4
XMLTalk

O XMLTalk é um *framework* que disponibiliza uma biblioteca de classes Java que auxiliam a construção de aplicações orientadas a serviços utilizando servidores de aplicação J2EE. Além disso, o XMLTalk facilita a criação de aplicações que seguem esta arquitetura através da adoção dos seus padrões de uso.

O uso do XMLTalk possibilita construir um componente de software distribuído e reutilizável (serviço) através da implementação de apenas um método de uma interface bem definida. Os serviços podem ser implementados de formas distintas, como XML WebServices, classes Java normais (POJO – *Plain Old Java Objects*), Scripts e *Enterprise Java Beans (EJB)*. Porém, as distintas implementações são acessadas de forma transparente para as aplicações clientes. Todas invocações para execução dos serviços passam antes por um *Gestor de Serviços,* que tem responsabilidade de manter uma interface única para os clientes, independente da maneira como o serviço está implementado. O gestor é responsável por executar a ligação (*Binding*) entre as aplicações cliente e os serviços.

Os serviços no XMLTalk são totalmente dinâmicos, devido ao mecanismo de disponibilização automática que pode disponibilizar e remover serviços em tempo de execução. A execução dos serviços pode ser programada de forma declarativa utilizando um documento XML (*Sitemap),* o que simplifica a construção e manutenção das aplicações. O *Sitemap* descreve as seqüências de serviços e seus respectivos parâmetros através do conceito de *pipeline de execução*.

O XMLTalk provê ainda suporte para que distintas aplicações tenham cada uma um esquema de segurança próprio, e com acesso diferenciado aos serviços. Além disso, o esquema de segurança possibilita o uso de autenticação com *Single Sign-On,* deixando para os serviços a responsabilidade da autorização aos recursos. Finalmente, o XMLTalk possui um mecanismo de *cache* das informações para obter um melhor desempenho dos sistemas construídos.

Apesar da já existirem implementações de *frameworks* com suporte adequado para a criação de aplicações orientadas a serviços, o XMLTalk foi concebido para acrescer algumas das características listadas no capítulo 2 aos servidores de aplicação J2EE. Servidores de aplicação J2EE, pelo seu modelo de programação, já suportam características como reuso caixa-preta e distribuição. A motivação da construção do XMLTalk é adicionar a estas características já existentes outras características como dinamismo e adaptabilidade, heterogeneidade ambiental, composição e coordenação. É importante ressaltar que o uso dessas características depende da aplicação construída. O objetivo é suportar a arquitetura orientada a serviços de maneira que seja simples incorporar tais características nas aplicações construídas com o *framework*.

Nas seções posteriores, cada uma das funcionalidades fornecidas pelo XMLTalk será detalhada e cada conceito envolvido será mais bem explicado.

4.1. Gestor de Serviços

O *Gestor de Serviços* é um conceito central no XMLTalk. Ele tem a responsabilidade de receber as requisições para os serviços e repassá-las de forma transparente para os clientes. O Gestor de Serviços é o único ponto de acesso entre os clientes e os serviços, e foi inspirado no padrão de projeto *BusinessDelegate* [Sun Blueprints, 2003]. Por questões de segurança, a maioria das corporações que utilizam sistemas distribuídos, exige que seja possível ter um controle extensivo para que, se necessário, seja possível monitorar o tráfego de informações ou até mesmo retirar o sistema do ar em caso de emergência. Por este motivo, no XMLTalk optou-se por ter uma entidade centralizadora por onde passam todas requisições de serviços e suas informações, diferentemente de outros *framworks* para criação de sistemas distribuídos.

O único protocolo que deve ser entendido pelo cliente é o que faz a ligação entre ele e o Gestor. Esse protocolo deve ser bem definido, porém não é necessariamente único. Isso significa dizer que distintas implementações de protocolos podem ser criadas, aumentando a quantidade de componentes aptos a se comunicar com o Gestor. A Figura 8 mostra o processo de uma requisição.

