocadas e como um objeto deve ser acessado. Com este tipo de abordagem, pode-se, portanto utilizar uma rede de comunicação para executar uma simulação

em um ambiente onde os sub-modelos encontram-se distribuídos entre diversos computadores.

Seguindo o *paradigma da orientação a objetos*, pode-se dizer que o sistema distribuído é composto de um conjunto de *objetos* que interagem entre si com o *objetivo* de executar certa tarefa, chamado *aplicativo*, onde um *objeto* é uma instância de uma *classe* e uma *classe* representa um elemento genérico de um sistema. Por sua vez, o comportamento de uma *classe* é definido por uma RdP. Uma definição mais completa destes elementos é dada a seguir:

- *Classes* correspondem à modelagem do sistema real (ou partes deste), como suas funcionalidade de interesse, utilizando-se para tanto as RdP. A representação gráfica das *classes* é mostrada na Figura 2-4.

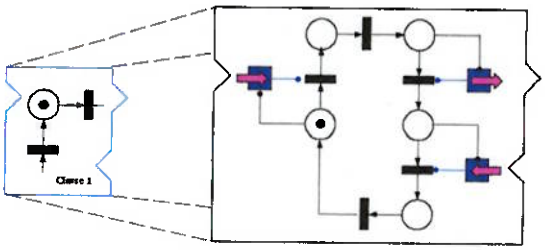


Figura 2-4 - Representação gráfica de uma classe

- *Componentes* são os agrupamentos de um ou mais *objetos* bem como de zero ou mais *componentes*. Entende-se por *objeto*, uma instância de uma *classe* existente. A representação gráfica dos *componentes* é mostrada na Figura 2-5.

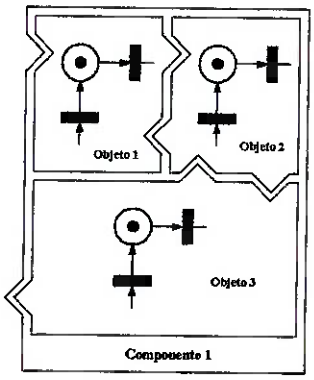


Figura 2-5 - Representação gráfica de um componente

- *Aplicativos* são compostos por um ou mais *componentes* para criar o modelo completo que se deseja simular. A representação gráfica dos *componentes* é mostrada na Figura 2-6.

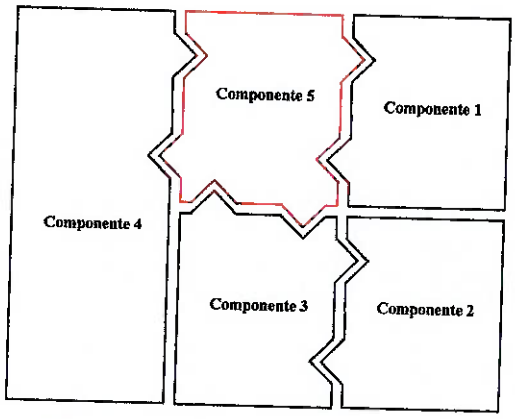


Figura 2-6 - Representação gráfica de um *aplicativo*, mostrando a interação entre os *componentes*

A interação entre dois *objetos* é feita través de chamadas a métodos disponibilizados por ele. A chamada a um método se dá através da troca de mensagens entre dois *objetos*. Para a RdP que implementa a *classe* do *objeto*, a troca de mensagens entre dois *objetos* representa a fusão de duas *transições* (sendo uma *transição* do *objeto* que faz a chamada e a outro do *objeto* invocado com a chamada).

Com essa estrutura modular dos modelos da *Plataforma de Simulação*, pode-se criar modelos de uma forma descentralizada envolvendo diversos modeladores trabalhando em diversos módulos simultaneamente. Para se ter proveito desta característica do método de modelagem, a *Plataforma de Simulação* em si deve ter a estrutura de um sistema distribuído.

O sistema distribuído pelo qual a *Plataforma de Simulação* é composto apresenta uma estrutura física composta por computadores interligados por uma rede de comunicação. Os computadores que fazem parte da *Plataforma de Simulação* podem ser classificados de acordo com o tipo de tarefa que irão desempenhar, e cujas definições são dadas a seguir:

- *Estação*: São os computadores que contém o ambiente de modelagem e simulação da plataforma. As *estações* são nós da rede de comunicação utilizada pela

Plataforma de Simulação para trocar informações entre computadores. Nas *estações* o usuário cria modelos e requisita o processamento de uma simulação;

- *Gerenciador*: Os *gerenciadores* são computadores que, assim como as *estações*, representam nós da rede de comunicação utilizada pela *Plataforma de Simulação*. A diferença em relação às *estações* é que os gerenciadores executam o programa de gerenciamento da *Plataforma de Simulação*. O gerenciador também pode conter o ambiente de modelagem e simulação.

A Figura 2-7 ilustra um exemplo de uma possível arquitetura física para a *Plataforma de Simulação*.

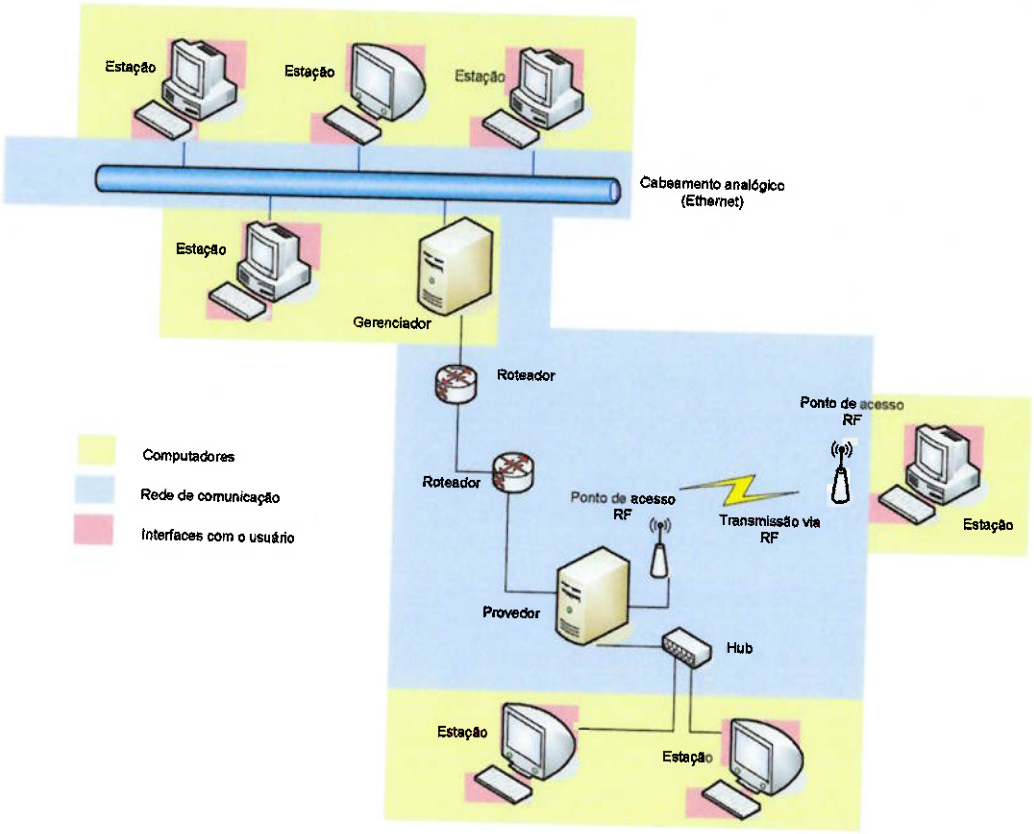


Figura 2-7 - Exemplo esquemático da arquitetura física da *Plataforma de Simulação*

Assim como existem duas classificações para os computadores que compõem a *Plataforma de Simulação*, existem também dois tipos de programas, um programa para ser executado nas *estações* e um segundo programa para ser executado nos *gerenciadores*. Estes

programas trocam informações entre si através da rede de comunicação e são descritas a seguir:.

A *Ferramenta de Modelagem e Simulação* (FMS) é o programa utilizado para criar as classes baseadas no formalismo das RdP e processar simulações de forma distribuída. Este programa é executado nas *estações* e possui uma interface gráfica para que o usuário possa interagir com ele, e pode conectar-se a outros programas da *Plataforma de Simulação* quando estiver executando uma simulação ou procurando por arquivos de modelos na rede.

A *Ferramenta de Gerenciamento de Domínio* (FGD) é um programa que gerencia e integra todos os programas e máquinas que compõem a *Plataforma de Simulação* e é executado nos *gerenciadores*. A FGD tem as seguintes funções:

- Estabelecer conexões com as FMSs de uma Plataforma de Simulação;
- Armazenar informações sobre modelos criados nas FMSs;
- Armazenar informações sobre status das FMSs;
- Receber pedidos de processamento de simulações das FMSs;
- Viabilizar o processamento de simulações de forma distribuída, dividindo-o entre as FMSs.

A Figura 2-8 mostra de forma esquemática os programas que compõem a *Plataforma de Simulação*.

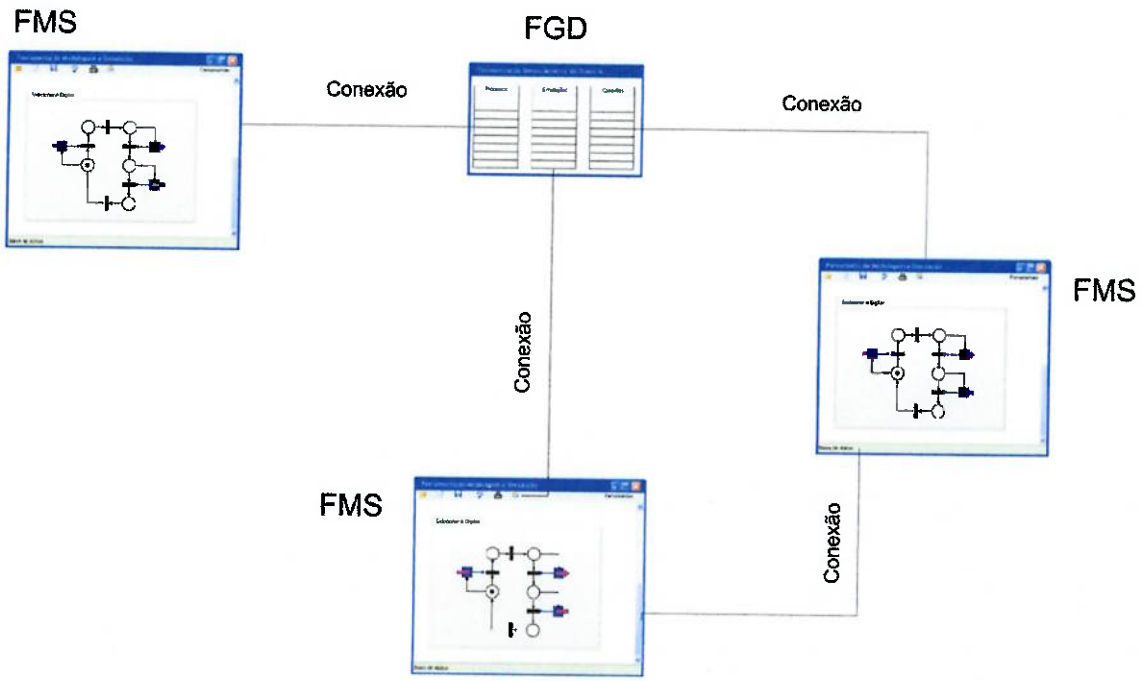


Figura 2-8 - Exemplo esquemático da arquitetura de programas na *Plataforma de Simulação*

Para organizar a forma pela qual se dá a interação entre os programas da *Plataforma de Simulação*, é definida uma hierarquia entre eles, baseada em três níveis de abstração definidos a seguir:

Um *domínio* é um conjunto de *estações* conectadas a um *gerenciador* (Figura 2-9). Dentro de um *domínio*, tem-se apenas um *gerenciador* que executa a FGD e uma ou mais *estações* que executam a FMS conectadas a ele. Além disso, o usuário que estiverem operando uma *Ferramenta de Modelagem e Simulação* conectada a um *domínio*, tem acesso ilimitado aos modelos pertencentes a ele.

Uma *federação* é o resultado de um acordo de cooperação entre *domínios*, que permite que membros de um *domínio*, utilizem modelos desenvolvidos em outros *domínios*, pertencentes a uma mesma *federação*. Como conseqüência direta, uma *federação*, para existir, deve ser composta por dois ou mais *domínios* (Figura 2-9). Ao contrário do que ocorre para membros de um mesmo *domínio*, membros de *domínios* distintos, apesar de poderem utilizar modelos pertencentes à *federação*, não possuem acesso à implementação dos mesmos. Para que esta troca de informações entre domínios possa acontecer, é preciso conectá-los de

alguma forma. Isto é feito através de uma conexão entre os *gerenciadores* de cada *domínio* de uma *federação*.

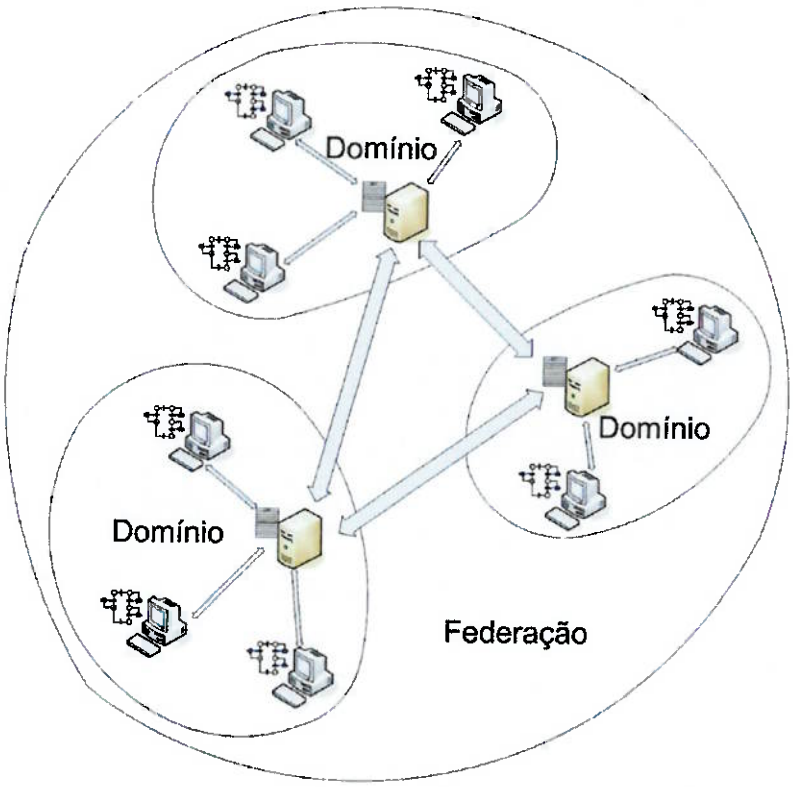


Figura 2-9 - Esquema da estrutura de domínios e federações na *Plataforma de Simulação*.

A arquitetura e hierarquia apresentadas para a *Plataforma de Simulação*, oferecem uma grande flexibilidade à plataforma, pois torna sua estrutura escalável, onde podem ser adicionados e removidos computadores, adaptando o tamanho da plataforma à complexidade do sistema que se deseja modelar.

Com as informações apresentadas sobre as RdP, evidencia-se como funciona a estrutura de módulos nos quais a *Plataforma de Simulação* é baseada. A informação que deve ser trocada entre módulos é em qual instante uma *marca* aparece ou é apagada de um *lugar* e qual é a *transição* ativa a inclusão ou exclusão de uma *marca* nos *lugares*. É importante lembrar, que os *objetos* que participam deste processo, podem estar localizados no mesmo computador, ou então em computadores diferentes, porém interligados através de uma rede de comunicação.

A seguir serão apresentadas duas tecnologias utilizadas para o desenvolvimento de plataformas distribuídas, CORBA e WS.

2.5. CORBA

Manipular a heterogeneidade de plataformas distribuídas é uma tarefa complexa. Em particular, desenvolver programas que utilizam de forma eficiente os recursos distribuídos através de uma rede de conexões torna-se um grande desafio. Muitas interfaces de programação e pacotes prontos com implementações destes recursos existem atualmente para auxiliar o desenvolvimento deste tipo de programas (Vinoski, 1997).

Em função das dificuldades nesta área de desenvolvimento e programas, foi fundado a *Object Managemnet Group* (OMG), em 1989, para elaborar, adotar e promover padrões para o desenvolvimento de programas em um ambiente heterogêneo distribuído. Desde que surgiu, a OMG cresceu e tornou-se um dos maiores consórcios de desenvolvimento de padrões para programas nesta área. Os membros do consórcio contribuem com idéias e tecnologias em resposta a pedidos de propostas enviadas a OMG. As respostas vêm em forma de especificações baseadas em métodos com alta disponibilidade e difusão no mercado da informática (OMG, 2005).

A seguir tem-se uma introdução ao conceito de *Object Management Architecture* (OMA) elaborado pela OMG, com foco em seu conjunto principal de especificações, o *Common Object Request Broker Architecture* (CORBA).

2.5.1 OMA

O OMA é um modelo baseado no *paradigma da orientação a objetos*, composto por um *Object Model* e um *Reference Model*. O *Object Model* define como *objetos distribuídos*

podem ser descritos, enquanto o *Reference Model* caracteriza as interações entre estes *objetos*. Criando-se especificações que se enquadram nos conceitos do OMA, estas permitem o desenvolvimento de interoperabilidade entre *objetos* distribuídos em ambientes heterogêneos.

No *Object Model*, um *objeto* é uma entidade encapsulada com uma identidade distinta e imutável, cujo *serviço* pode ser acessado através de uma interface pré-definida. Entende-se por *serviço* uma funcionalidade que o *objeto* disponibiliza e que pode ser acessada por outros *objetos*. Clientes fazem uma requisição de algum *serviço* oferecido por um *objeto* utilizando as interfaces definidas de acordo com o *Object Model*.

A Figura 2-10 mostra os componentes do *Reference Model*. O *Object Request Broker* (ORB) é responsável por facilitar a comunicação entre *objetos*. Existem quatro categorias de componentes que utilizam a especificação de interfaces para interagirem com o ORB, definidas a seguir:

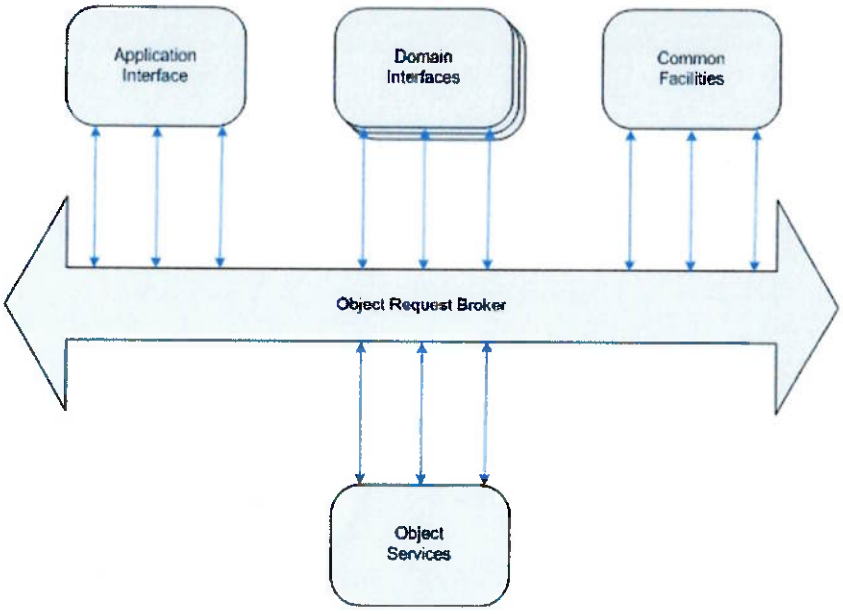


Figura 2-10 - Categorias de interfaces do *reference model* do OMA

Object Services interfaces são interfaces independentes da aplicação, que são utilizadas por muitas plataformas baseadas em *objetos* distribuídos. Por exemplo, um *serviço* que tem por finalidade procurar outros *serviços* disponíveis é necessário em praticamente qualquer plataforma. Dois exemplos de *Object Services* que desempenham esta tarefa são:

Se a definição de *tipos* do resto da plataforma mudar futuramente, de tal forma que se torne incompatível com as definições contidas no programa, o programa precisa ser recompilado com as novas informações.

No entanto, para alguns programas, o conhecimento estático sobre um IDL de definição de *tipos* é impraticável. Considere por exemplo, um *gateway* que permite que programas que utilizam outros tipos de programação distribuída (como o COM da *Microsoft*) acessem *objetos* CORBA. Se for necessário recompilar o *gateway* toda vez que um novo IDL de definição de *tipos* é adicionado à plataforma, o resultado seria uma grande dificuldade para gerenciar e manter o *gateway*. Ao invés de proceder desta forma, é melhor que o *gateway* possa encontrar e utilizar informações sobre tipos dinamicamente, na medida em que fosse necessário.

O CORBA *Interface Repository* (IR) permite que um IDL de definição de *tipos* seja acessado e programado em tempo de execução. O IR em si é um *objeto* CORBA cujas operações podem ser invocadas como em qualquer outro *objeto* CORBA. Utilizando o IR, programas podem percorrer uma hierarquia completa de informações sobre IDLs.

Como o IR permite que programas encontrem informações sobre *tipos* em tempo de execução, sua utilidade real está no suporte ao CORBA *dynamic invocation* (que é descrito no tópico *Dynamic Invokation e Dispatch*). O IR pode ser utilizado também como uma forma de gerar código estático para programas, como será visto no próximo tópico, já que as definições de um IDL em um IR são equivalentes às definições obtidas diretamente pelo arquivo que contém a definição do IDL.

2.5.2.5 Stubs e Skeletons

Além de criar *tipos* de linguagens de programação, os compiladores e tradutores de IDL geram *client-side stubs* e *server-side skeletons*. Um *stub* é um mecanismo que cria um

request quando demandado pelo cliente. Os *skeletons* encaminham os *requests* até a implementação do *objeto* CORBA. Como estes dois mecanismos fazem parte das especificações da OMG referente às IDLs, os *stubs* e *skeletons* costumam ser específicos para cada interface.

A transação entre *stubs* e *skeletons* muitas vezes é chamada de *static invocation*, pois estes são criados diretamente dentro do aplicativo cliente e da implementação do *objeto* CORBA. Portanto, os *stubs* e os *skeletons*, precisam conhecer a IDL do objeto CORBA que está sendo invocado *a priori*.

Language mapping usualmente mapeiam invocações de operações para algo que seja equivalente à chamada de uma função. Por exemplo, dado uma referência a um *objeto* chamado *Factory* em C++, o código que gera um *request* é visto na Figura 2-14:

```
// C++
Factory_var factory_objref;
// Initialize factory_objref using Naming or
// Trading Service (not shown), then issue request
Object_var objref = factory_objref->create();
```

Figura 2-14 - Código utilizado para gerar um request

Este código faz com que uma invocação da operação `create` no *objeto* que se deseja acessar, pareça uma chamada rotineira de uma função na linguagem C++. No entanto, esta chamada está invocando um *stub*. Pelo fato do *stub* estar integrado no programa que faz o *request* para o *objeto* (provavelmente implementado remotamente), este costuma ser chamado de *surrogates* ou *proxies*. O *stub* interage diretamente com o ORB do cliente para "codificar" (*marshal*) o *request*, ou seja, o *stub* auxilia na conversão do *request* de sua representação na linguagem de programação para uma representação adequada que será transmitida através da conexão até o *objeto* alvo.

Quando o *request* chega ao *objeto* alvo, o ORB do servidor e o *skeleton* trabalham de forma conjunta para "decodificar" (*unmarshal*) o *request* (convertê-lo do formato utilizado

para a transmissão, para o formato da linguagem de programação) e encaminhá-la até o *objeto*. Quando o *objeto* processa o *request*, a resposta é enviada de volta pelo mesmo caminho que percorreu na vinda: através do *skeleton*, passando pelo ORB do servidor, pela conexão e de volta para o ORB do cliente e pelo *stub*, antes de chegar ao programa cliente. A Figura 2-12 mostra onde o *stub* e o *skeleton* estão alocados em relação ao programa cliente, o ORB e a implementação do *objeto*.

Nota-se, portanto que os *stubs* e *skeletons* têm um papel importante na conexão entre o ambiente de linguagem de programação e o ORB. Neste sentido, estes se assemelham ao padrão de *Adapter* e *Proxy*. O *stub* adapta a forma de se chamar uma função através dos *language mappings* para o mecanismo de invocação de um *request* do ORB. O *skeleton* adapta o mecanismo de entrega do *request* do ORB à forma de chamada compatível com a implementação do *objeto*.

2.5.2.6 Dynamic Invokation e Dispatch

Além das *static invocations* que podem ser feitas pelos *stubs* e *skeletons*, CORBA contém duas interfaces para invocações dinâmicas, que são:

- *Dynamic Invokation Interface* (DII) – que permite que *requests* de clientes sejam invocados de forma dinâmica;
- *Dynamic Skeleton Interface* (DSI) – que permite a entrega de *requests* aos *objetos* de forma dinâmica.

Tanto o DII quanto o DSI podem ser vistos como um *stub* e um *skeleton* genéricos, respectivamente. Cada um deles é uma interface disponibilizada diretamente pelo ORB e nenhum dos dois depende da IDL do *objeto* que está sendo invocado.

Dynamic Invokation Interface (DII)

Usando o DII, um programa cliente pode invocar *requests* para qualquer *objeto* sem previamente ter conhecimento de sua interface. Por exemplo, considerando o *objeto gateway* descrito acima. Quando uma invocação é recebida pelo *objeto* estrangeiro, o *gateway* precisa transformar esta invocação em uma entrega de um *request* para o *objeto* CORBA desejado. Recompilar o programa do *gateway* para incluir novos *stubs* estáticos toda vez que um novo *objeto* CORBA é criado torna-se impraticável. Ao invés disso, o *gateway* pode simplesmente utilizar o DII para invocar *requests* para qualquer *objeto* CORBA. O DII é também útil para programas interativos como *browsers* que podem obter valores necessários para os argumentos que um *objeto* necessita diretamente do usuário.

É através de operações do tipo `create_request` disponibilizadas pela interface `CORBA::Object`, que aplicativos criam *pseudo-objetos* de *requests*. Como toda IDL herda o `CORBA::Object`, todo *objeto* automaticamente contém a operação `create_request`. Ao executar esta operação para uma referência de um *objeto* que se deseja acessar, um programa pode criar um *request* dinâmico para este *objeto*. Antes do *request* poder ser invocado, os argumentos necessários precisam ser preenchidos, invocando-se operações diretamente no *pseudo-objeto request*. Os tipos de argumentos podem ser determinados utilizando-se o *Interface Repository*.

Quando o *pseudo-objeto request* tiver sido criado e os argumentos necessários tiverem sido adicionados a ele, este pode ser invocado de três formas:

- *Synchronous Invocation*: o cliente invoca o *request* e então bloqueia o restante do processamento enquanto aguarda a resposta. Do ponto de vista do cliente, isso

equivale ao comportamento de uma RPC[4]. Este é o método mais usual para invocar *requests* em programas que utilizam CORBA, pois é utilizado também em *stubs* estáticos;

- *Deferred Synchronous Invokation*: o cliente invoca o *request* e continua o processamento do restante do programa enquanto o *request* é encaminhado, coletando a resposta em um instante posterior. Isto é útil quando um cliente precisa invocar um grande número de *serviços* independentes que requerem um tempo prolongado de processamento. Ao invés de invocar cada *request* em série e aguardar pelas respostas seqüencialmente, todos os *requests* são processados em paralelo e as respostas podem ser coletadas na medida em que são retornadas;
- *Oneway Invocation* – O cliente invoca o *request* e continua o processamento do programa sem coletar uma resposta. Esta forma é chamada de "*fire and forget*", pois a única maneira que um cliente tem de saber se um *request* foi recebido é por alguma outra via, como por exemplo, fazer com que o *objeto* invoque um *request* separadamente quando o primeiro *request* (enviado pelo cliente) tiver sido processado.

Apesar da flexibilidade oferecida por DIIs, é preciso estar ciente das desvantagens associados ao seu uso. Criar um *request* DII pode causar que o ORB acesse de forma transparente o IR para obter informações sobre os tipos de argumentos e respostas de um dado *serviço*. Como o IR é em si um *objeto* CORBA, cada *request* IR transparente feito pelo ORB pode na realidade ser uma invocação remota. A criação e invocação de um único *request* DII poderia demandar várias invocações remotas, fazendo com que um *request* DII torne-se mais lento e exija mais processamento do que uma invocação estática equivalente. Invocações

---

[4] RPC: Um mecanismo que permite que rotinas que se encontram em computadores remotos sejam executadas através de uma rede de comunicação. A comunicação com estas rotinas é feita passando-se argumentos, de forma que a comunicação entre os computadores fique oculta ao programa que faz a chamada à rotina.

estáticas não dependem do processo demorado de acessar o IR já que as informações sobre os *tipos* envolvidos no *request* estão compiladas diretamente no programa.

Dynamic Skeleton Interface (DSI)

De forma análoga ao DII, o lado do servidor dispõe de *Dynamic Skeleton Interfaces* (DSI). Assim como o DII permite que clientes invoquem *requests* sem ter acesso a *stubs* estáticos, o DSI permite que servidores possam ser codificados sem possuírem *skeletons* para os *objetos* que estão sendo invocados e que normalmente seriam compilados estaticamente dentro do programa.

O *objeto* externo *gateway* descrito anteriormente é um bom exemplo de um programa que necessita do mecanismo de DSI. Um *gateway* bidirecional precisa ser capaz de comportar-se tanto como um cliente quanto como um servidor, traduzindo *requests* de *objetos* estrangeiros em *requests* de *objetos* CORBA e transformar *requests* de programas CORBA em invocações de *objetos* estrangeiros. Como mencionado anteriormente, o programa pode utilizar DIIs quando desejar atuar como cliente. Para atuar como servidor, no entanto, o programa necessita de um mecanismo equivalente ao DII, permitindo que ele receba *requests* sem necessitar *skeletons* estáticos para cada interface de *objeto* compilado dentro dele. Recompilar o *gateway* cada vez que uma nova IDL for introduzida no âmbito dos *serviços* CORBA, não funcionaria de maneira satisfatória na prática.

Diferente da maioria dos outros sub-componentes CORBA, que faziam parte das especificações CORBA inicialmente, o DSI foi introduzido na versão 2.0. A razão principal para sua introdução foi de criar uma forma de implementar *gateways* entre ORBs, utilizando diferentes protocolos de comunicação. Apesar dos protocolos inter-ORB também terem sido introduzidos na versão 2.0, acreditava-se que os *gateways* tornariam-se a escolha predominante para operações entre ORBs. Pelo fato da maioria dos ORBs disponíveis

comercialmente já terem suporte ao *Internet Inter-ORB protocol* (IIOP) (descrito mais adiante), as previsões quanto ao uso de *gateways* não se tornaram realidade. Ainda assim, o DSI é uma ferramenta útil para certos tipos de aplicações, especialmente para criar "pontes" entre ORBs e entre *serviços* CORBA e *serviços* de outros tipos.

2.5.2.7 Object Adapters

O *Object Adapter* (OA) serve como uma espécie de "cola" para implementações de *objetos* CORBA e o ORB. Como descrito no padrão *Adapter*, um OA é um *objeto* que adapta a interface de outro *objeto* à interface esperada por quem faz a chamada. Em outras palavras, o OA é um *objeto* que permite que um *request* seja invocado sem que o cliente conheça de antemão a interface do *objeto*. A Figura 2-15 ilustra o funcionamento dos OAs.

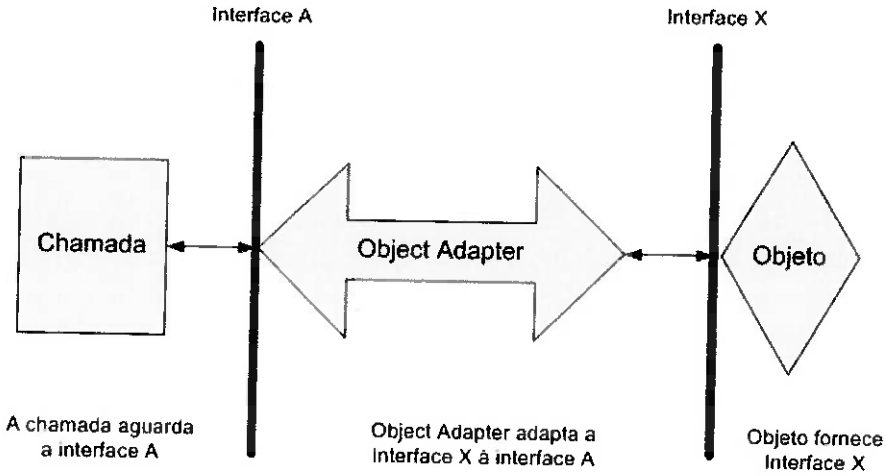


Figura 2-15 - Representação esquemática da funcionalidade dos OAs

Além disso, os OAs apresentam outros aspectos que demonstra a preocupação de se manter o ORB o mais simples possível, como pode ser percebido pelas suas funcionalidades descritas a seguir:

- *Object registration*: OAs geram referências a *objetos* CORBA;
- *Server process activation*: quando necessário, OAs inicializam processos no servidor nos quais *objetos* podem ser ativados;

- *Object activation*: OAs ativam *objetos* se estes não estiverem previamente ativados quando um *request* chega até eles;
- *Request demultiplexing*: OAs cooperam com o ORB no sentido de garantir que *requests* sejam encaminhados através de múltiplas conexões sem serem bloqueadas indefinidamente em uma conexão;
- *Object upcalls*: OAs entregam os *requests* aos *objetos* CORBA registrados.

Sem os OAs, a capacidade do CORBA lidar com diversos tipos de implementações ficaria comprometida. A falta de um OA significaria que os *objetos* se conectariam diretamente ao ORB para receber *requests*. Se existisse apenas um conjunto limitado de interfaces de *objetos* padronizados, apenas um número limitado de *objetos* poderia ser implementado e utilizado no CORBA. Criar um grande número de interfaces padronizadas iria aumentar desnecessariamente a complexidade e o tamanho do ORB.

Portanto, CORBA permite o uso de vários OAs. Um OA diferente é necessário para cada linguagem de programação. Por exemplo, um *objeto* implementado em C, seria registrado junto ao OA, fornecendo um ponteiro de uma estrutura que descreves seu estado juntamente com um conjunto de ponteiros de funções, correspondentes às operações definidas pelo padrão IDL da OMG. Já para a linguagem C++, teria-se um OA que permitisse que uma implementação de um *objeto* herdasse uma implementação de um *objeto* padronizado, fornecendo a interface de chamada ao *objeto*. Usando-se o OA da linguagem C para um *objeto* implementado em C++ ou vice-versa, seria algo "estranho" para programadores de ambas as linguagens.

Apesar das especificações CORBA determinarem que seja permitida a utilização de diferentes OAs, estas só definem um tipo de OA, o *Basic Object Adapter* (BOA). Quando especificado pela primeira vez, esperava-se que o BOA seria suficientemente robusto para atender à maioria de implementações de *objetos* e que outros OAs teriam apenas papéis

secundários para tratarem de casos especiais. O que não foi previsto pelos projetistas do BOA é que OAs costumam ter uma ligação forte com uma linguagem de programação. Com isto, para atingir o objetivo de tornar o BOA um OA compatível com diversas linguagens de programação, sua especificação precisou ser vaga e genérica em certos pontos, como por exemplo, na forma de determinar o mecanismo utilizado para registrar *objetos* de uma linguagem de programação para tornarem-se *objetos* CORBA. Isto gerou problemas de portabilidade entre diferentes implementações do BOA, pois os ORBs desenvolvidos por diferentes empresas tiveram que "preencher" as partes não especificadas de forma vaga com soluções proprietárias.

A OMG logo reconheceu este problema e a partir disso desenvolveu uma nova especificação de OA denominada *Portable Object Adapter* (POA). O POA é compatível com diferentes linguagens de programação, o que fez com que o problema de portabilidade seja resolvido. Outras melhorias foram adicionadas ao POA, como por exemplo, a forma utilizada para gerar sincronismo entre *objetos*, que fornece um gerenciador baseado em uma máquina de estados com quatro estados, permitindo que mais de um grupo de *objetos* seja controlado simultaneamente.

2.5.2.8 Inter-ORB Protocols

Nas versões anteriores ao CORBA 2.0, o maior problema em relação aos ORBs era a falta de interoperabilidade que apresentavam. Isto era causado pelo fato das especificações CORBA nesta época não definirem nenhum formato particular para comunicação com o ORB. O motivo principal pelo qual CORBA não especificava nenhum protocolo ORB antes da versão 2.0, era que a interoperabilidade não era um *objetivo* importante a ser alcançado.

CORBA 2.0 introduziu uma arquitetura para o ORB que fornece uma interoperabilidade direta de ORB para ORB e para interoperabilidade baseada em "pontes".

Interoperabilidade direta é possível quando ORBs residem dentro do mesmo domínio – em outras palavras, eles entendem as mesmas referências a *objetos*, o mesmo tipo de IDL e talvez compartilhem das mesmas informações de segurança. Interoperabilidade baseada em "pontes" é necessária quando ORBs de domínios distintos precisam se comunicar. O objetivo das "pontes" é mapear informações específicas de um ORB de um domínio para outro.

A arquitetura de interoperabilidade ORB é baseada no *General Inter-ORB Protocol* (GIOP), que especifica a sintaxe e o padrão de formato das mensagens para operações entre ORBs para qualquer tipo de conexão e protocolo de transporte. GIOP é projetada para ser fácil de implementar e permitir uma alta escalabilidade e performance.

O *Internet Inter-Orb Protcol* (IIOP) especifica como o GIOP é criado para funcionar com conexões do tipo TCP/IP. De certa forma, a relação entre IIOP e GIOP é parecida com a relação entre uma interface IDL de um *objeto* e sua implementação. GIOP especifica um protocolo, assim como as IDLs da OMG definem o protocolo entre um *objeto* e seu cliente. Por outro lado, o IIOP determina como o GIOP pode ser implementado para trabalhar com TCP/IP, assim como uma implementação de um *objeto* determina como sua interface deve ser criada. Para o CORBA 2.0, é obrigatório que exista compatibilidade com o GIOP e IIOP.

A arquitetura ORB de interoperabilidade também fornece outros *environment-sepecific inter-ORB protocols* (ESIOPs). ESIOPs, permitem que ORBs sejam criados para situações especiais nas quais uma infra-estrutura para o processamento distribuído já é utilizado. O primeiro ESIOP, que utiliza *Distributed Computing Environment*[5] (DCE), é chamada de *DCE Inter-ORB Protocol* (DCE-CIOP) e pode ser utilizado por ORBs em ambientes nos quais DCE já é empregado. Isso permite que o ORB utilize funções DCE pré-existentes, assim como

---

[5] DCE: norma criada e suportada por várias empresas de software que usa RPC para criar um ambiente heterogéneo de computação com um conjunto predefinido de serviços incluindo RPC, serviço de nomes, gestão de processos, segurança, etc.

permite uma integração simplificada do CORBA com programas DCE. O suporte a DCE-CIOP ou qualquer outro ESIOP na versão 2.0 do CORBA ORB é opcional.

Além dos protocolos padronizados de interoperabilidade, referências padronizadas a *objetos* também são necessárias para a interoperabilidade entre ORBs.

Apesar das referências a *objetos* serem "opacas" aos programas, os ORBs utilizam as informações das referências a *objetos* para determinar como fazer um *request* a um *objeto*. Em CORBA, existe uma especificação que padroniza as referências a *objetos* chamada *Interoperable Object Reference* (IOR). O IOR armazena as informações necessárias para fazer a localização e comunicação com um *objeto* utilizando-se diversos protocolos. Por exemplo, um IOR contém informações sobre IIOP, armazena nomes de *hosts* e números de *ports* TCP/IP.

A maioria dos ORBs disponíveis no mercado contém suporte ao IIOP e IOR e foram testados para garantir que haja interoperabilidade. Estes testes podem ser feitos diretamente testando-se a interoperabilidade com outros ORBs disponíveis no mercado, ou então com um sistema desenvolvido especificamente para isto pela OMG chamado CORBAnet, que pode ser utilizado interativamente através de um *Web browser* em Corbanet (2005).

2.6. Web Services

Um *Web Service* (WS) pode ser definido como sendo uma interface que descreve um conjunto de operações acessíveis através de trocas de mensagens padronizadas por meio de uma rede de comunicação. Esta interface é descrita em um documento chamado *service description*, que contém todos os detalhes necessários para a interação com um *serviço* disponibilizado. Isso inclui o formato das mensagens (que detalham a operação), protocolos de transferência de informações e a localização de *serviços*. A interface oculta os detalhes da implementação de um *serviço*, o que permite que esta seja feita independentemente do sistema

de *hardware* e *software* utilizado e também independa da linguagem de programação em que é escrita. Pode-se dizer ainda que um WS é a implementação de uma funcionalidade definida por uma interface (Farrel, Lublinsky, 2002).

Uma outra definição, dada pela IBM para WS é: "WS são um novo tipo de aplicativos *Web*. Este aplicativo funciona de forma independente, é auto-descritivo e tem uma arquitetura modular que pode ser publicada, encontrada e invocada através da *Web*. WS executam funções, que podem ser simples pedidos de informação até complicados processos de negócios. Uma vez que um WS é implementado, o aplicativo (e outros WS) pode encontrar e invocar o *serviço*".

Ainda de outra forma, pode-se dizer que os WS conectam computadores e dispositivos que utilizam a mesma rede de comunicação (como a Internet, por exemplo), para trocarem dados e combiná-los de uma forma conveniente pré-determinada. WS podem ser definidos como *objetos* de um *software* que são agrupados através de uma rede utilizando-se protocolos padronizados que executam funções e processos de negócios. A principal característica de WS é a criação "*on-the-fly*[6]" de *softwares* através do emprego de componentes *loose coupled*[7] que podem ser reutilizados, (Fensel, Bussler; 2002).

Com um WS pode-se, portanto criar rotinas e disponibilizá-las para outros programas que possam ter interesse em executá-las. Para que estas rotinas sejam acessadas, deve-se saber de antemão sua localização, ou então, deve-se dispor de um mecanismo que faz uma procura para localizar onde os *serviços* desejados encontram-se disponíveis. Uma vez localizado um

---

[6] On-the-fly: alteração de configurações sem que seja necessário desligar o sistema ou alterar o seu modo de operação.

[7] *Loose coupled* é a união entre WS sem que exista qualquer tipo de incompatibilidades, mesmo quando usarem tecnologias de sistemas incompatíveis. Sistemas loose coupled podem ser acoplados quando requisitados para criar serviços compostos ou desvinculados, de uma forma simples. Para tanto, os participantes neste processo devem compartilhar de um *framework* com uma semântica pré-estabelecida, para garantir que mensagens trocadas pelos participantes mantenham as informações trocadas consistentes (Loosely Coupled website).

serviço, deve-se enviar uma mensagem ao nó[8] da rede que contém sua implementação. Esta mensagem deve seguir uma formatação padronizada pré-determinada, e contém informações de qual rotina deseja ser executada, assim como informações sobre os parâmetros necessários para tanto.

2.6.1 Server Oriented Architecture

Desenvolver um programa que oferece e acessa *serviços* através da *Web* é uma tarefa relativamente simples. Porém, o fato de um programa utilizar *serviços*, não implica que este esteja seguindo o padrão *Service Oriented Architecture* (SOA). Existe uma grande diferença entre um aplicativo que usa um *serviço* e um aplicativo que é baseado em SOA.

SOA é um modelo conceitual detalhado de como projetar a lógica de aplicativos na forma de *serviços* através de um protocolo de comunicação. Quando se utiliza WS para estabelecer a comunicação deste *framework*, isto representa uma implementação *Web* da arquitetura SOA. A arquitetura SOA estabelece um paradigma dentro do qual WS é um componente chave. Isto significa dizer que quando se deseja migrar ou projetar um aplicativo de acordo com a arquitetura SOA, está se assumindo que o desenvolvimento de WS será uma parte fundamental do ambiente deste aplicativo.

Arquiteturas baseadas em *serviço* representam uma evolução da arquitetura baseada em componentes, que são focadas na comunicação entre processos. O CORBA é baseado neste tipo de arquitetura. A principal diferença entre essas duas abordagens é que os *serviços* podem ser oferecidos e acessados apenas pela interpretação de suas interfaces, enquanto que o acesso a *serviços* oferecidos segundo a arquitetura de componentes necessita da

---

[8]Nó: entende-se por nó de rede um computador conectado à rede de comunicação, apto a trocar informações com outros computadores.

implementação de um elemento adicional que é dependente da plataforma utilizada para fazer a ligação entre o *serviço* e seu requisitante (no caso do CORBA este dispositivo é o ORB).

Uma arquitetura SOA com WS baseados em *Extensible Markup Language* (XML) expõe a lógica do *serviço* oferecido para que possam ser acessados por terceiros, criando uma interface *loose coupled* entre o requisitante e o provedor do *serviço*. Para este tipo de modelo, o SOA fornece um mecanismo para a localização de *serviços* que são oferecidos na *Web*.

Não serão apresentados detalhes do modelo SOA, pois o conhecimento aprofundado desta arquitetura não compromete o entendimento dos métodos utilizados neste projeto. Maiores informações sobre a arquitetura SOA podem ser obtidas no *SOA Center* (2005). A Figura 2-16 mostra de forma simplificada os blocos construtivos do modelo orientado a *serviços*.