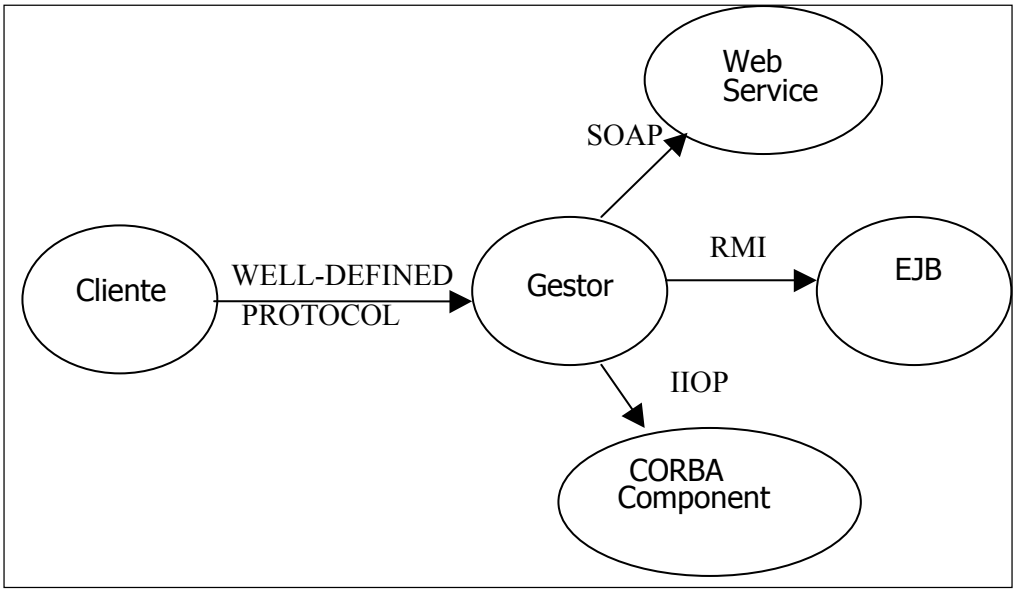


Figura 8 – Processo de uma requisição

No *framework,* o Gestor de serviços é implementado por um EJB do tipo *Session Stateless.* O protocolo pelo qual os clientes devem se comunicar com o Gestor de serviços é o protocolo padrão de EJB, *RMI Over IIOP (RMI-IIOP)*. Como o XMLTalk foi construído primordialmente para ser uma extensão dos servidores de aplicações J2EE, nada mais natural que implementar seu componente central como um EJB para que possa se fazer uso total das funcionalidades do servidor.

O Gestor também se responsabiliza por cuidar do registro de novos serviços e de aplicações. O mecanismo de disponibilização automática informa ao gestor, sempre que necessário, sobre a existência de um novo serviço ou uma nova aplicação que devem ser registrados.

Basicamente, o Gestor de serviços possui como interface remota os métodos listados no Quadro 11, que se encarregam do registro de novos serviços e aplicações. A interface descreve também o método que cuida de invocar o Gestor, passando como parâmetro o *pipeline de execução*.

```
public interface ServicesCoordinatorRemote extends EJBObject
{
  // Execução do pipeline
  Output execute(String aplId, byte[] ticket,
      IPipeline pipeline, RequestParameters requestParameters);
```

```
  // Registro de Serviços
  void registerService(IRegistar registar);
  ArrayList getRegisteredServices();
  void unregisterService(String serviceId);
  IRegistar getRegistar(String serviceId);
  // Registro de Aplicações
  void registerApplication(IApplicationDescriptor applDD);
  ArrayList getRegisteredApplications();
  void unregisterApplication(String applId);
  IApplicationDescriptor getApplDD(String applId);
}
```

Quadro 11 – Interface remota do gestor de serviços (*ServicesCoordinator*)

Como o gestor de serviços é o componente central da arquitetura proposta, ele interage diretamente com algumas outras entidades importantes para o *framework,* como o registro e o localizador de serviços. Pela interface descrita no quadro acima é possível identificar algumas dessas interações. A Figura 9 fornece uma visão geral dessas interações com as outras entidades do *framework.*