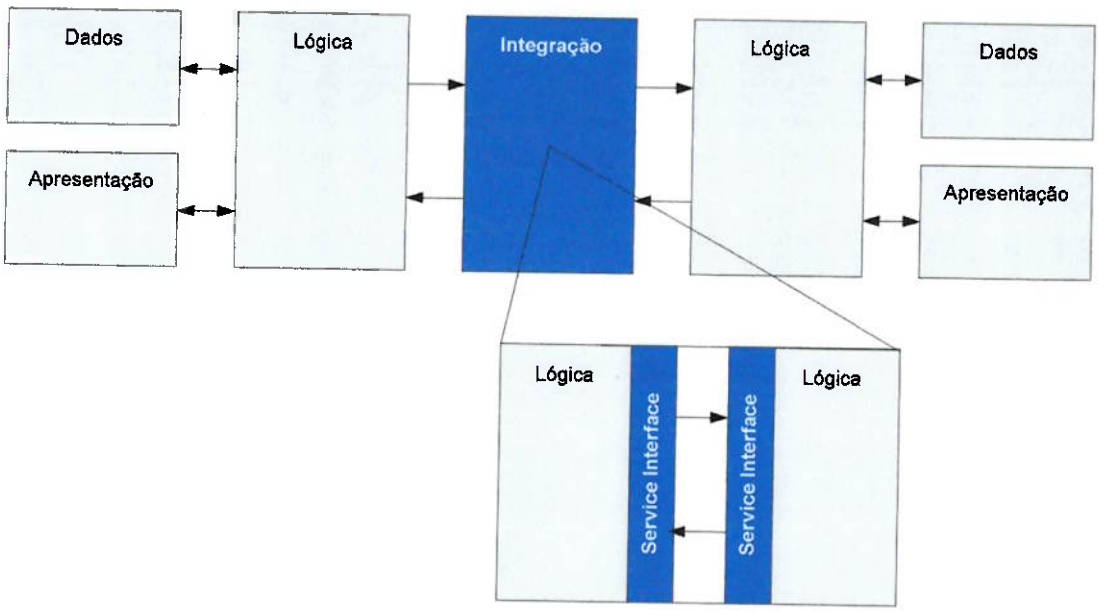


Figura 2-16 - Estrutura da arquitetura orientada a serviços

2.6.2 WS roles

Um aplicativo que estiver utilizando ou disponibilizando *serviços* pode desempenhar diferentes papéis de acordo com o cenário de interação no qual está envolvido. Dependendo do contexto em que o *serviço* é analisado, assim como o estado da tarefa em execução, o

mesmo WS pode mudar de papel (*role*), ou pode desempenhar múltiplos *roles* simultaneamente.

- *Service Provider:* um WS desempenhando o papel de provedor de *serviço* (*provider*) expõe uma interface pública com a qual pode ser invocado por requisitantes do *serviço* (*requestors*). Esta interface é disponibilizada publicando-se uma descrição do *serviço*. Em um modelo cliente-servidor, o *provider* pode ser comparado a um servidor. O termo "*service provider*" pode ser usado também para descrever a organização ou o ambiente que está disponibilizando o *serviço*. Um *service provider* pode agir também como um *requestor*. Por exemplo, um WS pode desempenhar o papel de *service provider* quando um *requestor* pede que uma função seja desempenhada por ele. Em seguida, pode atuar como um *requestor*, quando contata o *serviço* que primeiramente fez a requisição (que neste instante atua como *provider*), para pedir informações de status.
- *Service Requestor*: um *service requestor* é um WS ou o programa que envia uma mensagem requisitando um WS específico. O *service requestor* pode ser comparado a um cliente fazendo-se a analogia ao modelo cliente-servidor. Em certas ocasiões, um *service requestor* pode atuar como *service provider*. Por exemplo, em uma situação que segue o padrão pergunta-resposta, o WS que inicia o processo atua como *service requestor*, ao requisitar informações de um *service provider*. O mesmo WS desempenha o papel de *service provider* quando estiver respondendo à requisição original.
- *Intermediary:* o papel de *intermediary* é desempenhado pelo WS quando este recebe uma mensagem de um *service requestor* e encaminha esta mensagem a um *service provider*. Durante o processo de recepção e encaminhamento de uma mensagem, o *intermediary* desempenha também os papéis de *service provider* e

service requestor, respectivamente. Os *intermediaries* podem existir de diversas formas. Alguns são passivos e apenas "roteiam" a mensagem para o próximo nó, enquanto outros processam a mensagem antes de passá-la adiante. Normalmente, os *intermediaries* estão autorizados apenas a executar o cabeçalho da mensagem.

- *Initial Sender:* os *initial senders* são responsáveis por iniciarem a transmissão de uma mensagem e, portanto podem ser considerados *requestors*. O termo *inicial sender* serve para diferenciar o primeiro WS a enviar uma mensagem de *intermediaries* que também podem atuar como *service requestors*.
- *Ultimate Receiver:* o último WS a receber uma mensagem é chamado de *ultimate receiver*. Estes serviços representam o destinatário final de uma mensagem, e podem ser considerados também *service providers*.

2.6.3 Interação entre WS

Quando mensagens são trocadas entre dois ou mais WS, uma série de cenários de interação pode ocorrer. A seguir é dada a definição de alguns destes cenários, seus nomes e como podem ser identificados.

- *Message path:* A rota pela qual uma mensagem é encaminhada é chamada *message path*. O *message path* deve conter um *initial sender*, um *ultimate receiver* e pode conter zero ou mais *intermediaries*. O caminho de transmissão realizado por uma mensagem pode ser determinado dinamicamente por *intermediaries* de roteamento. A lógica de roteamento pode ser invocada em resposta à necessidade de manter um balanceamento de carga de processamento, ou pode ser baseado em características da mensagem ou outras variáveis lidas e processadas por *intermediaries*.

- *Activity*: Os padrões de mensagens formam a base para as *activities* (também conhecidas como *tasks*). Uma *activity* consiste em um grupo de WS que interagem e colaboram para executar uma função ou um grupo lógico de funções. A diferença entre uma *choreography* e uma *activity* está no fato de que uma *activity* geralmente está associada a uma função específica de um programa como, por exemplo, o processamento de uma tarefa de negócios.

2.6.4 Estrutura da descrição de WS

Como mencionado anteriormente, existem diversas formas de implementar WS e devido à crescente aceitação do modelo orientado a serviços, torna-se cada vez mais comum utilizar WS baseados em XML. Um WS baseado em XML é descrito através de uma série de documentos com definições que constituem a descrição do *serviço*. A Figura 2-18 mostra a relação entre os diferentes tipos de documentos que contém estas definições descritas a seguir.

Segundo a definição dada por Erl (2004), documentos de definições são como blocos construtivos da descrição de um *serviço* onde:

Abstract + *Concrete* = *Service Definition*

Service Definition + *Supplementary Definitions* = *Service Description*

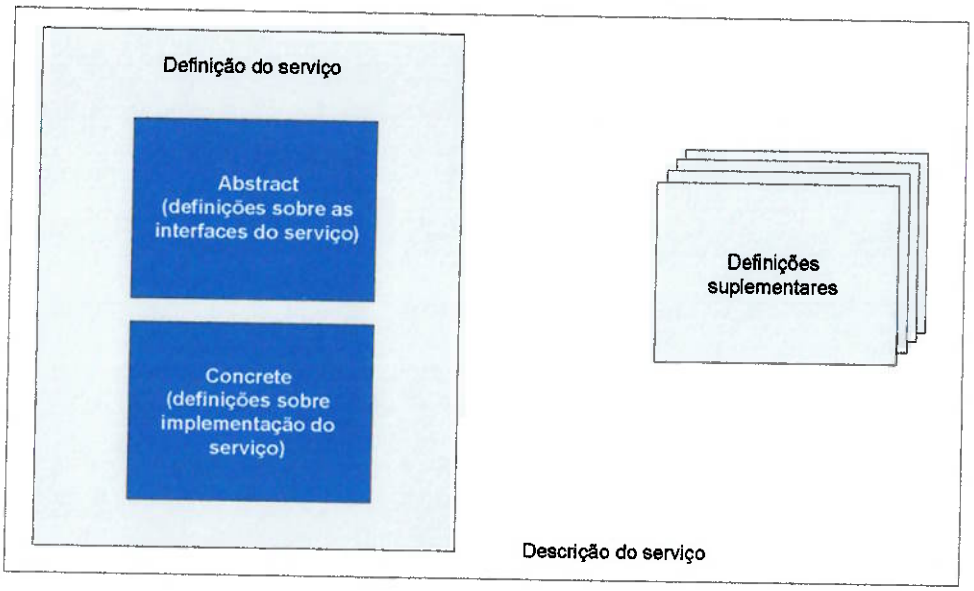


Figura 2-18 - Conjunto de documentos que descrevem um *serviço*

- *Abstract*: a descrição de uma interface de um WS independente dos detalhes de sua implementação, é chamada *abstract*. Em um documento *Web Service Description Language* (WSDL), esta definição da interface *abstract* é feita através de `interface construct` e `message construct`. Estes são criados com a ajuda de um `types construct` declarado separadamente. Estes elementos são descritos em mais detalhes no tópico que trata de documentos WSDL. Em uma arquitetura baseada em componentes, a interface do *serviço* é frequentemente comparada a uma IDL. O termo "*abstract*" substituiu o termo "*service interface definition*" de acordo com as especificações da arquitetura de WS publicadas pela W3C (2003).
- *Concrete*: Informações mais específicas sobre a localização e implementação de um WS podem ser encontradas na parte *concrete* de um documento WSDL, representados pelos elementos `binding`, `service` e `endpoint` (ou `port`).

- *Service Definition*: Geralmente, o conteúdo de um documento WSDL constitui a *service definition* que inclui as definições de interface (*abstract*) e implementação (*concrete*).
- *Service Description*: Muitas vezes, a *service description* consiste de apenas um documento WSDL que fornece uma *service definition*. No entanto, existem casos em que uma *service description* pode conter uma série de documentos adicionais além da *service definition* com definições suplementares (como por exemplo, como o *serviço* se relaciona com outros *serviços*).

2.6.5 WS de primeira geração

Na medida em que a utilização de WS foi se expandindo para o desenvolvimento de programas *Web*, surgiu a necessidade de se criar uma especificação que padronizasse este tipo de arquitetura de sistema, para que *serviços* oferecidos por diferentes desenvolvedores tivessem a capacidade de interoperar. A *World Wide Web Consorsium* (W3C) surgiu desta necessidade. A W3C é de um consórcio internacional onde as organizações membro, uma equipe interna e o público trabalham juntos para elaborar padrões para tecnologias relacionadas ao desenvolvimento de aplicativos para a *Web*. Dentre os padrões elaborados pela W3C encontram-se as especificações de WS, SOAP e WSDL, descritas mais adiante. Daqui em diante, qualquer menção feita a WS, refere-se a *serviços* web baseados na arquitetura e padrões elaborados pela W3C, a menos que seja explicitado que este não é o caso.

As especificações da W3C para WS baseiam-se em documentos e mensagens com formato XML para criar interfaces com recursos distribuídos em uma rede. O protocolo utilizado para a troca de informações é chamado *Simple Object Access Protocol* (SOAP) tratado em um tópico específico mais adiante. Não existem restrições quanto ao tipo de

protocolo de transporte que deve ser utilizado para a troca de informações entre nós da rede nas especificações da W3C, porém, a forma mais comum de se fazer requisições de *serviços* é utilizando o protocolo HTTP.

2.6.6 Componentes de um WS

Existem três especificações que formam a "espinha dorsal" do *framework* para desenvolver, disponibilizar e fazer uso de WS. A primeira delas é o SOAP. O SOAP é um protocolo baseado em XML e serve como um mecanismo de troca de informações em um ambiente descentralizado e distribuído. SOAP permite que exista uma interface entre aplicativos que independa da plataforma de programação, sistema operacional e do tipo de dados que um programa utiliza. Pelo fato de WS normalmente serem implementados empregando-se o protocolo HTTP como base para a troca de informações, normalmente eles podem ser incorporados facilmente à plataforma de uma empresa fazendo-se poucas alterações à infra-estrutura pré-existente, sendo que na maioria dos casos nem ao menos é necessário modificar configurações de *firewalls*.

As outras duas tecnologias necessárias aos WS são *Universal Description, Discovery, and Intergration* UDDI e o WSDL. De forma parecida aos *name services* e *directory services* do CORBA, UDDI é um mecanismo de busca que localiza onde e quem está disponibilizando um *serviço* específico que precisa ser acessado. O WSDL especifica a interface destes *serviços*, quais dados devem ser fornecidos, e quais são retornados. Desta forma, tem-se que o SOAP, UDDI e WSDL, acoplados ao princípio de modelagem orientado a *serviços*, formam uma SOA básica estruturada em XML. A Figura 2-19 mostra o relacionamento entre estes três padrões tecnológicos. Utilizando estes conceitos, sistemas de diferentes domínios, ambientes independentes ou arquiteturas diferentes, podem operar de uma forma cooperativa para desempenhar *serviços*. SOAP, UDDI e WSDL podem ser implementados através de diferentes

protocolos de internet como, por exemplo, HTTP, FTP e SMTP. A Figura 2-20 mostra os diferentes níveis de protocolos utilizados pelos WS e onde ficam alocadas as camadas que contém a implementação do SOAP, UDDI e WSDL.

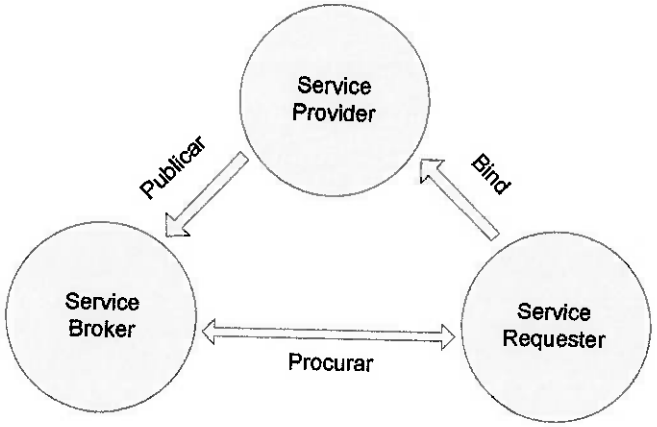


Figura 2-19 - Relacionamento entre UDDI, SOAP e WSDL

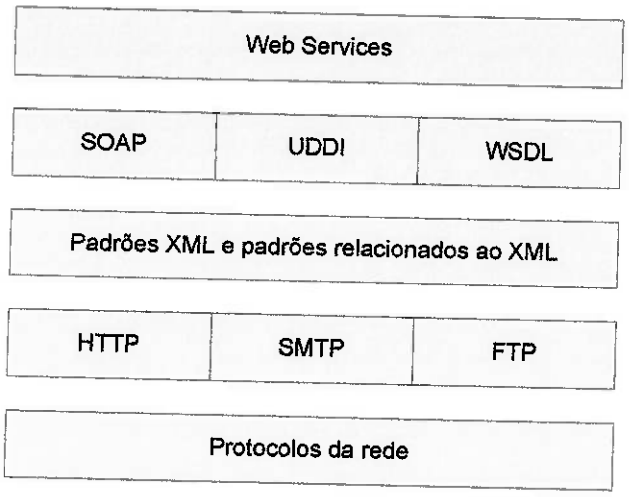


Figura 2-20 - Níveis de protocola na estrutura de um WS

WS tem uma série de vantagens sobre outros métodos utilizados para desenvolver processamento distribuído com alta interoperabilidade, mesmo sendo uma abordagem relativamente nova que ainda está em fase de adoção e sendo "experimentada" pelo mercado. Existem algumas características que separam os WS dos métodos desenvolvidos anteriormente e fazem com que este tenha maior chance de ser bem-sucedido:

- é um padrão apoiado pelos principais desenvolvedores de programas como a Microsoft, IBM e SUN. Nenhuma outra tecnologia de processamento distribuído oferece suporte e compatibilidade tão amplos por empresas;
- o processamento de WS é *loose coupled*. As tentativas anteriores ao WS de atingir interoperabilidade utilizavam um ambiente comum para os aplicativos nos dois nós da transação. WS permite que o cliente e o servidor de *serviços* adotem uma metodologia que elimina este ambiente intermediário;
- usando XML, WS dispõe de um modelo flexível para a troca de dados independente da plataforma que está utilizando os *serviços*;
- utilização de protocolos que são padrões da Internet, o que significa que a maioria das organizações já dispõe de muitos softwares de comunicação e a infra-estrutura necessária para adotar WS. Alguns poucos protocolos novos precisam ser incorporados e linguagens de desenvolvimento pré-existentes podem ser usadas para isto.

As desvantagens de se utilizar WS podem ser divididas em duas categorias. Primeiramente, WS não é um método amplamente testado. Existem suspeitas de que WS é apenas "uma moda passageira" e, como muitos outros métodos empregados para realizar processamento distribuído, não será uma solução suficientemente prática e eficiente para tornar-se um padrão definitivo para este tipo de aplicação. Esta é uma questão que apenas o tempo irá responder, porém as vantagens oferecidas pelo WS lhe dão boas chances de sucesso.

A segunda desvantagem apontada em relação aos WS é a grande dependência do padrão XML. Apesar de existirem muitas vantagens em se utilizar XML, o tamanho das mensagens trocadas entre nós não é uma delas. O uso de XML amplia vertiginosamente o volume de dados que necessita ser armazenado, transportado e processado. Criar ligações de

processamento entre domínios requer uma representação flexível. Esta flexibilidade depende de uma quantidade maior de informações que precisam ser especificadas, o que aumenta o tempo necessário para acessar e processar um *serviço*. Por outro lado, a W3C recentemente criou um grupo de trabalho para desenvolver uma representação "binária" de documentos XML em uma conexão, o que pode melhorar o desempenho na transmissão de dados. A Figura 2-21 mostra mais uma vez a estrutura da arquitetura SOA e identifica onde as diferentes tecnologias são empregadas neste modelo, inclusive como esta pode interagir com outras tecnologias.

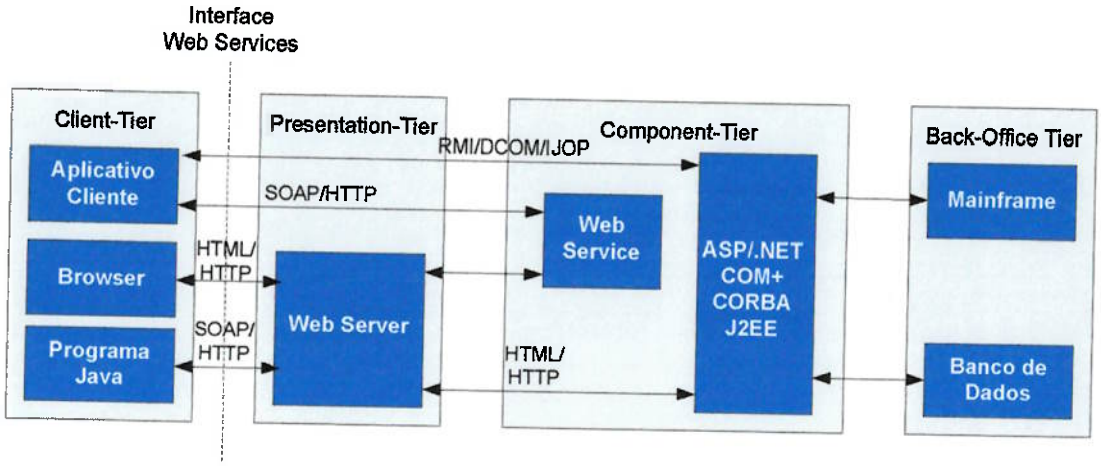


Figura 2-21 - Arquitetura de aplicativos para a web e tecnologias utilizadas na comunicação

2.6.7 Web Service Description Language (WSDL)

WS precisam ser definidos de forma consistente para que possam ser encontrados e interfaceados com outros *serviços* e aplicativos. O WSDL é uma especificação da W3C que fornece a linguagem utilizada para descrever definições de *serviços* (*service definitions*).

A camada de integração introduzida no *framework* dos WS estabelece uma interface de programação padronizada, universalmente aceita. Como mostrado na Figura 2-21, WSDL permite a troca de informação entre estas camadas, pelo fato de conter descrições padronizadas dos dois extremos da conexão.

A melhor forma de entender como um WS é expresso e definido por um documento WSDL, é analisando cada um de seus elementos que coletivamente representam sua definição.

O primeiro elemento (ou *construct*) a ser apresentado, chama-se `definitions`, e pode ser comparado a um envelope, dentro do qual se encontram todas as definições do *serviço*. O documento XML da Figura 2-22 mostra um exemplo de uma definição de *serviço*, expressa pelo elemento `definitions`.

```
<definitions>
<interface name="Catalog">
...
</interface>
<message name="BookInfo">
...
</message>
<service>
...
</service>
<binding name="Binding1">
...
</binding>
</definitions>
```

Figura 2-22 - Definição de serviço expresso pelo elemento service

A definição de um WSDL pode conter uma coleção dos seguintes *constructs*:

- `interface`
- `message`
- `service`
- `binding`

A Figura 2-23 mostra como os dois primeiros *constructs* representam a definição da interface do *serviço* e os dois últimos fornecem os detalhes de sua implementação.

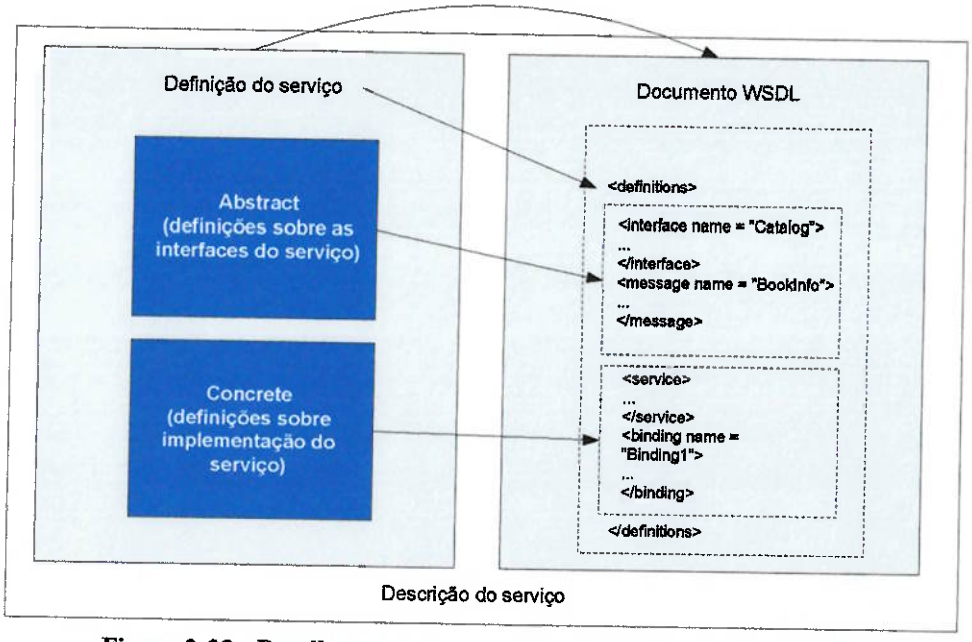


Figura 2-23 - Detalhamento da estrutura de um documento WSDL

2.6.7.1 Abstract Interface definition

As interfaces individuais de um WS são representadas pelos elementos `interface`. Estes `constructs` contêm um grupo de operações lógicas relacionadas. Em uma arquitetura baseada em componentes, uma interface WSDL pode ser comparada a uma interface de um componente. Uma operação é, portanto, equivalente a um método de um componente, pois representa uma ação ou função única (Figura 2-24).

```
<definitions>
  <interface name="Catalog">
    <operation name="GetBook">
      ...
    </operation>
  </interface >
</definitions>
```

Figura 2-24 - Definição dos elementos de uma operação

Um elemento `operation` típico consiste de um grupo de mensagens de entrada e saída relacionadas. A execução de uma operação requer a transmissão ou troca destas mensagens entre o *service requestor* e o *service provider*.

Mensagens de operação são representadas pelo `construct message` que são declarados separadamente abaixo do elemento `definitions`. O nome das mensagens são referenciados nos sub-elementos de entrada e saída das operações (Figura 2-25).

```
<definitions>
  <message name="BookInfo">
    ...
  </message>
  <interface name="Catalog">
    <operation name="GetBook">
      <input name="Msg1" message="BookInfo" />
    </operation>
  </interface>
</definitions>
```

Figura 2-25 - Entradas e saídas de uma operação em documentos WSDL

```
<definitions>
  <message name="BookInfo">
    <part name="title" type="xs:string">
      Field Guide
    </part>
    <part name="author" type="xs:string">
      Mr. T
    </part>
  </message>
</definitions>
```

Figura 2-26 - Exemplo de utilização do elemento "part "

Um elemento `message` pode conter um ou mais parâmetros de entrada e saída pertencentes a uma operação. Cada elemento do tipo `part` define um destes parâmetros, fornecendo a ele um conjunto com um nome, valor e tipo de dados. Em uma arquitetura baseada em componentes, um documento WSDL `part` é equivalente a um parâmetro de entrada ou saída (ou um valor de retorno) de métodos utilizados em componentes (Figura 2-26).

A seguir é apresentado um resumo dos principais `constructs` que podem ser utilizados para criar uma definição de interface *abstract*:

- `interfaces` representam interfaces de *serviços*, e podem conter múltiplos `operations`;
- `operations` representam funções de um WS e podem fazer referência a múltiplas mensagens;
- `messages` representam conjuntos de parâmetros de entrada e saída, e podem conter múltiplos `parts`;
- `parts` podem representar dados de parâmetros de `operations` de entrada ou saída.

2.6.7.2 Definição da implementação – Concrete

Como os elementos descritos nos tópicos anteriores, um documento WSDL pode estabelecer detalhes da implementação de *bindings* para protocolos como, por exemplo, o SOAP e HTTP. Dentro de um documento WSDL, o elemento *service* representa um ou mais nós onde um WS pode ser acessado. Estes elementos contêm informações sobre o protocolo de acesso e a localização do nó e são armazenados em um conjunto de elementos do tipo `endpoint` (Figura 2-27).

```
<definitions>
  <service name="Service1">
    <endpoint name="Endpoint1" binding="Binding1">
      ...concrete implementation details...
    </endpoint>
  </service>
</definitions>
```

Figura 2-27 - Exemplo de utilização do elemento "endpoint"

Uma vez que a descrição de como acessar um WS foi feita, podem-se definir os requisitos para invocar cada uma de suas operações. O elemento `binding` associa o formato do protocolo e dos dados de mensagens necessárias para acessar uma operação (Figura 2-28). O `construct operation` que reside dentro dos blocos `binding` tem sua definição alocada dentro dos elementos interface.

```
<definitions>
  <service>
    <binding name="Binding1">
      <operation>
        <input name="Msg1" message="book" />
      </operation>
    </binding>
  </service>
</definitions>
```

Figura 2-28 - Exemplo de utilização do elemento "binding"

A descrição das definições *concrete* dentro de um documento WSDL pode ser resumida da seguinte forma:

- `service` contém conjuntos de definições de nós representados individualmente pelos elementos `endpoint`;
- `bindings` são elementos associados aos *constructs* do tipo `operation`;
- `endpoints` são elementos de referência dentro dos *constructs* `binding` e portanto relacionam as informações de um nó às operações oferecidas por ele.

2.6.7.3 Constructs adicionais

Uma forma adicional de fornecer suporte a tipos de dados para definições de WS é o elemento `types`. Este *construct* permite que *XSD schemas*[9] possam ser embarcados ou importados para dentro do documento de definição (Figura 2-29).

```
<definitions>
  <types>
    <xsd:schema
      targetNamespace="http://www.examples.ws/"
      xmlns="http://www.w3.org/2000/10/XMLSchema">
      ...
    </xsd:schema>
  </types>
</definitions>
```

Figura 2-29 - Elemento XSD embarcado no documento de definição

Por fim, um elemento opcional chamado `documentation` permite que informações suplementares sejam adicionadas à definição (Figura 2-30).

```
<definitions>
  <documentation>
    I wrote this service definition some time ago,
    when I was younger and times were simpler for us all...
  </documentation>
</definitions>
```

Figura 2-30 - Inserção do elemento "documentation" nas definições

2.6.8 SOAP

SOAP é um protocolo unidirecional, baseado em XML e utilizado para transmitir informações. Apesar de sua unidirecionalidade, mensagens SOAP podem ser combinadas de

---

[9] XSD: Uma forma de descrever e validar em uma ambiente XML. Um *schema* é um modelo de estrutura das informações.

tal forma que possa haver uma interação do tipo *request-response*, ou até mesmo formas mais sofisticadas de diálogo.

Além dos nós da rede que recebem e enviam mensagens SOAP, o roteamento das mensagens é feito pela passagem por nós intermediários. Mensagens SOAP podem ser roteadas através de um nó intermediário para chegar ao destinatário como descrito no tópico WS *roles*. Os nós intermediários em SOAP não devem ser confundidos com os nós intermediários de protocolos que estejam sendo utilizados em uma camada inferior ao SOAP como, por exemplo, o TCP/IP.

SOAP descreve uma linguagem baseada em XML para troca de dados estruturados e de tipos. Como é feito em muitos protocolos, o SOAP é composto de duas partes: uma parte com a descrição das mensagens que são enviadas, incluindo o formato e regras de representação, e uma segunda com a seqüência de mensagens trocadas. O SOAP é uma especificação do esqueleto do formato das mensagens. Este tipo de especificação permite que mensagens sejam moldadas para usos específicos em aplicações. Além do protocolo em si, existem *bindings* (ligações) que descrevem como uma mensagem SOAP pode ser transportada utilizando diferentes tipos de protocolos de transporte. Atualmente, HTTP e SMTP são os únicos protocolos que tem especificações de *binding* elaborados pela W3C, porém outros *bindings* podem ser desenvolvidos e utilizados por terceiros.

2.6.8.1 Processando mensagens SOAP

Os dois principais nós que irão processar uma mensagem SOAP, são: o nó que enviou a mensagem e o nó a quem a mensagem é destinada. Além destes dois nós, existem nós intermediários, que recebem as mensagens e em um outro instante as encaminham em direção ao receptor final. Estes nós intermediários também desempenham um papel no processamento de mensagens SOAP.

Diferentemente do corpo da mensagem, o cabeçalho pode ser:

- explicitamente direcionado a um receptor único e concreto através de uma URI[10];
- direcionado a um receptor baseado na sua posição relativa na corrente de processamento;
- direcionado usando-se algum critério definido pelo programa.

O criador do cabeçalho pode especificar se o receptor deve processar o cabeçalho ou se o cabeçalho pode ser ignorado. Se existir algum requisito necessário para entender algum elemento da mensagem, o receptor precisa parar o processamento da mensagem e enviar uma mensagem SOAP de falha. Isto é útil para que se possa ter certeza de que informações de segurança que precisam ser processadas de forma segura cheguem e sejam interpretadas pelo receptor final de forma correta.

2.6.8.2 Ordem de processamento

O processamento de mensagens SOAP precisa ser feito na seguinte ordem. Primeiramente, o receptor precisa decidir qual sua função. Ele será o receptor intermediário ou o receptor final? O receptor poderá consultar informações no cabeçalho ou corpo da mensagem para tomar esta decisão.

Em seguida, o nó que receber a mensagem deve identificar elementos do cabeçalho destinados a ele e deve entender e decidir se deve ou não processar estes elementos. Caso o nó não seja capaz de processar os elementos, todo o processo deve parar e uma mensagem de falha deve ser gerada.

Se todos os elementos do cabeçalho que são obrigatórios podem ser processados, o nó deve processá-los e no caso do receptor final, processar o corpo da mensagem. O nó pode

---

[10] URI: Endereço codificado por um idenfificador Universal de Recursos (URI), para recursos disponíveis na web

escolher por ignorar um elemento do cabeçalho que não seja de processamento obrigatório. Outros tipos de falhas devem ser gerados também nesta fase.

Finalmente, se o receptor for um nó intermediário, este deve remover os elementos do cabeçalho destinados unicamente a ele e então encaminhar a mensagem ao próximo receptor.

Neste ponto, algumas questões que não foram explicadas podem surgir:

- Como um receptor sabe o que deve fazer com a mensagem? O destinatário de uma mensagem é sempre o próximo receptor, mas é também o receptor final?
- Como um receptor decide qual ordem irá utilizar para processar os cabeçalhos?
- Como um nó determina quem será o próximo receptor para que a mensagem possa ser roteada até ele?

Estas são perguntas pertinentes, porém o SOAP não tem respostas para elas. Estas decisões devem ser tomadas usando-se algum algoritmo embutido dentro do programa, ou então determinado por algum outro método que está fora do escopo do SOAP.

Uma vez que estas decisões foram tomadas, instruções que refletem as respostas geradas podem ser incorporadas nos cabeçalhos da mensagem. Por exemplo, o criador da mensagem pode incluir informações sobre o roteamento e instruções mais detalhadas sobre como processar as instruções do cabeçalho. Outra abordagem seria cada nó incluir instruções para o receptor seguinte.

2.6.8.3 Formato das Mensagens

A forma básica mínima de uma mensagem SOAP é mostrada no documento XML da Figura 2-31. Uma codificação usando apenas tipos incorporados no protocolo SOAP e sem definições adicionais é recomendada nas especificações. Este esquema mínimo para uma mensagem SOAP permite que ela seja validada sem necessitar de documentos XML *shema*. No entanto, XML *shemas* específicos para dadas aplicações são permitidos, que necessitam de

validações adicionais. Cada mensagem SOAP é identificada como um documento XML 1.0 que contém um elemento com o nome do envelope. Este elemento é qualificado com o *namespace* http://www.w3c.org/2002/06/soap-envelope. Além de identificar o *namespace* como sendo um *namespace* SOAP, o URL identifica a versão do SOAP que está sendo utilizada. O envelope contém sub-elementos de um cabeçalho (opcional) e de um corpo (obrigatório) que será descrito mais adiante.

Além do que foi discutido, não é preciso mais nenhum elemento dentro do envelope SOAP que identifica o tipo de mensagem. Não há necessidade de se incluir a identidade do nó que criou e enviou a mensagem, tampouco do receptor, da data de criação ou titulo da mensagem. Espera-se que cada programa defina estes elementos se forem necessários. Com exceção das especificações quanto à formatação, o receptor determina por ele mesmo como interpretar o conteúdo da mensagem. Espera-se que o receptor faça isso, utilizando e entendendo o *namespace* que associa elementos e atributos com o aplicativo implementado pelo receptor.

```
<? Xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-
envelope">
<env:Header>
</env:Header>
<env:Body>
</env:Body>
</env:Envelope>
```

Figura 2-31 - Mensagem SOAP

2.6.8.4 O cabeçalho de mensagens SOAP

O cabeçalho de uma mensagem SOAP mostrado na Figura 2-32 que é uma versão modificada da mensagem da Figura 2-32 é uma parte opcional de uma mensagem SOAP.

Seu nome local é *header* e é identificado usando o mesmo *namespace* que o envelope. O cabeçalho pode conter zero ou mais identificadores de *namespaces* de sub-elementos. Dois

outros atributos, `role` e `mustUnderstand`, podem ser associados com sub-elementos do cabeçalho. No exemplo, `hd r1` é identificado no namespace www.widgets.com/logging/.

```
<? Xml version='1.0' ?>
<env:Envelope xmlns:env=
"http://www.w3.org/2003/05/soap-envelope">
<env:Header>
<sec:hdr1 xmlns:sec="http://www.widgets.com/logging"
sec:actor=
"http://www.w3.org/2003/05/soap-envelope/role/next"
sec:mustUnderstand="true">
</sec:hd r1>
</env:Header>
<env:Body>
</env:Body>
</env:Envelope>
```

Figura 2-32 - Mensagem SOAP com cabeçalho

Diferentemente do corpo da mensagem, que não pode ser alterado, o cabeçalho é uma parte dinâmica da mensagem. Nós intermediários às vezes precisam apagar elementos do cabeçalho que são destinados unicamente a eles e podem inserir novos elementos se necessário.

Cada elemento do cabeçalho será processado no máximo por um WS. No entanto, outros poderão examinar estes elementos, mesmo que não sejam destinados a eles. Caso o WS seja um intermediário, este deve apagar os elementos do cabeçalho destinados a ele e pode adicionar outros elementos para receptores subseqüentes antes de repassar a mensagem.

2.6.8.5 O corpo de mensagens SOAP

É natural que se pense no SOAP como sendo um protocolo do tipo *request-response*, porém não há necessidade de se retornar uma responsta para uma dada mensagem que foi recebida. Ainda assim, sub-elementos do corpo da mensagem foram definidos e são a conseqüência lógica de certas entradas. Por causa disto, a discussão sobre o corpo da

mensagem será dividida entre elementos do corpo da mensagem do tipo *request* e *response*. O SOAP trata a comunicação em cada sentido como eventos separados e não-relacionados.

O corpo da mensagem de um *request* feito em SOAP pode conter um ou mais sub-elementos. Caso existirem múltiplos sub-elementos, eles podem representar uma única unidade de trabalho, múltiplas unidades de trabalho ou então uma combinação de trabalho e dados. Elementos de mensagens são análogos a documentos em papel que contém uma estrutura de acordo com o conteúdo descrito, como por exemplo, pedidos de compra, itinerários, receitas, etc...

O conteúdo de uma mensagem de resposta pode ser um documento, uma resposta RPC, ou uma falha SOAP. Assim como um documento pode ser recebido, um documento pode resultar de uma receita de um documento.

Uma mensagem de resposta SOAP, pode conter uma falha SOAP. Os únicos sub-elementos que são definidos pelas especificações SOAP, são as falhas, que são geradas em resposta a erros e podem conter também outras informações de status. Sub-elementos do tipo `code` e `reason` são necessários dentro do elemento de falha. Outros dois sub-elementos, `node` e `role` e detalhes são opcionais. `Code` é uma estrutura que contém um valor que designa a falha em um alto nível e um sub-elemento opcional chamado `subcode` que informa sobre mais detalhes da falha. `Node` identifica o nó SOAP que identificou a falha. `Role` identifica qual a operação que o nó estava processando quando a falha ocorreu. Finalmente, `detail` contém informações específicas da aplicação sobre a falha.

2.6.8.6 Caracterísiticas do SOAP

A chave para o sucesso do SOAP está na habilidade de se estender e adicionar funcionalidades a ele. As características do SOAP são abstratas e referem-se à troca de

mensagens entre nós. Estas funcionalidades podem incluir confiabilidade, garantia de entrega e segurança.

Quando uma funcionalidade é implementada em um nó SOAP, esta é implementada modificando-se o modelo de processamento SOAP. Se a funcionalidade afetar a interação entre dois nós consecutivos, a funcionalidade é implementada como parte do SOAP *binding*. Uma limitação de um protocolo *binding* é que ele relaciona dois nós conectados por uma única transmissão. Transmissões de uma ponta a outra, podem ser implementadas usando-se diferentes protocolos, necessitando de múltiplas transmissões. Nestes casos, a funcionalidade deve ser expressa em blocos de cabeçalho SOAP e implementadas no modelo de processamento.

Funcionalidades são expressas como módulos alocados em cabeçalhos SOAP. Um módulo, ou *Message Exchange Pattern* (MEP) é um *template*, definido nas especificações SOAP (W3C, 2005), usado para descrever a troca de mensagens entre nós SOAP. A parte principal de se especificar um *binding* é descrever como ou protocolo é utilizado para implementar MEPs que este deve saber interpretar. Dois MEPs, *request-response* e *response*, já foram definidos. O MEP *request-response* é exatamente o que se espera. O MEP de *response* é o envio de uma resposta SOAP depois de receber um *request* não SOAP.

2.6.8.7 HTTP Binding

Existe uma especificação para o *binding* de HTTP (W3C, 2002). A não ser que seja informado explicitamente, SOAP através de HTTP é transmitido usando este *binding*. O *binding* permite aos MEPs de *request-response* e *response* sejam utilizados e especifica como o HTTP deve lidar com mensagens para implementar padrões. Para o MEP *request-response*, mensagens com *requests* são enviadas utilizando-se chamadas do tipo `HTTP POST`. O `HTTP URI` identifica o nó de destino assim como o programa que recebe a mensagem. A resposta

correspondente é enviada utilizando-se HTTP response, fornecendo um meio natural de correlacionar a resposta com o pedido.

No *binding* HTTP, a mensagem de resposta SOAP é enviada em resposta a um *request* HTTP. Se o MEP for *request-response*, a mensagem SOAP de *request* foi enviada através de uma chamada `HTTP POST`. Se o MEP for de resposta, o *request* foi transmitido como uma chamada `HTTP GET`. O *binding* HTTP apenas entende este tipo de MEPs para realizar *requests*. Quando utilizado desta forma, a interação será indistinguível de um envio convencional de informações através de HTTP. O MEP de resposta pode apenas ser utilizado quando não existem intermediários entre o primeiro nó que enviou a mensagem e o nó receptor final. A informação retornada é identificada apenas pelo URL, pois não há um envelope SOAP para transmitir informações ao provedor de *serviços*.

2.6.9 UDDI

Um dos componentes fundamentais da arquitetura orientada a *serviços* é o mecanismo utilizado por potenciais requisitantes para encontrar descrições de WS. Para estabelecer esta parte do *framework* de um WS, um diretório central para hospedagem de descrições de *serviços* é necessário. Este diretório pode tornar-se parte integral de uma organização ou comunidade na Internet e é considerada uma extensão da infra-estrutura.

É por isso que as especificações UDDI estão se tornando cada vez mais importantes. Um elemento chave do UDDI é a padronização de registros de perfis armazenados neste diretório, também conhecidos como *registry*. Diferentes implementações do registro podem ser criadas, dependendo do objetivo do *serviço* e para qual tipo de aplicação é intencionado.

A partir da versão 3 do UDDI, este tornou-se um padrão OASIS (2004). As especificações do UDDI são baseadas em tecnologias desenvolvidas pela W3C e da *Internet Engineering Task Force* como o XML, SOAP e *Domain Name Services*. (OASIS UDDI).

Um *public business registry* é um diretório global com descrições de *serviços* de negócios internacionais. Instâncias deste registro são hospedadas por grandes corporações (também chamadas de *node operators*) em uma série de servidores UDDI dedicados. Registros UDDI são replicados automaticamente entre instâncias de repositório. Algumas empresas também atuam como registradores de UDDI, permitindo que terceiros adicionem ou editem suas descrições de WS. O *public business registry* é complementado por um grande número de *service marketplaces* que oferecem WS genéricos.

Private registrys são repositórios de descrições de *serviço* hospedados dentro de uma organização. Normalmente, quem tem acesso a estes diretórios são parceiros de negócios. Um registro a usuários internos é chamado de *internal registry*.

Os registros de um UDDI são organizados utilizando-se seis tipos básicos de dados:

- *business entities*: oferecem informações sobre o *serviço* registrado, incluindo o nome, a descrição e um identificador único, são representados pelo elemento `businessEntity`.;
- *business services*: representam o *serviço* oferecido pelo WS registrado. O elemento de descrição neste caso é o `businessServices`;
- *specific pointers*: As informações oferecidas pelos *Specific pointers* são utilizadas para mapear a implementação do *serviço* através de um endereço. Com esta informação, um desenvolvedor pode aprender a conectar-se ao *serviço*;
- *service types*: Registro UDDI fornece um meio de apontar para definições de *serviços* através de um *tModel*. O *tModel* é um documento XML que representa a definição do *service type* UDDI e também pode conter informações sobre formatos de mensagens, assim como protocolos de mensagens e de segurança;
- *business relationship*: representados pelo elemento `publisherAssertion`, permitem estabelecer uma relação entre dois ou mais `businessEntitys`;

- *subscriptions*: Possibilitam informar aos associados do *registry* quando alguma informação sobre um cadastro de *serviço* foi atualizada.

O documento XML apresentado na Figura 2-33 contém um registro de *serviço* UDDI. Pode-se identificar todos os elementos descritos acima encapsulados dentro do elemento raiz `businessEntity`. Este arquivo é um exemplo extraído de Erl (2004).

Apesar de terem sido apresentados os fundamentos do UDDI (com foco da estrutura de entidades de serviços), a discussão não foi aprofundada até o cerne do UDDI registry: o tModel. Este componente disponibiliza o acesso aos detalhes técnicos necessários para que requisitores possam interagir com WS através de uma interface. Maiores informações referentes ao tModel, podem ser encontradas nas especificações do UDDI (OASIS, UDDI, 2005b).

```
<businessEntity xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  businessKey="e9355d51-32ca-49cf-8eb4-1ce59afbf4a7"
  operator="Microsoft Corporation"
  authorizedName="Thomas Erl"
  xmlns="urn:uddi-org:api_v2">
  <discoveryURLs>
    <discoveryURL useType=
      "businessEntity">http://test.uddi.microsoft.com/discovery
      ?businesskey=e9355d51-32ca-49cf-8eb4-1ce59afbf4a7
    </discoveryURL>
  </discoveryURLs>
  <name xml:lang="en">
    XMLTC Consulting Inc
  </name>
  <description xml:lang="en">
    XMLTC has been building end-to-end enterprise
    eBusiness solutions for corporations and
    government agencies since 1996. We offer a
    wide range of design, development and
    integration services
  </description>
  <businessServices>
    <businessService
      serviceKey="1eeecfa1-6f99-460e-a392-8328d38b763a"
      businessKey="e9355d51-32ca-49cf-8eb4-1ce59afbf4a7">
      <name xml:lang="en-us">
        Corporate Home Page
      </name>
      <bindingTemplates>
        <bindingTemplate
          bindingKey="48b02d40-0312-4293-a7f5-4449ca190984"
          serviceKey="1eeecfa1-6f99-460e-a392-8328d38b763a">
          <description xml:lang="en">
            Entry point into the XMLTC Web site
            through which a number of resource
            sites can be accessed
          </description>
          <accessPoint URLType="http">
            http://www.xmltc.com/
          </accessPoint>
          <tModelInstanceDetails />
        </bindingTemplate>
      </bindingTemplates>
      <categoryBag>
        <keyedReference
          tModelKey="uuid:c1acf26d-9672-4404-9d70-39b756e62ab4"
          keyName="Namespace" keyValue="namespace" />
      </categoryBag>
    </businessService>
  </businessServices>
</businessEntity>
```

Figura 2-33 - Estrutura de um registro UDDI

3. Mecanismos de Integração e coordenação da Plataforma de Simulação

Pelo fato da *Plataforma de Simulação* em si ser um sistema distribuído e baseada na interação entre módulos que contém informações sobre os modelos elaborados, esta requer que mecanismos de integração e gerenciamento da troca de informações entre módulos executados em computadores remotamente sejam criados. Os métodos que foram tratados no capítulo anterior como tecnologias de integração para aplicativos distribuídos, CORBA e WS, são especialmente adequados para esta tarefa, oferecendo a interoperabilidade e escalabilidade necessárias à *Plataforma de Simulação*. Tanto o CORBA quanto os WS encapsulam funcionalidades desejadas em arquivos com programas ou bibliotecas que podem ser publicados e acessados por aplicativos externos para executarem algum *serviço* desejado. A aplicação deste conceito para a *Plataforma de Simulação* consiste em fazer o encapsulamento dos módulos nos formatos especificados por estas tecnologias e assim publicá-los em uma rede como *serviços* ou *objetos* CORBA, fazendo com que possa ocorrer um processamento distribuído entre diversos computadores. Portanto, tem-se modelos elaborados em RdP que podem ser convertidos em *serviços* ou *objetos* CORBA que interagem entre si.

Para que os módulos convertidos possam processar uma simulação segundo o formalismo das RdP e respeitar a hierarquia de modelos elaborada para a *Plataforma de Simulação*, torna-se necessário trocar informações de três naturezas entre eles que são:

- informações sobre *transições*;
- informações sobre o avanço do tempo e sinais de controle e monitoramento da simulação;

- informações sobre a hierarquia e permissões de acesso dos módulos.

A seguir é apresentada uma visão mais detalhada das funções e da arquitetura de cada um dos três agrupamentos citados acima.

3.1. Informações de transição

A relação ente os *objetos* elaborados com a *Plataforma de Simulação* se dá através da fusão entre suas *transições*. Desta forma, um *objeto* A pode conter uma *transição* t_{A1}, cujo *lugar* seguinte encontra-se em um outro *objeto* B. Este segundo *objeto*, contém uma *transição* t_{B1} que é ativada pela marcação nos *lugares* do *objeto* A. Na realidade as *transições* t_{A1} e t_{B1} representam a mesma *transição*, porém em *objetos* diferentes. Desta forma, a *transição* t_{A1} está "fundida" com a *transição* t_{B1}. Um exemplo de como se dá a interação entre dois *objetos* é mostrado na Figura 3-1, onde ocorre a fusão das *transições* t_{A1} e t_{B1} e posteriormente a fusão entre t_{B2} e t_{A2}.

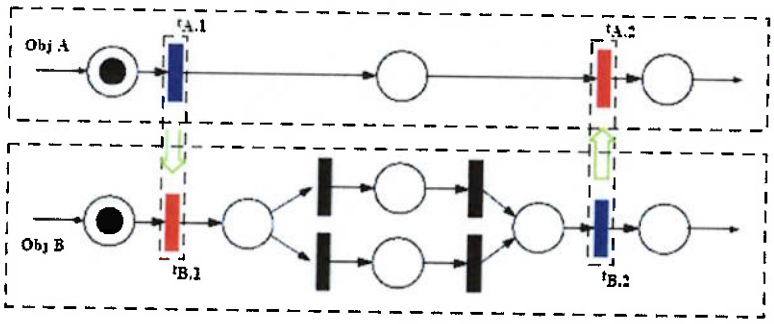


Figura 3-1 - Representação da fusão de *transições* entre dois *objetos* (Junqueira)

Traduzindo esta interação entre *objetos* para a implementação da *Plataforma de Simulação* utilizando CORBA e WS, tem-se que os *objetos* A e B representam implementações de *objetos* CORBA ou *serviços web* que apresentam na sua interface, publicados os métodos t_{A1} e t_{A2} para o *objeto* A e t_{B1} e t_{B2} para o *objeto* B, que podem ser invocados remotamente.

As *transições* são, portanto o elemento que interliga os vários módulos de um modelo elaborado na *Plataforma de Simulação*. É apresentado a seguir como gerenciar este caso para que o processamento de uma simulação possa ocorrer satisfatoriamente.

3.2. Informações sobre o avanço do tempo e sinais de controle e monitoramento da simulação

Partindo-se da abordagem conservativa para o processamento de RdP em um ambiente distribuído, é preciso desenvolver uma forma de sincronizar o progresso da simulação entre os *objetos* que fazem parte de um mesmo modelo. A forma encontrada para este sincronismo entre os *objetos*, é baseada na criação de uma seqüência de conexão entre eles, formando um "anel" lógico por onde constantemente circula um rótulo com dados sobre o instante de tempo da simulação. Desta forma informações de monitoramento e controle da simulação são recebidas e enviadas por todos os *objetos* que participam do processamento de uma mesma simulação de forma cíclica, informando qual o próximo instante de tempo e quando este será adotado, assim como outras informações de erro e mensagens do tipo *broadcast* para todos os módulos. Um esquema da conexão em "anel" entre os *objetos* pode ser visto na Figura 3-2.

O rótulo que circula pelo anel lógico contém 5 campos com as seguintes informações:

- *Station Identity Field*: este campo indica qual foi o último *objeto* a modificar o valor dos outros campos da mensagem;
- *Future time Field*: este campo contém a informação do tempo de simulação requerido pelo *objeto* indicada no campo *Station Identity Field*;
- *Status Field*: este campo indica o status atual o *objeto* indicada no campo *Station Identity Field*. A **Erro! Fonte de referência não encontrada.** mostra os possíveis valores que este campo pode assumir;

- *Instruction Field*: este campo contém informações para todos os *objetos*, como iniciar, parar, e pausa na simulação;
- *Error Field*: este campo é utilizado para o tratamento de erros de simulação, como por exemplo, quando um *objeto* é desconectado devido a um problema de comunicação na rede.

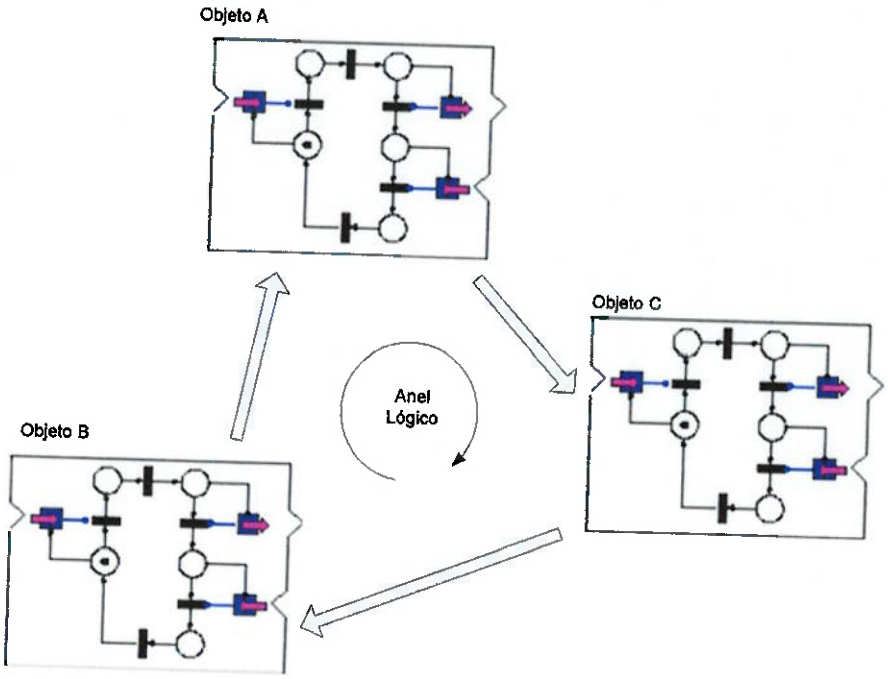


Figura 3-2 – Conexões em "anel" para monitoramento e controle da simulação

Tabela 3-1 - Significado dos valores do campo de status

Valor do campo de Status	Significado
0	Nenhuma estação está utilizando a mensagem.
1	A estação está verificando o status de outras estações
2	A estação está enviando uma ordem para todas as outras estações para atualizarem o valor do tempo de simulação para o valor indicado no campo *future time field*.
3	A estação encontra-se em um estado de *deadlock*.

Para estabelecer o "anel" lógico entre os *objetos*, é necessário que, ao implementar os *objetos* na forma de *objetos* CORBA ou WS, seja incluído neles um componente que

estabeleça esta conexão. Uma das formas de se fazer isso é inserir uma estrutura de *sockets* em cada *objeto*. Os *sockets* são componentes que implementam conexões TCP/IP do tipo cliente-servidor com base nos dados da *porta* e endereço IP do programa ao qual desejam se conectar. Assim, cada componente teria um *client-socket* e um *server-socket* embutido em seu código para estabelecer uma conexão com um *objeto* e receber pedidos de conexão de outros *objetos*. Desta forma, quando o *server-socket* receber a mensagem com as informações de monitoramento e controle descritas acima, o *objeto* CORBA ou WS irá processar as informações contidas nela, alterar suas informações (se necessário), utilizar o *client-socket* para estabelecer uma conexão com outro *objeto*, enviar a mensagem processada, fechar a conexão e aguardar até que a mensagem complete uma volta através do "anel" lógico e chegue novamente através do estabelecimento de uma conexão do *server-socket* requisitada pelo *objeto* que se encontra na posição anterior no "anel" lógico e recomece o ciclo.

Informações sobre o funcionamento de *sockets* e redes TCP/IP podem ser encontradas nos Apêndices A e B.

3.3. Informações sobre a hierarquia e permissões de acesso dos módulos

Ao se iniciar uma simulação, deve-se primeiramente fazer uma busca para encontrar todos os módulos necessários para completar o modelo que será simulado. Para encontrar estes módulos, deve-se analisar as *transições* que interligam os *objetos* da *Plataforma de Simulação*, pois elas são os elementos que unem todo o modelo.

Além disso, como visto na introdução, os *objetos* da *Plataforma de Simulação* têm uma hierarquia baseada em *domínios* e *federações* que deve ser respeitada. Quando um *objeto* é implementado na forma de um *objeto* CORBA ou WS, as informações de *domínio* e *federação* devem ser compiladas juntamente com os dados do modelo. Desta forma, quando o processamento de uma simulação for requisitado, e os *objetos* necessários estiverem sendo

procurados, as permissões de acesso estarão contidas nos próprios *objetos* CORBA ou WS, estabelecendo o tipo de acesso permitido entre diferentes *serviços*.

Para que possam ser encontrados os *objetos* CORBA ou WS, são utilizados os *serviços de diretório* como *object services*, *naming service* e *trading service*, no caso do CORBA e o UDDI na caso dos WS. Estes diretórios podem conter restrições de acesso e privilégios definidos para usuários de um *domínio* e *federação*. Desta forma, pode-se encontrar uma descrição dos modelos gerados na *Plataforma de Simulação* em um diretório virtual da rede. Quando um módulo de interesse é localizado pode-se acessar sua descrição de interface, IDLs no caso de *objetos* CORBA e WSDL no caso de WS. Com estes documentos, torna-se possível acessar o local onde o *serviço* está implementado e interligar os *objetos* da *Plataforma de Simulação* para iniciar o processamento de uma simulação.

Os *objetos* CORBA ou WS, precisam conter um método que faça uma busca por *serviços* que estejam relacionados a eles, ou seja, cada *objeto* precisa encontrar quais são os *objetos* que serão usados na fusão de suas interfaces, através de um comando de busca. O mecanismo de procura e acesso a *objetos* que compõe o modelo com auxílio de *serviços de diretórios* é mostrado na Figura 3-3.

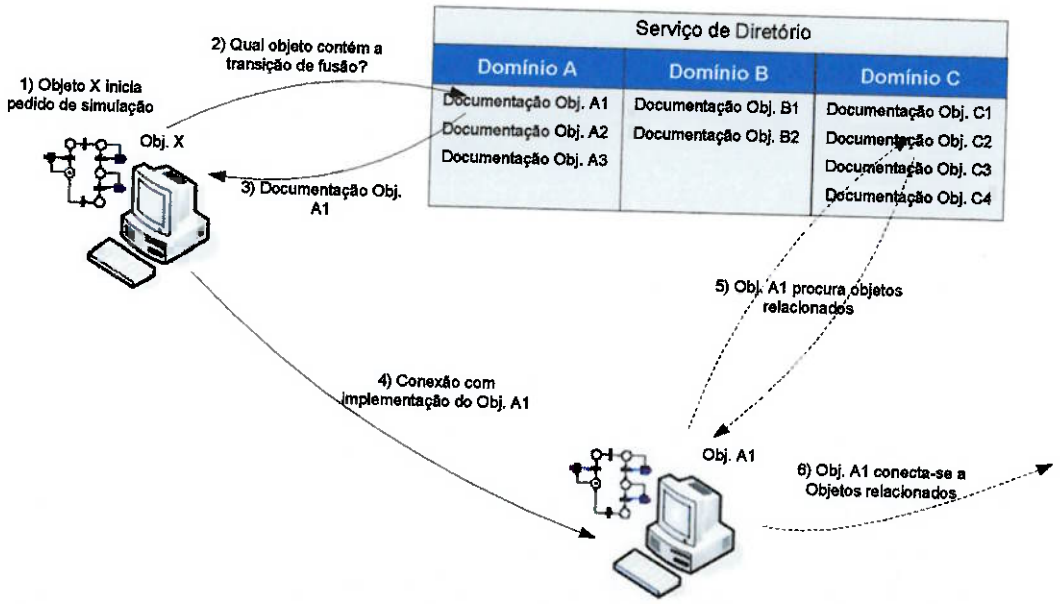


Figura 3-3 - Gerenciamento da hierarquia e acesso a objetos através de serviços de diretórios

3.4. Implementação com WS

A aplicação considerada para implementar com WS, é composta por um cliente que acessa um *serviço*, que por sua vez acessa outro *serviço*. O cliente invoca o *serviço* através da chamada à função `primeira`, cujo nome remete ao fato deste ser o primeiro *serviço* acessado. Esta função recebe um parâmetro de entrada do tipo `string` e retorna também uma `string`. A função `primeiro` é implementada no *serviço* chamado *primeiro*. Quando a função `primeiro` é executada pelo *serviço*, este faz uma invocação a um outro *serviço* chamado *segundo*. A invocação ao *serviço segundo* é feito através da chamada à `segundo`. A função `segundo` também recebe como parâmetro de entrada e retorna uma `string`, sendo que a `string` de entrada consiste da `string` recebida pelo cliente concatenada com a `string` "`O primeiro serviço foi executado.`". Quando o *serviço segundo* recebe esta `string`, adiciona a ela mais um pedaço com o seguinte texto: "`O segundo serviço foi executado`". Após adicionar este texto à `string`, esta é retornada ao *serviço primeiro*, que por sua vez repassa a `string` como resultado da função `primeiro` ao cliente. Quando a função `primeiro` retorna no cliente, o `string` recebido é mostrado na tela.

A programação do teste descrito foi desenvolvido com a plataforma Delphi versão 7, pois esta disponibiliza uma forma relativamente fácil e rápida de se implementar *serviços web*, com interface gráfica e assistentes para a criação de servidores de *serviços* e clientes.

Para que um *serviço* possa ser acessado, é preciso publicá-lo assim como dispor de um *servidor web* para que possa ser acessado. Quando se utiliza o SOAP baseado em HTTP, é preciso ter um *servidor web* sendo executado no computador que ofereça o *serviço*, para que ele possa receber mensagens SOAP no formato HTTP e transmiti-las ao *serviço* que fará seu processamento. Este *servidor web* pode ser programado dentro de um aplicativo com as próprias ferramentas do Delphi, ou então, pode-se utilizar um *servidor web* disponível no mercado. Para testar os *serviços* desenvolvidos neste exemplo, foi utilizado o *servidor web* da Microsoft que está incluído na distribuição do Windows XP, o *Internet Information Server* (IIS) versão 5.1. Outros servidores como o *Apache* poderiam ter sido utilizados também.

Quando se inicia um projeto novo no Delphi, existem alguns *templates* prontos para diferentes tipos de programas. No caso de WS, existem *templates* que criam a estrutura de um WS cliente e servidor. Primeiramente é analisado o cliente.

3.4.1 Programa cliente

O programa cliente consiste de um formulário com uma caixa de texto para o parâmetro que é passado à função `primeiro`, e um campo de texto para mostrar o resultado da função na tela, além de um botão que, ao ser clicado, faz a chamada à função `primeiro` (Figura 3-4).

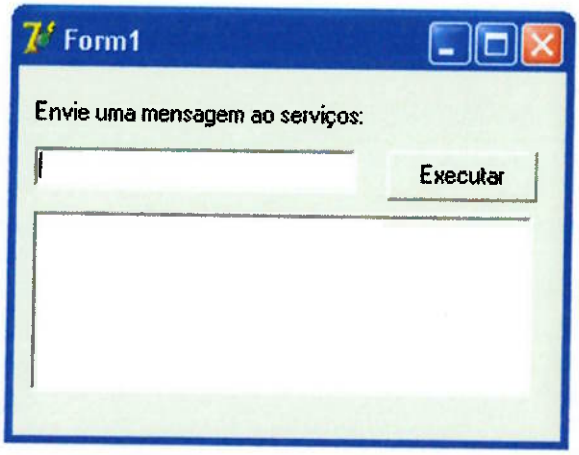


Figura 3-4 - Formulário do programa cliente

Além do formulário, o cliente contém duas `units` (o equivalente ao conceito de `classes` em C++) que implementam a interface com o *serviço primeiro*. A estrutura destas `units` são criadas automaticamente pelo Delphi, através do assistente de importação de especificações de *serviços*. Este assistente cria a estrutura do código necessário para implementar a interface com *serviços* baseado no documento WSDL de um *serviço*. Portanto, através do documento WSDL do *serviço primeiro*, é criada uma `unit` com a descrição da interface com o *serviço*, e uma função que cria um componente capaz de gerar mensagens SOAP que são enviadas ao servidor que está disponibilizando o *serviço primeiro*. O código desta `unit` é mostrado na Figura 3-5 e Figura 3-6:

```
unit Iprimeiro1;

interface

uses InvokeRegistry, SOAPHTTPClient, Types, XSBuiltIns;

type

  Iprimeiro = interface(IInvokable)
  ['{C9911108-2135-4DB6-1EB4-E79637D92694}']
    function  primeiro(const a: WideString): WideString; stdcall;
  end;
```

Figura 3-5 - Implementação da Unit primeiro

```
function GetIprimeiro(UseWSDL: Boolean=System.False; Addr:
string=''; HTTPRIO: THTTPRIO = nil): Iprimeiro;

implementation

function GetIprimeiro(UseWSDL: Boolean; Addr: string; HTTPRIO:
THTTPRIO): Iprimeiro;
const
  defWSDL = 'D:\Caio - documentos\Poli\PMR2550 - Projeto de
Conclusão do Curso II\Projetos\Web
Services\Cliente\1\Iprimeiro.xml';
  defURL  = 'http://localhost/Scripts/ws1.exe/soap/Iprimeiro';
  defSvc  = 'Iprimeiroservice';
  defPrt  = 'IprimeiroPort';
var
  RIO: THTTPRIO;
begin
  Result := nil;
  if (Addr = '') then
  begin
    if UseWSDL then
      Addr := defWSDL
    else
      Addr := defURL;
  end;
  if HTTPRIO = nil then
    RIO := THTTPRIO.Create(nil)
  else
    RIO := HTTPRIO;
  try
    Result := (RIO as Iprimeiro);
    if UseWSDL then
    begin
      RIO.WSDLLocation := Addr;
      RIO.Service := defSvc;
      RIO.Port := defPrt;
    end else
      RIO.URL := Addr;
  finally
    if (Result = nil) and (HTTPRIO = nil) then
      RIO.Free;
  end;
end;

initialization
  InvRegistry.RegisterInterface(TypeInfo(Iprimeiro),
'urn:primeiroIntf-Iprimeiro', 'utf-8');
  InvRegistry.RegisterDefaultSOAPAction(TypeInfo(Iprimeiro),
'urn:primeiroIntf-Iprimeiro#primeiro');

end.
```

Figura 3-6 - implementação da Unit primeiro - continuação

Pode-se identificar a declaração da função `primeiro` na parte da interface desta `unit`. O código presente na parte da implementação serve para criar o componente que irá elaborar a mensagem SOAP.

A função `primeiro` é chamado no evento `OnClick` do botão que se encontra na `unit` que descreve o formulário. O código do evento `OnClick` é mostrado na Figura 3-7.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo1.Lines.Add(GetIprimeiro.primeiro(Edit1.Text));
end;
```

Figura 3-7 - Evento OnClick do botão Executar

A Figura 3-8 mostra o resultado da execução dos *serviços primeiro* e *segundo* depois de apertado o botão `Executar`.

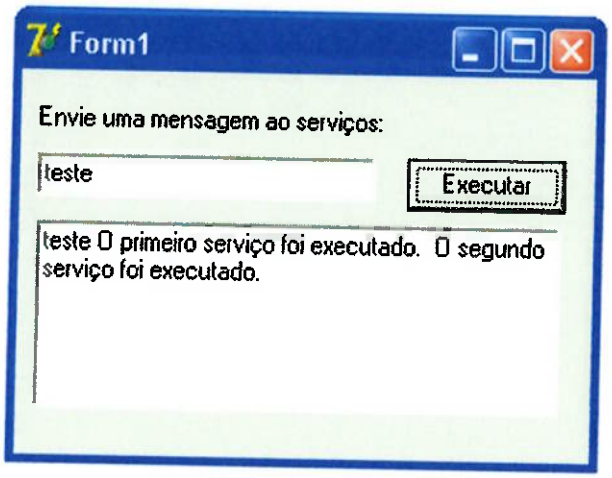


Figura 3-8 - Tela do programa cliente depois que os serviços primeiro e segundo foram executados

3.4.2 O serviço primeiro

O *serviço primeiro* contém a implementação da função `primeiro` chamada pelo programa cliente. A estrutura de um *serviço* pode ser criada a partir de um assistente de servidores de *serviços* no Delphi. Este assistente gera as seguintes `units`:

- `primeiroIntf`: `unit` com a interface do *serviço primeiro*, com o código mostrado na Figura 3-9:

```
{ Invokable interface Iprimeiro }

unit primeiroIntf;

interface

uses InvokeRegistry, Types, XSBuiltIns;

type

  { Invokable interfaces must derive from IInvokable }
  Iprimeiro = interface(IInvokable)
  ['{B549E819-CA6C-44E2-ABDB-79A4991C8354}']
    function primeiro(a: string):string; stdcall;
    { Methods of Invokable interface must not use the default }
    { calling convention; stdcall is recommended }
  end;

implementation

initialization
  { Invokable interfaces must be registered }
  InvRegistry.RegisterInterface(TypeInfo(Iprimeiro));

end.
```

Figura 3-9 - código da unit de interface do serviço primeiro

- `primeiroImpl`: `unit` com a implementação da interface do *serviço primeiro*, que contém o código listado na Figura 3-10. Note que nesta `unit` encontra-se a implementação da função `primeiro` e que esta função faz uma chamada a função `segundo`, implementada no *serviço segundo*.

```
type

{ Tprimeiro }
  Tprimeiro = class(TInvokableClass, Iprimeiro)
  public
    function primeiro(a: string):string; stdcall;
  end;

implementation

uses Isegundo1;

{ Tprimeiro }

function Tprimeiro.primeiro(a: string): string;
begin
  result := GetIsegundo.segundo(a + ' O primeiro serviço foi
executado. ');
end;

initialization
  { Invokable classes must be registered }
  InvRegistry.RegisterInvokableClass(Tprimeiro);

end.
```

Figura 3-10 - código da unit primeiro

- `wm1: unit` com os componentes que recebem mensagens SOAP com pedidos de clientes, invocam a função do *serviço* e elaboram mensagens SOAP com o resultado da execução da função. Os componentes deste módulo podem ser vistos na Figura 3-11. O componente `WSDLHTMLPublish1` que pode ser visto na Figura 3-11 serve para publicar o documento WSDL do *serviço*. Além destas `units`, existe uma quarta `unit` que contém a interface necessária para acessar o *serviço segundo*. Novamente foi utilizado o assistente de importação de WSDL, porém desta vez foi utilizado o documento WSDL do *serviço segundo*.

Figura 3-11 - Unit com componentes utilizados para servidores de WS

3.4.3 Serviço segundo

O serviço *segundo* é implementado da mesma forma que o *serviço primeiro*, através do assistente de criação de servidores de *serviços web*. Este *serviço* contém apenas uma função, a função `segundo` que recebe a `string` do *serviço primeiro* e retorna a `string` alterada.

O código da implementação da função `segundo` pode ser visto na Figura 3-12:

```
unit segundoImpl;

interface

uses InvokeRegistry, Types, XSBuiltIns, segundoIntf;

type

  { Tsegundo }
  Tsegundo = class(TInvokableClass, Isegundo)
  public
    function segundo(b: string):string; stdcall;
  end;

implementation

{ Tsegundo }

function Tsegundo.segundo(b: string): string;
begin
  result := b + ' O segundo serviço foi executado.';     end;

initialization
  { Invokable classes must be registered }
  InvRegistry.RegisterInvokableClass(Tsegundo);

end.
```

Figura 3-12 - Código da implementação da função segundo

A implementação da função `primeiro` e `segundo` são relativamente simples, porém poderiam incluir lógicas complexas que representassem o processamento de RdP por exemplo. Da mesma forma, os parâmetros de entrada e saída das funções `primeiro` e `segundo` são apenas `strings`, porém poderiam ser dados de fusão entre *transições* ativas dos modelos em simulação, ou informações sobre o *domínio* e a *federação* da qual o serviço pertence, no momento em que estiver procurando *serviços* relacionados a ele.

3.5. Implementação com CORBA

O exemplo desenvolvido utilizando a tecnologia CORBA, também programado com o Delphi 7, consiste de dois aplicativos servidores, *servidor1* e *servidor2*, que contém os *objeto CORBA Primeiro* e *Segundo*, respectivamente. Um terceiro programa cliente, acessa o *objeto Primeiro* implementado no *servidor1*. Este *objeto* contém um método chamado `primeiro`, que recebe como parâmetro de entrada uma `string` e retorna também uma `string` como resultado do processamento. Por sua vez, a função `primeiro` do *servidor1* faz uma chamada ao método `segundo` do *objeto segundo* implementado no *servidor2*. Este método, assim como o método `primeiro` do *servidor1*, tem como parâmetros de entrada e saída uma `string`.

O programa cliente contém um formulário com uma caixa de texto na qual o usuário digita uma `string` que será passada como parâmetro de entrada da função `primeiro`. A chamada ao método `primeiro` é feito quando o usuário pressiona o botão *Executar* contido no formulário. Quando a função `primeiro` recebe este parâmetro, esta repassa-o como parâmetro de entrada para o método `segundo`, porém antes disso acrescenta a `string` "O serviço 'primeiro' foi executado" a ela. O método `segundo`, ao receber o parâmetro enviado pelo método `primeiro`, acrescenta a ela a `string` "O serviço 'segundo' foi executado" e a retorna como resultado. Este resultado é recebido pela função `primeiro`, que o repassa como resultado para o programa cliente (Figura 3-13).

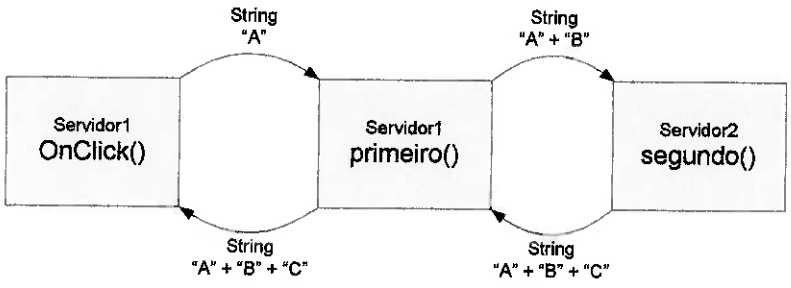


Figura 3-13 - Parâmetros de entrada e saída dos métodos primeiro e segundo

3.5.1 Programa cliente

O programa cliente contém apenas uma `unit` que contém o código do formulário utilizado para que o usuário entre com o parâmetro inicial que é passado à função `primeiro`, fazer sua chamada e mostrar o resultado na tela. O formulário pode ser visto na Figura 3-15.

A função `primeiro` é executada ao se clicar no botão *Executar*. O código contido neste evento pode ser visto na Figura 3-15.

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, CorbaObj,
  StdCtrls, server1_TLB;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Edit1: TEdit;
    Memo1: TMemo;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    InfoServer: IPrimeiro;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  InfoServer := TPrimeiroCorbaFactory.CreateInstance('');
  Memo1.Lines.Add(InfoServer.primeiro(Edit1.Text));
end;

end.
```

Figura 3-14 - Código da unit do formulário do programa cliente

Nota-se que nas declarações de importação (comando `uses`) encontra-se uma `unit` chamada `server1_TLB`. Esta `unit` contém o *stub* do *objeto* CORBA *Primeiro*. O código desta `unit` será visto quando o programa *servidor1* for explicado. No código do evento `OnClick` do botão do formulário pode ser visto que um *objeto* chamado `InfoServer` é criado. Este *objeto* implementa a interface do *objeto Primeiro*. Além da função `primeiro`, a implementação desta `unit` contida no *servidor1*, cria um *factory object*. A função do *factory object* é criar o *objeto Primeiro* para executar a função `primeiro`. A chamada a esta função pode ser vista na segunda linha de código do evento `OnClick`.

Figura 3-15 - Formulário do programa cliente

No programa cliente, a `string` recebida é mostrada no formulário. O resultado de uma execução com o parâmetro inicial "Teste" é mostrado na Figura 3-16.

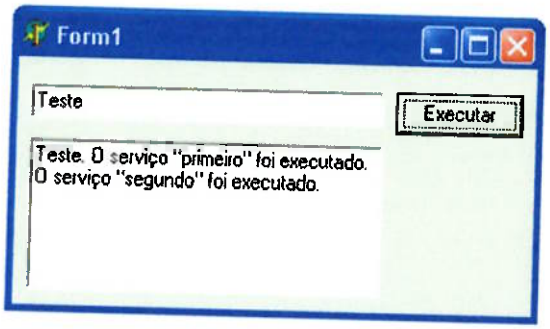


Figura 3-16 - Formulário do programa cliente após a execução dos métodos dos *objetos* CORBA remotos

3.5.2 O programa servidor1

Assim como existe um assistente para criar servidores de WS, o Delphi 7 possui um assistente para criar *objetos* CORBA. Este assistente é utilizado para implementar o *objeto*

Primeiro no aplicativo servidor. Foi mencionado que o programa cliente utiliza a `unit servidor1_TLB`, de onde utiliza as definições do *stub* para acessar o *objeto Primeiro*. Na realidade, esta `unit` é criada no `servidor1` pelo assistente e consiste na interface do *objeto* CORBA que se deseja criar. A interface contém também a descrição de um `object factory` que foi utilizado pelo programa cliente para criar o *objeto Primeiro*. Além de fornecer a interface, é através desta `unit` que o Delphi gera um documento IDL para o *objeto*, que pode ser publicado para que outros aplicativos saibam como conectar-se aos serviços oferecidos pelo *objeto*. A `unit servidor1_TLB` gerada pelo assistente contém apenas as declarações necessárias para criar um *objeto*. Cabe ao programador decidir quais propriedades e métodos compõem o *objeto*. Existe um editor no Delphi 7, chamado *Type Library Editor*, pelo qual, pode-se adicionar estes métodos e propriedades, sem que seja necessário escrever o código de programação diretamente.

Utilizando o editor, foi adicionado ao *objeto Primeiro* o método chamado `primeiro`, e seus parâmetros de entrada e saída. Os trechos de código que definem a interface do *objeto*, o *stub* e o *skeleton* estão mostrados na Figura 3-17.

```
//Interface do objeto Primeiro
IPrimeiro = interface(IDispatch)
    ['{88D7532D-D957-4107-B3BD-23CDCF0994D6}']
    function  Primeiro(const Mensagem1: WideString): WideString;
safecall;
  end;
...
//Stub do objeto Primeiro
TPrimeiroStub = class(TCorbaDispatchStub, IPrimeiro)
  public
    function Primeiro(const Mensagem1: WideString): WideString;
safecall;
  end;

//Skeleton do objeto Primeiro
  TPrimeiroSkeleton = class(TCorbaSkeleton)
  private
    FIntf: IPrimeiro;
  public
    constructor Create(const InstanceName: string; const Impl:
IUnknown); override;
    procedure GetImplementation(out Impl: IUnknown); override;
stdcall;
  published
    procedure Primeiro(const InBuf: IMarshalInBuffer; Cookie:
Pointer);
  end;
```

Figura 3-17 - Interface, stub e skeleton do objeto Primeiro

Entre outros elementos, esta `unit` também contém a definição do *object factory* utilizado para criar o *objeto Primeiro*. O Delphi contém um tipo específico para criação destes *objetos* e o código da Figura 3-18 mostra como utilizá-lo.

```
  TPrimeiroCorbaFactory = class
    class function CreateInstance(const InstanceName: string):
IPrimeiro;
  end;
```

Figura 3-18 - Declaração do *object factory* na unit servidor1_TLB

O assistente de criação de *objetos* CORBA cria também uma segunda `unit` onde é implementada a interface do *objeto Primeiro*, chamada impPrimeiro. O único código que precisa ser colocado manualmente nesta `unit`, é a lógica dos métodos que o *objeto* oferece,

neste caso, o único método presente é a função `primeiro`. O código da `unit impPrimeiro`, pode ser visto na Figura 3-19. No final da listagem, encontra-se uma seção denominada `initialization`, que cria o *objeto* quando o aplicativo é inicializado, permitindo que este possa ser invocado por programas clientes.

O código que implementa a função `primeiro` nesta `unit`, contém na primeira linha um comando para criar um *objeto* do tipo `Segundo`. Na parte da declaração, vê-se que o *objeto Segundo* tem uma interface do tipo `ISegundo`, que refere-se ao *objeto Segundo* que se encontra no *servidor2*. Da mesma forma que o programa cliente teve que importar o *stub* do *objeto Primeiro*, o *objeto Primeiro*, dentro da implementação do seu método `primeiro`, utiliza o `stub` do *objeto Segundo* para fazer uma chamada ao método `segundo`. Portanto, pode-se notar na parte de importações (comando `uses`) da `unit impPrimeiro`, a `unit server2_TLB`.

Na segunda linha de implementação da função `primeiro`, encontra-se a chamada ao método `segundo` do *objeto Segundo*.

```
unit impPrimeiro;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
ComObj, StdVcl,
  CorbaObj, server1_TLB, server2_TLB;

type

  TPrimeiro = class(TCorbaImplementation, IPrimeiro)
  private
    { Private declarations }
  public
    { Public declarations }
  protected
    segundo: ISegundo;
    function primeiro(const Mensagem1: WideString): WideString;
safecall;
  end;

implementation

uses CorbInit;

function TPrimeiro.primeiro(const Mensagem1: WideString):
WideString;
begin
  segundo := TSegundoCorbaFactory.CreateInstance('');
  result := segundo.segundo(Mensagem1 + '. O serviço "primeiro"
foi executado.');
end;

initialization
  TCorbaObjectFactory.Create('PrimeiroFactory', 'Primeiro',
'IDL:server1/PrimeiroFactory:1.0', IPrimeiro,
    TPrimeiro, iMultiInstance, tmSingleThread);
end.
```

Figura 3-19 - Código da unit impPrimeiro

O documento IDL gerado pelo Delphi para o *objeto Primeiro* pode ser visto na Figura 0-8.

```
module server1
{
  interface IPrimeiro;


  interface IPrimeiro
  {
    wstring Primeiro(in wstring Mensagem1);
  };

  interface PrimeiroFactory
  {
    IPrimeiro CreateInstance(in string InstanceName);
  };
     };
```

Figura 3-20 - Documento IDL do objeto Primeiro

3.5.3 Programa servidor2

O aplicativo *servidor2* é implementado de forma semelhante ao aplicativo *servidor1* com a ajuda do assistente de criação de *objetos* CORBA, gerando uma `unit` de interface para o *objeto Segundo*, que também contém um *object factory*, o *stub* e o *skeleton*, chamada `servidor2_TLB`. A `unit` de implementação `impSegundo`, contém o código mostrado na Figura 3-21 para implementar a função `segundo` e o IDL mostrado na Figura 3-22.

```
function TSegundo.segundo(const Mensagem2: WideString):
WideString;
begin
  result := Mensagem2 + ' O serviço "segundo" foi executado.';
end;
```

Figura 3-21 -Implementação da função segundo

```
module server2
{
  interface ISegundo;

  interface ISegundo
  {
    wstring Segundo(in wstring Mensagem2);
  };

  interface SegundoFactory
  {
    ISegundo CreateInstance(in string InstanceName);
  };

};
```

Figura 3-22 - Documento IDL do objeto Segundo

Além dos aplicativos desenvolvidos como teste, para que a interação entre os *objetos* distribuídos possa ocorrer, é necessário dispor de um ORB que faça a integração entre eles. A Borland, empresa desenvolvedora do Delphi (entre outras palataformas de programação), criou um ORB para ser utilizado por aplicativos com tecnologia CORBA. O ORB da Borland chama-se *VisiBroker* e a versão utilizada para executar os programas de teste desenvolvidos foi a versão 3.3. Além do ORB, o *VisiBroker* contém alguns *objetos* CORBA para que os programas desenvolvidos possam interoperar de forma correta.

O primeiro *objeto* incluído no *VisiBroker* é o *Smart Agent*. O objetivo do *Smart Agent* é providenciar um *serviço de diretório* para desempenhar a tarefa de encontrar *objetos* CORBA para clientes que requisitam algum *serviço* que se encontra em um aplicativo servidor CORBA (Long, Brian, 2005). Os *Smart Agents* podem ser encontrados pelo ORB através de um *broadcast* pela rede.

O segundo *objeto* incluído no VisiBroker é o *Object Activation Domain.* (OAD) Este *objeto* permite que *objetos* CORBA sejam registrados e assim não precisam estar sendo executados para que possam oferecer um *serviço* a um cliente. Através dos registros contidos no OAD, o ORB pode acessá-lo (o OAD) como se fosse o *objeto* requisitado pelo cliente. Se o OAD tiver um registro do *objeto* desejado, este trata de iniciar o aplicativo que contém sua implementação e redirecionar a conexão do ORB a ele. Quando não existe um OAD ou um objeto não está registrado a nenhum OAD, para que um cliente possa acessá-lo, o aplicativo que contém sua implementação precisa estar sendo executado.

Para que o *VisiBroker* funcione, é imprescindível que o computador esteja ligado a uma rede TCP/IP. A localização de *objetos* só é possível se existir pelo menos um *Smart Agent* sendo executado na rede. É importante lembrar que cada computador que estiver executando programas que utilizem objetos CORBA, precisam também dispor de um ORB e inicia-lo para que possa ocorrer a interoperação entre *objetos.*

Seria possível utilizar outras implementações de ORBs para executar o aplicativo cliente desenvolvido neste projeto, porém optou-se pelo VisiBroker, por ser o mais renomado ORB disponível no mercado e estar incluído na distribuição da plataforma Delphi 7 (versão Enterprise). Na Internet existem vários sites com comentários e *links* para *downloads* de outras implementações do ORB. Uma boa referência para o assunto é o site *The free CORBA page* (2005), que contém *links* para distribuições gratuitas de ORBs entre outros assuntos relacionados ao CORBA.

Poderia-se criar programas que, ao invés de utilizarem *stubs* e *skeletons* para trocar informações entre *objetos*, fizessem invocações dinâmicas (DII) por parte do cliente e acessassem o servidor também de forma dinâmica (DSI). Para que isso seja possível, é preciso manter um registro de IDLs em um IR. O VisiBroker oferece um *serviço* deste tipo, chamado `irep` e portanto, poder-se-ia criar clientes que acessassem as informações sobre a interface do

objeto em tempo de execução. Desta forma, basta que se tenha uma *object reference* que é passada para o IR para acessar o *objeto* remoto.

Da mesma forma que os *serviços* gerados utilizando a arquitetura de WS poderiam representar os modelos em RdP da *Plataforma de Simulação*, os *objetos* CORBA desenvolvidos poderiam exercer esta mesma tarefa. Assim, um modelo completo seria executado através da interação entre *objetos* CORBA através de aplicativos e ORBs. Mais uma vez, o gerenciamento de tempo de simulação poderia ser feito inserindo-se *sockets* na implementação dos *objetos*, e métodos publicados que recebessem e enviassem informações sobre a conexão, para criar o "anel" lógico entre os *objetos*.

4. Análise dos Resultados e Comparação dos Métodos

As implementações apresentadas no capítulo anterior em CORBA e WS não visam apresentar uma solução pronta que possa ser integrada diretamente na *Plataforma de Simulação*, mas sim entender melhor os mecanismos utilizados por cada uma das tecnologias e prover um meio de compará-las qualitativamente.

A seguir, é apresentada uma estrutura de como os WS e o CORBA poderiam ser utilizados para a implementação da *Plataforma de Simulação*. São feitos também alguns comentários apontando as vantagens e desvantagens de cada método.

4.1. Arquitetura da Plataforma de Simulação baseada em WS

A Figura 4-1 mostra como a *Plataforma de Simulação* poderia ser estruturada, caso utilizasse a tecnologia de WS para gerenciar a comunicação entre *serviços*.

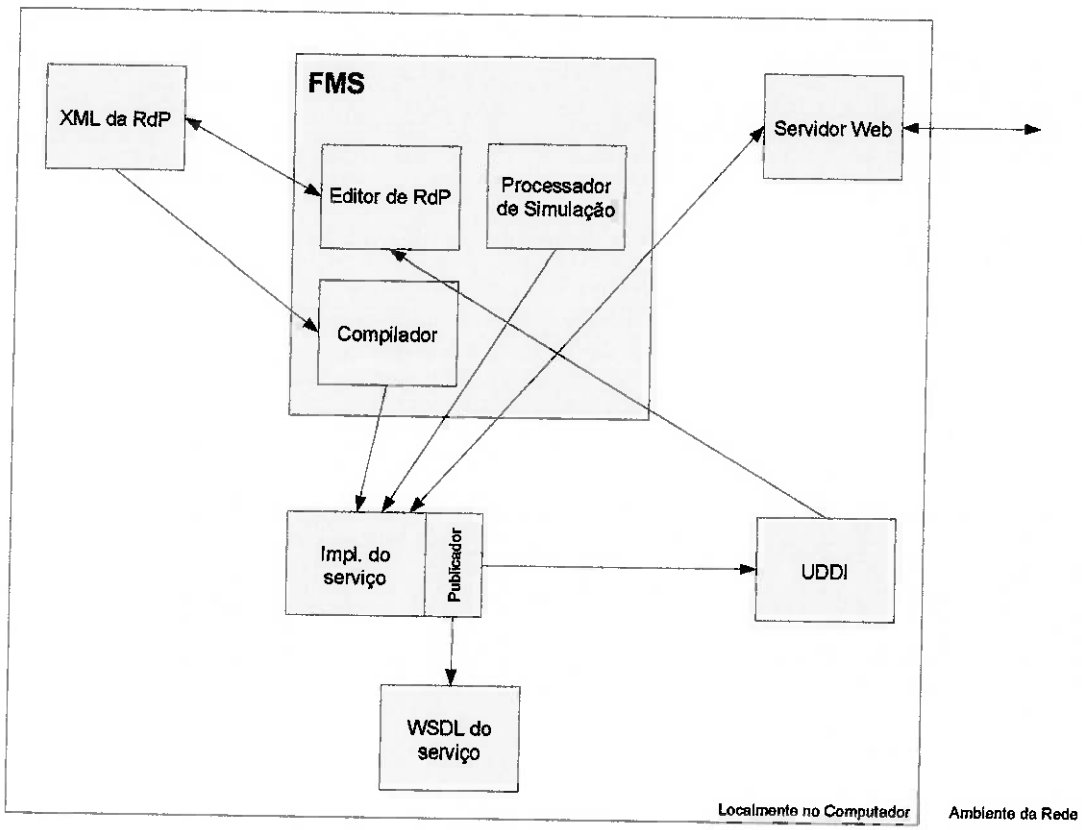


Figura 4-1 - Arquitetura da Plataforma de Simulação baseada em WS

Na estrutura baseada em WS, a FMS contém três elementos básicos que são:

- Editor de RdP: este elemento seria um ambiente com interface para que o usuário criasse modelos baseados em RdP. Estes modelos seriam armazenados na forma de documentos XML como indicado na Figura 4-1.
- Compilador: o compilador, baseado no documento XML gerado pelo editor, gera um *serviço* e o compila para que possa ser acessado por outros clientes. O *publicador* que se encontra dentro do *serviço*, é utilizado para publicar a interface do *serviço* para que terceiros possam encontrá-lo e saber como acessá-lo. O documento WSDL do *serviço* pode conter também o elemento `documentation`, no qual pode ser inserido o arquivo XML com a descrição completa da lógica da RdP gerada, caso seja interessante divulgar esta informação. Os *serviços* gerados pelo compilador devem conter também os *sockets* utilizados para estabelecer a rede

de comunicação em "anel" entre os *serviços* durante o processo de simulação, para que possa haver um controle do processamento e do avanço do tempo de simulação.

- Processador de simulação: este elemento serve para iniciar o processamento de um *serviço*. Ele invoca o primeiro *serviço* que inicia a simulação do modelo. A partir desta inicialização, as interfaces entre os *serviços* fazem com que os *objetos* que contém os modelos interajam através de suas interfaces.

O UDDI oferece um *serviço de diretório*, no qual as informações sobre *serviços* criados possam ser encontrados e utilizados pelas pessoas com acesso às suas informações. Quando um desenvolvedor pretende utilizar um *serviço* pré-existente para interoperar com um modelo que esteja criando, este deve conhecer sua interface e, portanto o seu documento WSDL.

Para que *serviços* remotos possam acessar *serviços* disponibilizados localmente, é preciso fornecer um servidor para o protocolo HTTP. Este servidor pode ser um servidor já existente no mercado, como o IIS (que foi utilizado para executar os *serviços* de teste desenvolvidos neste projeto), o Apache ou algum outro. Estes servidores precisam conter suporte a extensões de *serviços*. No caso de WS isto significa que os servidores precisam saber reconhecer uma mensagem com cabeçalho SOAP e encaminha-la ao *serviço* correto. Além disso, pode-se desenvolver um servidor dedicado a tarefa de disponibilizar os *serviços* na rede. Criar um servidor HTTP e fazê-lo funcionar com SOAP é uma tarefa relativamente simples e pode ser feita no Delphi com algumas poucas linhas de código. Dusyukov (2005), explica como é possível criar servidores no Delphi com os componentes Indy, que tenham suporte ao SOAP.

4.2. Modelo de arquitetura para a Plataforma de Simulação baseada em CORBA

A Figura 4-2, mostra como a *Plataforma de Simulação* poderia ser estruturada, caso utilizasse a tecnologia CORBA para gerenciar a comunicação entre *objetos*.

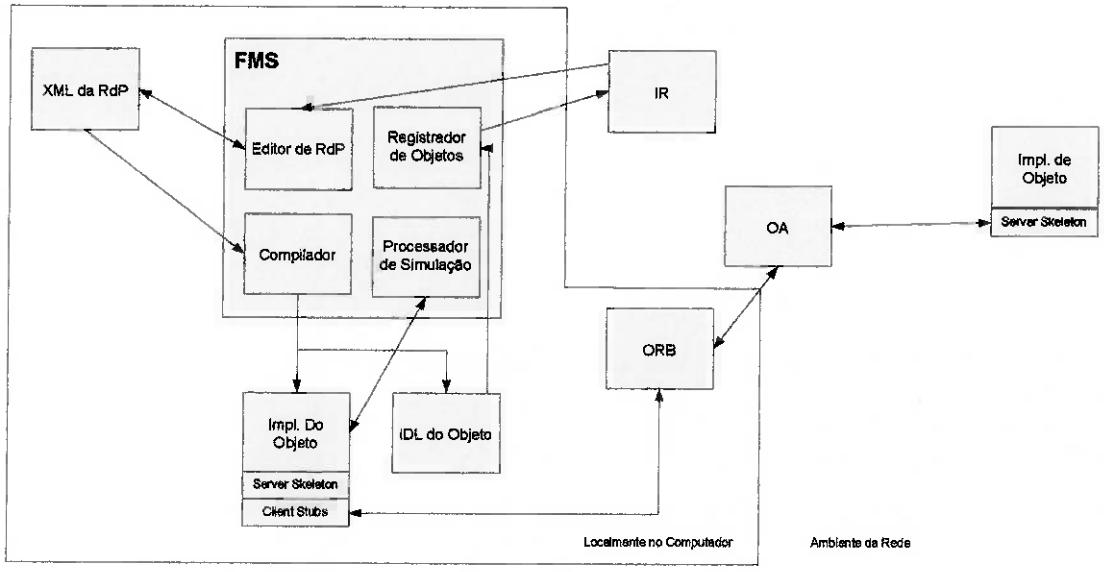


Figura 4-2 - Arquitetura da Plataforma de Simulação baseada em CORBA

Neste modelo, existem quatro elementos contidos na FMS, descritos a seguir:

- Editor de RdP: este elemento seria um ambiente com interface para que o usuário criasse modelos baseados em RdP. Estes modelos seriam armazenados na forma de documentos XML como indicado na Figura 4-2.
- Compilador: o compilador tem a tarefa de interpretar os documentos XML e transformá-los em uma implementação de um objeto CORBA. A implementação gerada teria um *skeleton* para processar chamadas de *objetos* externos, e *stubs* para acessar outros *objetos*. Os *stubs* e *skeletons* são utilizados para descrever a interface das fusões de *transições* entre *objetos*, entre outras funções, como enviar informações sobre as conexões entre os *objetos* para controlar e gerenciar o tempo de simulação. A Figura 4-3 mostra que quando um *objeto* é criado pelo compilador, este inclui os *sockets* necessários para estabelecer a rede de

gerenciamento de tempo entre os *objetos*. Além da implementação dos *objetos*, o compilador gera também um documento IDL que descreve as interfaces do *objeto*.

- Registrador de Objetos: os *objetos* criados pelo compilador precisam ser registrados junto a um OA e suas IDLs podem ser registradas em um IR. O registro do *objeto* junto a um OA, serve para que outros *objetos* possam encontrá-lo e acessá-lo, enquanto que o registro da IDL no IR, serve para que outros *objetos* possam ser criados com *stubs* de acesso aos seus métodos. Ou seja, um desenvolvedor pode criar um outro *objeto* com uma RdP que faz a fusão com as *transições* oferecidas pelo primeiro *objeto*.
- Processador de Simulação: o processador de simulação é um cliente que acessa o primeiro *objeto* de um modelo que deve ser simulado. Desta forma, o processador invoca este *objeto* e faz um pedido para que a simulação seja iniciada.

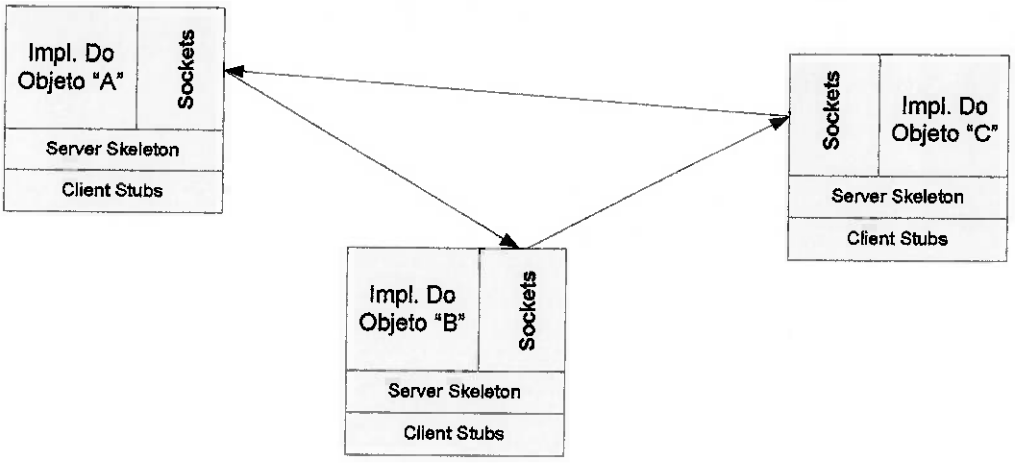


Figura 4-3 - Objetos CORBA gerados pelo compilador com a estrutura de sockets embutida

Além da FMS, a arquitetura da *Plataforma de Simulação* contém um ORB para que a comunicação entre os *objetos* possa ser efetuada. Na Figura 4-2, pode-se notar que, quando o *processador de simulação* inicia um *objeto*, este contém interfaces com outros *objetos* remotos. Para acessá-los, é necessário ter um ORB sendo executado e dispor de um OA na rede para encontrar o *objeto* desejado.

5. Comentários finais e Conclusões

Com base nos estudos realizados, verificou-se que o fato da *Plataforma de Simulação* ser um sistema distribuído e modular, fez com que esta tenha uma estrutura similar aos aplicativos orientados a *objetos* ou então aplicativos orientados a *serviços*, o que permite que sejam empregadas as tecnologias CORBA e WS, respectivamente.

O trabalho confirmou que ambas as tecnologias são capazes de implementar e executar o programa de teste proposto satisfatoriamente. Isto significa que a principio, ambas poderiam ser utilizadas como base para implementar a integração das partes da *Plataforma de Simulação*. Existem, porém, características distintas entre os dois métodos que podem ser decisivas na escolha entre uma das tecnologias.

O primeiro aspecto refere-se à interoperabilidade alcançada por cada um dos métodos. Em CORBA, a ligação entre o programa que implementa um determinado *objeto* e o ORB, e alguns *objetos* que o ORB acessa é *tight-coupled*, o que significa dizer que o programa e o ORB têm uma forte interação, relacionada à linguagem de programação utilizada para o desenvolvimento dos *objetos* e o ORB. Isto pode limitar a utilização de *objetos* em diferentes plataformas. Além disso, a implementação do ORB em si está ligada a um tipo de sistema operacional para o qual foi desenvolvido.

Por outro lado, os WS são totalmente baseados em documentos XML, o que faz com que esta tecnologia seja totalmente independente do sistema de hardware e software utilizados na implementação de *serviços*. Assim, caso se deseje projetar uma plataforma capaz de ser utilizada com diferentes sistemas, e permitir que terceiros elaborem suas próprias soluções

para interagir com a *Plataforma de Simulação*, a tecnologia baseada em WS é mais apropriada do que o CORBA.

O segundo aspecto refere-se ao desempenho das duas tecnologias. Estudos mostraram que a diferença de tempo de execução e resposta entre o CORBA e WS pode chegar a uma relação de 400:1 (Elfwing, 2002). Isto significa que se o tempo de execução for um parâmetro importante para a escolha do método utilizado na *Plataforma de Simulação*, deve-se abrir mão da interoperabilidade oferecida pelos WS e optar pelo CORBA e mesmo com a identificação dos pontos críticos e alterações efetuadas na forma com que as mensagens SOAP são passadas pelo protocolo HTTP, a diferença ainda continua alta, de 7:1.

Outros aspectos que podem ser mencionados referem-se à facilidade de programar e compilar *objetos* e *serviços* utilizando cada um dos métodos. Muitas das diferenças entre o processo de desenvolvimento com CORBA e WS dependem de qual linguagem de programação está sendo utilizada. Caso seja utilizada uma estrutura de *Plataforma de Simulação* baseado nas soluções apresentadas no tópico de análise de resultados, nota-se que é preciso desenvolver um conversor de arquivos XML (que representam as RdP) em código de programação a ser compilado por algum compilador, para disponibilizar os *serviços* e *objetos*. A dificuldade de desenvolver este conversor para cada um dos métodos é relativamente a mesma, porém, outras configurações que precisam ser feitas após a compilação, como registrar *objetos* e disponibilizá-los para terceiros, no caso do CORBA, mostram-se muito dependentes do tipo e versão de ORB que está sendo utilizado. Os WS dependem do tipo de servidor *web* que está sendo utilizado para disponibilizar *serviços* na rede, porém o registro neste caso é relativamente menos trabalhoso. Além disso, pelo fato de todas as especificações e acoplamentos entre partes remotas serem feitas através de documentos XML no caso dos WS, não é preciso desenvolver um interpretador e compilador para cada linguagem de

programação diferente que se queira utilizar, pois o XML é exatamente o mesmo para qualquer sistema operacional e linguagem de programação.

APÊNDICE A - UMA VISÃO GERAL DE REDES TCP/IP

O texto a seguir, são trechos adaptados de Lorenzo (2005).

Atualmente, dentre os vários protocolos de comunicação de alto nível entre máquinas que existem, o mais difundido é o protocolo conhecido por TCP/IP. Este é um conjunto de protocolos originalmente desenvolvido pela Universidade da Califórnia em Berkeley, sob contrato para o Departamento de Defesa dos EUA, e tornou-se o conjunto de protocolos padrão das redes locais e remotas, suplantando conjuntos de protocolos bancados por grandes empresas do mercado, como a *IBM* (com o protocolo *SNA*), *Microsoft* (com o protocolo *NetBIOS/NetBEUI*) e *Novell* (com o protocolo *IPX/SPX*).

Mesmo antes do popularização da *Internet*, o TCP/IP já era o protocolo obrigatório para grandes redes, formadas por produtos de muitos fornecedores diferentes, e havia sido escolhido pela *Microsoft* como o protocolo preferencial para o seu sistema operacional *Windows NT*, devido às limitações técnicas do seu próprio conjunto de protocolos, o *NetBEUI*.

Todos os *softwares* de redes são baseados em alguma arquitetura de camadas, e normalmente nos refere-se a um grupo de protocolos criado para funcionar em conjunto como uma pilha de protocolos (em inglês, *protocol stack*, por exemplo "*the TCP/IP stack*"). O termo "pilha" é utilizado porque os protocolos de uma dada camada normalmente interagem somente com os protocolos das camadas imediatamente superior e inferior.

1 O modelo de pilha de 4 camadas do TCP/IP

O TCP/IP foi desenhado segundo uma arquitetura de pilha, onde diversas camadas de software interagem somente com as camadas imediatamente acima e abaixo. Há diversas semelhanças com o modelo conceitual OSI da *International Standarization Organization* (ISO), mas o TCP/IP é anterior à formalização deste modelo e, portanto possui algumas diferenças.

OSI é um modelo conceitual, e não a arquitetura de uma implementação real de protocolos de rede. Mesmo os protocolos definidos como padrão oficial pela ISO, a entidade criadora do modelo OSI, não foram projetados e construídos segundo este modelo.

O nome TCP/IP vem dos nomes dos protocolos mais utilizados desta pilha, o *Internet Protocol* (IP) e o *Transmission Control Protocol* (TCP). Mas a pilha TCP/IP possui ainda outros protocolos, dos quais serão apresentados aqui apenas os mais importantes, indispensáveis para que o TCP e o IP desempenhem corretamente as suas funções.

Visto superficialmente, o TCP/IP possui 4 camadas, desde os programas de rede até o meio físico que carrega os sinais elétricos até o seu destino (Tabela A-1):

Tabela A-1 - Camada de protocolos TCP/IP

Camada		Protocolo
4	Aplicação (Serviço)	FTP, TELNET, LPD, HTTP, SMTP/POE3, NFS, etc.
3	Transporte	TCP, UDP
2	Rede	IP
1	Enlace	Ethernet, PPP, SLIP

Além das camadas propriamente ditas, tem-se uma série de dispositivos, que realizam a interface entre as camadas (Tabela A-2):

Tabela A-2 - Componentes auxiliares para a utilização dos protocolos

Camada	Componentes
Aplicação / Transporte	DNS, Sockets
Rede / Enlace	ARP, DHCP

1.1 Descrição das funções de cada camada TCP/IP

Os protocolos de enlace têm a função de fazer com que informações sejam transmitidas de um computador para outro em uma mesma mídia de acesso compartilhado (também chamada de rede local) ou em uma ligação ponto-a-ponto (ex: modem).. A preocupação destes protocolos é permitir o uso do meio físico que conecta os computadores na rede e fazer com que os bytes enviados por uma máquina cheguem a uma outra máquina diretamente desde que haja uma conexão direta entre eles.

Já o protocolo de rede, o (IP), é responsável por fazer com que as informações enviadas por uma máquina cheguem a outras máquinas mesmo que eles estejam em redes fisicamente distintas, ou seja, mesmo que não exista uma conexão direta entre eles. Como o próprio nome (*Internet*) diz, o IP realiza a conexão entre redes. E é ele quem assegura a capacidade da rede TCP/IP de se "reconfigurar" quando uma parte da rede está fora do ar, procurando um caminho (rota) alternativo para a comunicação.

O objetivo dos protocolos de transporte é diferente dos protocolos que se encontram em um nível inferior da *pilha*. Ao invés de conectar dois equipamentos, estes têm a tarefa de conectar dois programas. Pode-se assim ter em um mesmo computador vários programas trabalhando com a rede simultaneamente, por exemplo, um w*eb browser* e um leitor de e-mail. Da mesma forma, um mesmo computador pode estar executando ao mesmo tempo um

web server e um servidor POE3. Os protocolos de transporte (UDP e TCP) atribuem a cada programa um número de porta, que é anexado a cada pacote de mensagens de modo que o TCP/IP saiba para qual programa entregar cada mensagem recebida pela rede.

Finalmente os protocolos de aplicação são específicos para cada programa que faz uso da rede. Desta forma existe um protocolo para a conversação entre um *web server* e um *web browser* (HTTP), um protocolo para a conversação entre um cliente *Telnet* e um servidor (*daemon*) *Telnet*, e assim por diante. Cada aplicação de rede tem o seu próprio protocolo de comunicação, que por sua vez utiliza-se dos protocolos das camadas mais baixas para poder atingir o seu destino.

Pela Tabela A-2, nota-se que existem dois protocolos de transporte no TCP/IP. O primeiro é o UDP, um protocolo que trabalha com *datagramas*, que são mensagens com um comprimento máximo pré-fixado e cuja entrega não é garantida. Caso a rede esteja congestionada, um *datagrama* pode ser perdido e o UDP não informa os programas desta ocorrência. Outra possibilidade é que o congestionamento em uma rota da rede possa fazer com que os pacotes cheguem ao seu destino em uma ordem diferente daquela em que foram enviados. O UDP é um protocolo que trabalha sem estabelecer conexões entre os programas que estão se comunicando.

Já o TCP é um protocolo orientado a conexão. Ele permite que sejam enviadas mensagens de qualquer tamanho e cuida de quebrá-las em pacotes que possam ser enviados pela rede. Ele também é responsável por rearranjar os pacotes quando chegam ao destino e de retransmitir qualquer pacote que seja perdido pela rede, de modo que a mensagem entregue, seja idêntica a mensagem enviada originalmente.

1.1.1 Descrição de Dispositivos de redes TCP/IP

O *Domain Name Service* (DNS) é um dispositivo que será visto com maiores detalhes mais adiante e fornece os nomes lógicos da Internet como um todo ou de qualquer rede TCP/IP isolada.

Temos ainda o ARP que é um dispositivo que realiza o mapeamento entre os endereços TCP/IP e os endereços *Ethernet*, de modo que os pacotes possam atingir o seu destino em uma rede local (pois quem efetivamente entrega os pacotes de informações na rede local é o *Ethernet*, não o TCP ou o IP).

Existem também os componentes que ficam na interface entre os níveis 3 e 4 e entre os níveis 1 e 2.

O *sockets* é uma API[11] para a escrita de programas que trocam mensagens utilizando o TCP/IP. Ele fornece funções para testar um endereço de rede, abrir uma conexão TCP, enviar *datagramas* UDP e esperar por mensagens da rede.

Em uma rede TCP/IP, cada máquina (ou melhor, cada placa de rede, caso a máquina possua mais do que uma) possui um endereço numérico formado por 4 octetos (4 *bytes*), geralmente escritos na forma w.x.y.z. Além deste *endereço IP*, cada máquina possui uma máscara de rede (*network mask* ou *subnet mask* ou *netmask*), que é um número do mesmo tipo, mas com a restrição de que ele deve começar por uma seqüência contínua de bits em 1, seguida por uma seqüência contínua de bits em zero. Ou seja, a máscara de rede pode ser um número como 11111111.11111111.00000000.00000000 (255.255.0.0), mas nunca um número como 11111111.11111111.00000111.00000000 (255.255.7.0).

A máscara de rede serve para quebrar um endereço IP em um endereço de rede e um endereço de *host*[12]. Todos os computadores em uma mesma rede local (fisicamente falando,

---

[11] API: Application Programmer Interface é uma interface que permite que um programa utilize recursos oferecidos por um outro programa, fazendo-se uma chamada a este programa, ou sendo chamado por ele.

por exemplo, um mesmo barramento *Ethernet*) devem ter o mesmo endereço de rede, e cada um deve ter um endereço de *host* diferente. Tomando-se o endereço IP como um todo, cada computador em uma rede TCP/IP (inclusive em toda a *Internet*) possui um endereço IP único e exclusivo.

1.1.2 Endereçamento e roteamento

O *InterNIC* controla todos os endereços IP em uso ou livres na Internet, para evitar duplicações, e reserva certas faixas de endereços chamadas de endereços privativos para serem usados em redes que não irão se conectar diretamente à *Internet*.

Quando o IP recebe um pacote para ser enviado pela rede, ele quebra o endereço destino utilizado a máscara de rede do computador e compara o endereço de rede de destino com o endereço de rede dele mesmo. Se os endereços de rede forem iguais, isto significa que a mensagem será enviada para um outro computador na mesma rede local, neste caso o pacote é repassado para o *protocolo de enlace* apropriado (em geral o *Ethernet*). Se os endereços forem diferentes, o IP envia o pacote para o *default gateway*, que é o equipamento que fornece a conexão da rede local com outras redes. Este equipamento pode ser um *roteador*[13] dedicado ou pode ser um servidor com múltiplas placas de rede, e se encarrega de encaminhar o pacote para a rede local onde está o endereço IP do destino.

É importante que o endereço IP do *default gateway* esteja na mesma *subnet* que o da máquina sendo configurada, caso contrário ela não terá como enviar pacotes para o *default gateway* e assim só poderá se comunicar com outros *hosts* na mesma *subnet*.

---

[12] Host: servidor de dados ou programas

[13] Roteador: dispositivo de comunicação que interliga duas ou mais redes e determina o melhor caminho a ser seguido por um pacote de mensagem para chegar ao seu destino, através de informações sobre as redes e algoritmos que decidem pelo melhor caminho.

Resumindo, uma máquina qualquer em uma rede TCP/IP deve ser configurada com pelo menos estes três parâmetros: o seu *endereço IP* exclusivo, a sua *máscara de rede* (que deve ser a mesma utilizada pelas demais máquinas na mesma LAN[14]) e o *endereço IP* do *default gateway*.

1.1.3 Como se processa a comunicação em uma rede TCP/IP.

Suponha que o *host* com o *endereço IP* 172.16.1.101 deseje enviar um pacote para o endereço 172.16.2.102. Caso a *máscara de rede* seja 255.255.0.0, fazendo-se a operação AND binário do endereço fonte será 172.16.0.0, e o AND binário do endereço destino será 172.16.0.0, indicando que ambos possuem o mesmo endereço de rede e, portanto estão diretamente conectados no nível de enlace.

Neste caso, o nível IP envia um pacote ARP pela rede *Ethernet* para identificar qual o endereço *Ethernet* do *host* cujo IP é 172.16.2.2. Este pacote é enviado como um *broadcast*[15], de modo que todos os *hosts* conectados no mesmo segmento *Ethernet* receberão o pacote, e o *host* configurado para o endereço desejado irá responder ao pacote ARP indicando o seu endereço *Ethernet*. Assim o IP pode montar o pacote *Ethernet* corretamente endereçado e enviá-lo ao seu destino.

Agora suponha que a *máscara de rede* não fosse 255.255.0.0, mas sim 255.255.255.0. Neste caso, os endereços de rede da origem e destino seriam, respectivamente, 172.16.1.0 e 172.16.2.0. Como os endereços de rede são diferentes, isto significa que não se tem conectividade direta (no nível de enlace) entre os dois *hosts*, portanto o pacote deverá ser entregue por intermédio de um *roteador*.

---

[14] LAN. *Local Area Network* é uma rede de computadores que se encontra em uma área específica, como uma rede doméstica, ou uma rede em um escritório. A topologia deste tipo de rede define sua estrutura física.

[15] Broadcast: um pacote de informações enviado para todos os pontos de uma rede

Digamos que o *default gateway* seja 172.16.1.1 (observe que o endereço de rede do *default gateway* é 172.16.1.0, o mesmo do *host* de origem). Então o *host* irá enviar um pacote ARP pela rede para descobrir o endereço *Ethernet* do *default gateway*, e enviará o pacote para este.

Ao receber o pacote, o *default gateway* verificará que o endereço IP de destino é o IP de outro *host* diferente do dele, e irá processar qual o endereço de rede do destino. Pode ser que o pacote esteja endereçado para uma rede local com a qual o *default gateway* tenha uma conexão direta, ou pode ser que o *default gateway* tenha que direcionar o pacote para um outro *roteador* mais próximo do destino final. De qualquer forma, o *default gateway* segue o mesmo processo de gerar o endereço de rede utilizando a *netmask*, e em seguida enviar um pacote ARP pedindo o endereço *Ethernet* do próximo *host* a receber o pacote. A diferença é que um *roteador* não tem um *default gateway*, mas sim uma *tabela de roteamento*, que diz quais endereços de rede podem ser alcançados por quais *roteadores*.

Este exemplo considerou apenas a comunicação entre dois equipamentos, não entre dois programas, ficando, portanto apenas no nível de rede da pilha TCP/IP. O próximo nível tem um processamento simples e funciona da seguinte forma: o IP verifica que tipo de pacote foi recebido (TCP, UDP ou outro) e repassa o pacote para o protocolo apropriado.

O protocolo de transporte irá então verificar o número de *porta* contido no *pacote* e qual programa está utilizando ela. Este programa será notificado da chegada de um *pacote*, e será responsabilidade dele decodificar e utilizar de alguma forma as informações contidas no *pacote*.

APÊNDICE B - COMO FUNCIONAM OS SOCKETS

Os *sockets* foram desenvolvidos com o intuito de estabelecer um eio de se comunicar entre dois PCs que seja similar ao método utilizado para manipulação de arquivos, como: abrir, ler, escrever e fechar (Colibri, Felix John, 2005).

A principal diferença entre um sistema de arquivo e uma rede de comunicação é que um sistema de arquivos está sempre disponível e pronto para ser acessado em uma máquina, enquanto que em uma rede de comunicação é preciso primeiramente encontrar o computador que se deseja acessar e verificar se está disponível.

A comunicação entre computadores através de *sockets* segue o modelo *Cliente Servidor* onde: servidor é iniciado e aguarda por clientes quando um cliente deseja comunicar-se, deve procurar por um servidor, iniciar a comunicação, que se prolonga até o instante em que o cliente e/ou o servidor resolvem interrompê-la.

Portanto, tem-se *sockets* que têm a funcionalidade de clientes e de servidores em uma conexão. Como o protocolo utilizado pelo *sockets* é o TCP/IP, a identificação do servidor é feita através de um número de IP. Quando um cliente que deseja enviar uma informação a um servidor, este precisa procurá-lo na rede e, ao encontrá-lo, este precisa informar que tipo de operação deseja realizar; enviar um e-mail, fazer o *download* de uma página, fazer o *download* de um arquivo, etc. As operações de operações oferecidas pelo servidor são chamadas *serviços*. Cada *serviço* é associado a um número de porta, assim, o padrão para, por exemplo, enviar uma página da Internet é a porta 80 e para enviar e-mail a porta de número 110. O número da porta está associado a um tipo de protocolo que especifica o formato e a estrutura da comunicação. Ao se criar um novo protocolo, pode-se associar um número de

porta que ainda não tenha nenhum protocolo associado, como será o caso nos programas de exemplo documentados mais adiante.

1.2 Estrutura dos *sockets*

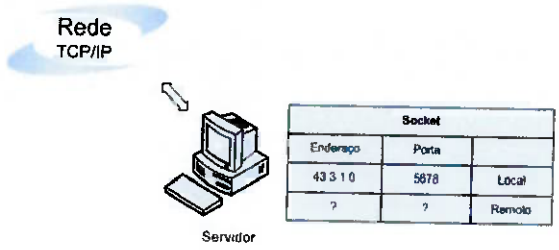
Basicamente, um *socket* é uma estrutura de dado que contém quatro propriedades:

- endereço local;
- porta local;
- endereço remoto;
- porta remota.

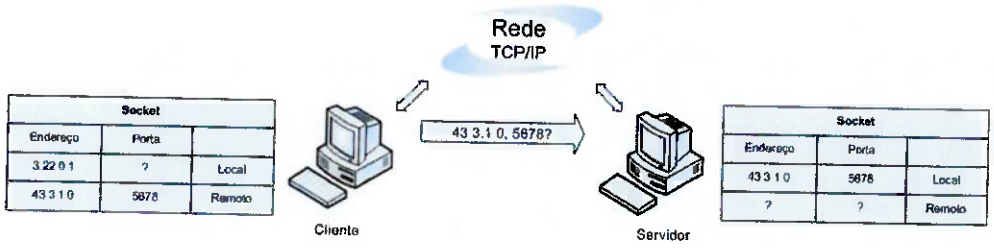
Para iniciar a comunicação entre dois computadores, precisa-se criar uma estrutura de *socket* através de um programa e preencher os dados de endereço e porta local. Para que a conexão seja feita, um cliente precisa localizar um servidor que esteja disponível no número de porta especificado. Quando o servidor é encontrado na rede, as outras propriedades da estrutura do *socket* são preenchidas e o *serviço* pode ser oferecido pelo servidor. Para ilustrar este processo, a Figura B-1 mostra de forma esquemática as etapas para a conexão via *sockets* entre dois computadores.

Um servidor tem a capacidade de conectar-se a vários clientes simultaneamente, portanto as propriedades das conexões com os clientes não podem ser armazenadas em apenas uma estrutura de *socket*. Para viabilizar isto, a cada nova conexão com um cliente, o servidor automaticamente cria uma nova instância de um *socket*, que irá se dedicar àquela conexão. Com isso, as conexões estabelecidas são executadas em uma estrutura dedicada e à parte, enquanto o servidor continua livre e pode aguardar por outros pedidos de conexão de clientes. Este processo está ilustrado na Figura B-2.

O servidor é iniciado com o endereço IP: 43.3.1.0



Um cliente é iniciado no IP: 3.22.0.1 e tenta conectar-se ao servidor.



O servidor recebe o pedido de conexão e os dados de ambos os sockets são completados.

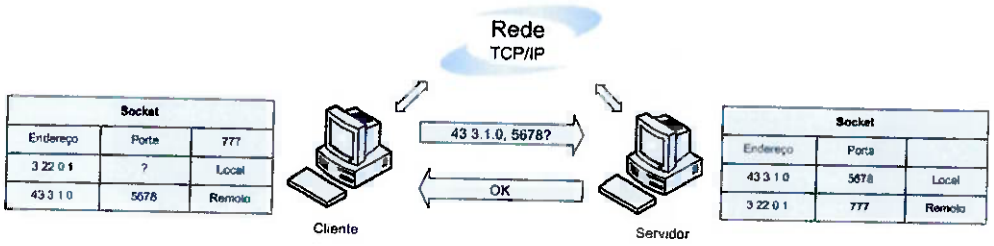
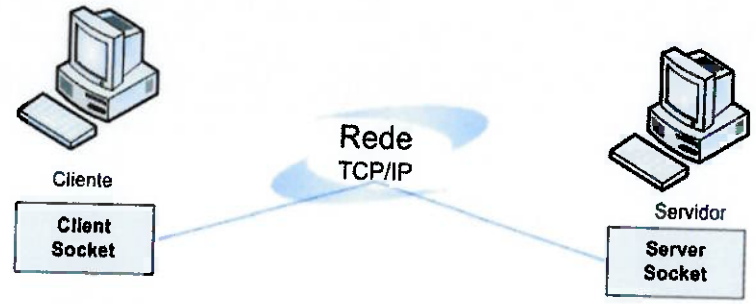
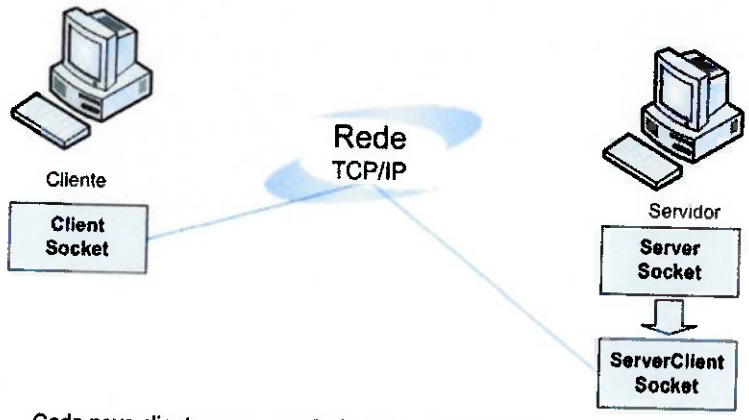


Figura B-1 - Conexão entre sockets cliente e um socket servidor

O servidor utiliza um *ServerSocket* para receber os pedidos de conexão de clientes. Cada novo cliente inicialmente conecta-se ao *ServerSocket*.



O servidor cria um novo *ClientServerSocket* para onde direciona a conexão com o cliente, assim o *ServerSocket* fica livre para receber novos pedidos de conexão.



Cada novo cliente usa seu próprio ClientServerSocket.

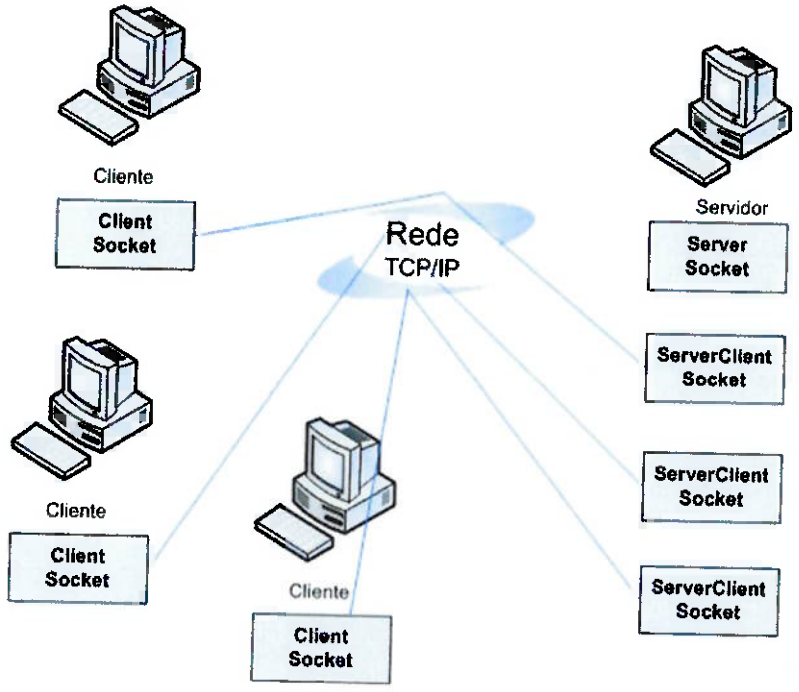


Figura B-2 - Representação da relação de múltiplos clientes para um servidor

1 Conexões blocking e non-blocking

O pacote *Indy* utilize-se de *sockets* do tipo *blocking* (ou *synchronous*). Este tipo de *sockets* funciona de forma muito semelhante ao processo de leitura e escrito em um arquivo comum. Quando deseja-se ler ou escrever dados em uma conexão, utiliza-se funções com *Read* e *Write*, que somente retornam quando a operação tiver sido completada e impedem com isso que o restante do programa seja processado neste meio tempo. À primeira vista, pode parecer que trabalhar com conexões do tipo *blocking* traz desvantagens por inserir gargalos de processamento no programa, mas este não é o caso. Um exemplo pode ser dado imaginando-se uma situação em um servidor precisa ler dados de uma conexão enviados por um cliente, através de um *socket* do tipo *blocking*. Como o servidor não sabe quando os dados irão chegar, este inicia a rotina de leitura de dados da conexão e espera até o momento em que as informações enviadas pelo cliente cheguem e possam ser lidas, bloqueando o processamento do restante do programa até o termino da leitura. Como será visto adiante, pode-se utilizar recursos de processamento paralelo neste caso, fazendo com que a leitura dos dados da conexão ocorra simultaneamente com a execução do restante do programa, evitando o problema de bloqueio.

Os *sockets* surgiram primeiramente nos sistemas operacionais Unix, onde era possível executar processamentos paralelos em programas. Porém, quando os *sockets* foram adaptados ao sistema operacional *Windows* pela primeira vez, surgiu um problema, pois este sistema operacional não era capaz de executar processamento paralelo. Para solucionar este problema, o método de funcionamento dos *sockets* teve que ser adaptado, surgindo então o conceito de *sockets non-Blocking*, que adequavam-se a sistemas operacionais que não permitiam processamento paralelo, e ainda assim conseguiam evitar o problema do bloqueio da execução do restante do programa

Isto foi conseguido mudando-se o conceito inicial dos *sockets* que trabalhavam de uma forma seqüencial, para um modelo de *sockets* baseada em eventos, onde as rotinas de leituras, escrita, abertura e fechamento eram executadas no disparo de eventos associados a estas ações. Com o surgimento do *Win32*[16], esta limitação foi retirada, porém o conceito de *sockets non-Blocking* prevaleceu e atualmente muitas implementações de *sockets* permitem criar conexões tanto do tipo *blocking*, quanto *non-blocking*.

Nos sistemas operacionais *Unix* e *Linux*, o único modo de conexões de *sockets* que existe são do tipo *blocking*, motivo pelo qual os componentes *Indy* seguem esta linha de programação, fato que viabiliza que o pacote seja multi-plataformas.

Abaixo estão relacionadas às vantagens de cada um dos dois tipos de conexão:

1.3 Vantagens das conexões blocking:

- **Fácil de programar:** é relativamente fácil programar com conexões do tipo *blocking*, pois todo o código referente à conexão pode ficar agrupado em um único local, seguindo uma seqüência em séria desde sua abertura até o fechamento.
- **Fácil exportar para sistemas *Unix* e *Linux***: como o *Unix* e *Linux* utilizam este padrão de conexão, torna-se relativamente fácil escrever código de programação com portabilidade entre estes sistemas operacionais e implementar esta lógica nestas plataformas.
- **Funcionam bem com processamento paralelo**: como o código de conexões do tipo *blocking* é seqüencial e fica encapsulado em um único local, fica relativamente fácil utilizá-lo com processamento paralelo.

---

[16] Win32: Sistema operacional da Microsoft à partir da versão Windows 95, que utiliza instruções de 32 bits e passou também a suportar processamento paralelo.

eliminando as vantagens oferecidas por este recurso. Neste projeto, por serem utilizados os componentes *Indy*, serão utilizadas conexões do tipo *blocking* com recursos de *multi-threading*.

A seguir apresenta-se as principais vantagens em se utilizar *threading* em *sockets* com conexões do tipo *blocking*:

- **Priorização** – É possível ajustar prioridades para diferentes *threads* de conexões, permitindo que certas rotinas tenham mais ou menos tempo de processamento.
- **Encapsulamento** – Cada conexão fica encapsulada em uma seqüência de instruções diminuindo a chance de diferentes conexões interferirem umas nas outras.
- **Segurança** – Cada *thread* pode ter diferentes atributos de segurança.
- **Múltiplos processadores** – *Threading* automaticamente fará uso das vantagens em máquinas com múltiplos processadores.
- **Concorrência** – *Threading* promove um processamento concorrente, sem o qual cada conexão teria que ser manipulada por um único processo. Para que isso funcione, cada tarefa precisa ser dividida em pequenas tarefas que podem ser executadas rapidamente. Se por acaso alguma tarefa demorar muito a ser executada, todas as outras tarefas ficarão paradas. Com *threading*, cada tarefa pode ser programada como uma seqüência completa e fechada de instruções e a divisão do tempo de processamento no processador fica a cargo do sistema operacional.

BIBLIOGRAFIA

BERALDI, R.; NIGRO, L. "Distributed Simulation of Timed; Petri Nets: A Modular Approach Using Actors and Time; Warp" IEEE Concurrency, vol. 7, no.4, 52-62, 1999

BEZNOSOV, KONSTSNTIN, FLINN, DONALD J., HARTMAN, BRET, KAWAMOTO, SHIRLEY. Introduction to Web services and their security. University of British Columbia, Vancouver, Canada. Information Security Technical Report, 2005.

BUSSLER, CHRISTOPH, FENSEL, DIETER. The Web Service Modeling Framework, Division of Mathematics and Computer Science, Faculty of Science, Vrije Universiteit Amsterdam Neederland, 2002.

CENTERO, MARTHA A. An Introduction to Simulation Modeling; Department of Industrial & System Engineering, College of Engineering and Design, Florida International University, USA 1996.

CORBANET; Demonstrating ORB Interworking through WWW Integration. Disponível em <http://archive.dstc.edu.au/events/dse96/demos/flyers/corbanet-flyer.html>; Acesso em 12 de Junho de 2005.

DOSYUKOV, S. Distributed Information System. From A to Z. – Borland Developers Network, 2003. Disponível em <http://bdn.borland.com/article/0,1410,30025,00.html>; Acesso em 02 de Abril de 2005.

ELFWING, ROBERT; LUNDBERG, LARS; PAULSSON ULF, Performance of SOAP in Web Service Environment Compared to CORBA; Department of Software Engineering and Computer Science, Nlekinge Institute of Technology, SE-372 25 Ronneby, Sweden, 2002.

ERL, T. Server-Oriented Architecture: A Field Guide to Integrating XML and Web Services; Prentice Hall PTR; First Edition; 2004.

FARREL, MICHAEL JR., LUBLINSKY, BORIS, Web Services the implementation Iceberg, eAI Journal - Digital Magazine (<http://www.bijonline.com/Default.asp>); June 2002;

FENSEL, D., BUSSLER, C. The Web Service Modeling Framekork WSMF. Division of Mathematics and Computer Science, Faculty of Science, Vrije Universiteit Amsterdam (VU), De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands. Publishet at the Electronic Commerce Research and Application, 2002.

FREE CORBA PAGE; Free ORBs; Disponível em: <http://adams.patriot.net/~tvalesky/freecorba.html>; Acesso em 22 de Novembro de 2005.

HARVEY, M. Multi-threading – the Delphi way; Disponível em: <http://www.pergolesi.demon.co.uk/prog/threads/ToC.html#Introduction>; 2000. Acesso em 05/07/2005.

HO, Y.C., CAO, X.R. Pertubation Analysis of Discrete Event Dynamic Systems, Kluwer Ac. Publishers, 1991.

HOWER, CHAD Z. Why Indy? <http://www.atozed.com/indy/Texts/WhyIndy.iwp> Acesso em: 10/01/2005

INDY PROJECT, THE The Indy Project; <http://www.indyproject.org>. Acesso em: 18/09/2004

JUNQUEIRA, FABRICIO, MIYAGI, PAULO E., VILLANI, EMÍLIA. A Petri Net Based Platform for Distributed Modeling and Simulation of Productive Systems; Escola Politécnica da USP, 2005. Trabalho apresentado no Emerging Technologies and Factory Automation (ETFA), 2005.

LEE, W. B., LAU, H. C. W., Multi-agent modeling of dispersed manufacturing networks, Pergamon, Expert Systems with Applications, No. 16, pp. 297-306, 1999.

LORENZA, FERNANDO, Arquitetura de Redes TCP/IP; Disponível em: <http://www.lozano.eti.br/>. Acesso em: 22/09/2005

IBM, Web Services Architecture Overview, Setembro de 2000; Disponível em <http://www-128.ibm.com/developerworks/web/library/w-ovr/index.html> . Acesso em 5 de Outubro de 2005.

LONG, BRIAN; CORBA and Delphi; Disponível em: <http://www.blong.com/Conferences/DCon99/Corba/Corba.htm#_Toc462140981>; Acesso em 22 de Novembro de 2005.

LOOSELY COUPLED WEBSITE, Glossary. Disponível em <http://looselycoupled.com/glossary/loose%20coupling>. Acesso em 12 de Agosto de 2005.

MOORE, KEDRA. E; BRENNAN, JOHN E. Alpha/Sim Simultion Software Alphatec, Inc. Burlington, MA 018003-4562, USA, 1996.

MURATA, T., Petri Nets - Properties, Analysis and Applications, Proceedings of the IEEE, vol.77, no. 4, 1989.

MURPHY, NIALL, Introduction to CORBA for Embedded Systems, Disponível em <http://www.embedded.com/98/9810fe2.htm> Acesso 15 de Maio de 2005.

OASIS UDDI, About UDDI, Disponível em <http://www.uddi.org/about.html>; Acesso em 15 de Outubro de 2005.

OASIS UDDI, UDDI Spec Technical Committee Draft, Dated 20041019; Disponível em <http://uddi.org/pubs/uddi-v3.0.2-20041019.htm>; Acesso em 15 de Outubro de 2005.

OMG; Catalog of OMG IDL / Language Mappings Specifications. Disponível em: <http://www.omg.org/technology/documents/idl2x_spec_catalog.htm>. Acesso em 12 de Junho de 2005.

PIDD, MICHAEL The Management School, Lancaster University, UK. 1994.

PILHOFER, F. Design and Implementation of Portable Object Adapters, Johann Wolfgang Goethe – Universität. Frankfurt am Main, Alemãnha. 23 de Junho de 1999. Disponível em <http://www.fpx.de/fp/Uni/Diplom/diplom.html>. Acesso em 15 de Setembro de 2005.

SANZ, R., ALONSO, M., CORBA for control systems, Pergamon, Annual Reviews in Control, No. 25, pp. 169-181, 2001.

SEILA, ANDREW F. Introduction to Simulation, Terry College of Business, The University of Geórgia, USA, 1995.

SHI, Y., GREGORY, M., International manufacturing networks – to develop global competitive capabilities, Journal of Operations Management, Vol. 1, No. 16, pp. 195-214, 1998.

SUN MICROSYSTEMS, Introduction to CORBA, Disponível em <http://java.sun.com/developer/onlineTraining/corba/corba.html>. Acesso 20 de Junho de 2005.

VINOSKI, S.; CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments. IONA Technologies, Inc., 60 Aberdeen Ave. Cambrige, MA USA, 1996.

W3C - WORLD WIDE WEB CONSORCIUS, Web Interface Definition Language, 22 de Setembro de 1997, Disponível em <http://www.w3.org/TR/NOTE-widl-970922> Acesso 12 de Junho de 2005.

W3C - WORLD WIDE WEB CONSORCIUS, WSDL 2.0 http Binding Namespace, 30 de Setembro de 2003, Disponível em <http://www.w3.org/2006/01/wsdl/http> Acesso 15 de Outubro de 2005.

W3C - WORLD WIDE WEB CONSORCIUS, SOAP Specifications Version 1.2, 24 de Junho de 2003, Disponível em <http://www.w3.org/TR/soap/> Acesso 12 de Junho de 2005.

W3C - WORLD WIDE WEB CONSORCIUS, Web Services Architecture, 11 de Fevereiro de 2003, Disponível em <http://www.w3.org/TR/ws/arch/> Acesso 12 de Junho de 2005.

WEBOPEDIA, OpenDoc; Disponível em <http://www.webopedia.com/TERM/O/OpenDoc.html>; Acesso 26 de Outobro de 2005.

WIKIPEDIA A ENCICLOPÉDIA LIVRE, Framework, Disponível em <http://pt.wikipedia.org/wiki/Framework>; Acesso em 01/11/2005.