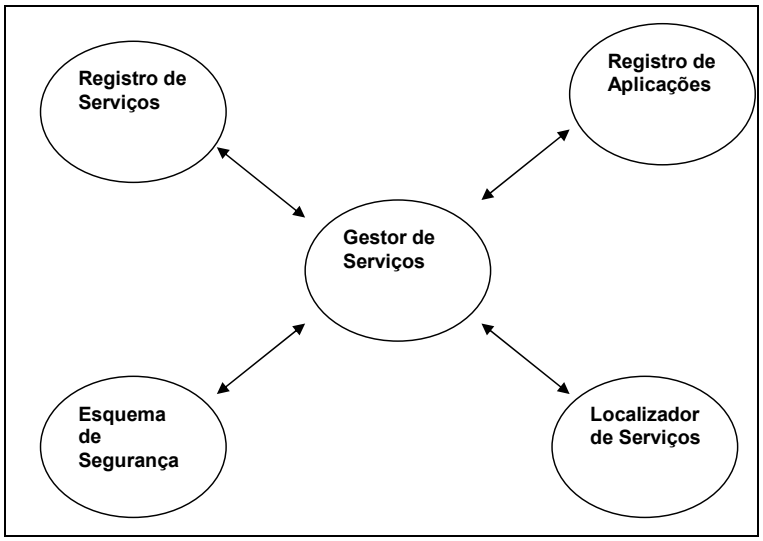


Figura 9 – Relacionamentos do Gestor de Serviços

Como o conceito de Gestor não é amarrado a um protocolo de comunicação especifico, é possível que, em trabalhos futuros, seja implementada uma comunicação entre cliente e Gestor através de SOAP, fazendo com que o

XMLTalk possa ser acessado de qualquer plataforma. A especificação J2EE 1.4 já prevê que EJBs possam ser acessados através de SOAP, e tal acesso é de responsabilidade do servidor. Com isso, o XMLTalk poderá ser acessado por clientes distintos e não só por Java (e CORBA via IIOP) como é atualmente.

4.2. Pedidos de Serviços

Para que um serviço seja executado, é necessário que um *pedido de serviço* seja criado e enviado ao gestor. O gestor de serviços, por sua vez, recebe um conjunto de pedidos de serviços agrupados em um pipeline de execução, e se encarrega de executá-los. Ao final da execução, é retornado para o cliente o resultado da seqüência de operações solicitadas.

Um pedido de serviço é uma estrutura que armazena algumas informações essenciais para a execução de um determinado serviço. A interface apresentada pelo Quadro 12 descreve os métodos que devem ser implementados para se obter uma classe referente a um pedido de serviço.

```
public interface IServiceRequest extends java.io.Serializable
{
  public String getServiceId();
  public String getProperty(String name);
  public void setProperty(String name, String value);
  public Map getProperties();
  public IServiceRequest getCopy();
  public boolean needsContext();
}
```

Quadro 12 – Interface de um pedido de serviço (*IServiceRequest*)

O pedido de um serviço carrega como informação essencial um identificador (*serviceId)* que designa o serviço a ser executado. Além disso, possui uma série de propriedades que podem ser atribuídas livremente. Tipicamente as propriedades dizem respeito a como o serviço deve se comportar de acordo com a aplicação sendo executada, ou ainda, de acordo com o contexto apropriado para o serviço. Um dos parâmetros que um serviço sempre recebe ao

ser invocado é o pedido de serviço que deu origem a sua execução; a partir deste, é possível que ele altere o seu comportamento de acordo com as propriedades atribuídas.

As diversas implementações dos pedidos de serviços possuem algumas diferenças semânticas, principalmente de acordo com o tipo de tarefa que o serviço irá executar. Por exemplo, existem pedidos que precisam do contexto de dados (seção 4.8) para serem executados (como por exemplo o *TransformerServiceRequest*), outros não (*GenerateServiceRequest*). Já pedidos que forem de serialização de dados (*SerializerServiceRequest*), necessariamente determinam o final do *pipeline* de execução. A Figura 10 a seguir mostra a hierarquia de classes proposta para os pedidos de serviços.

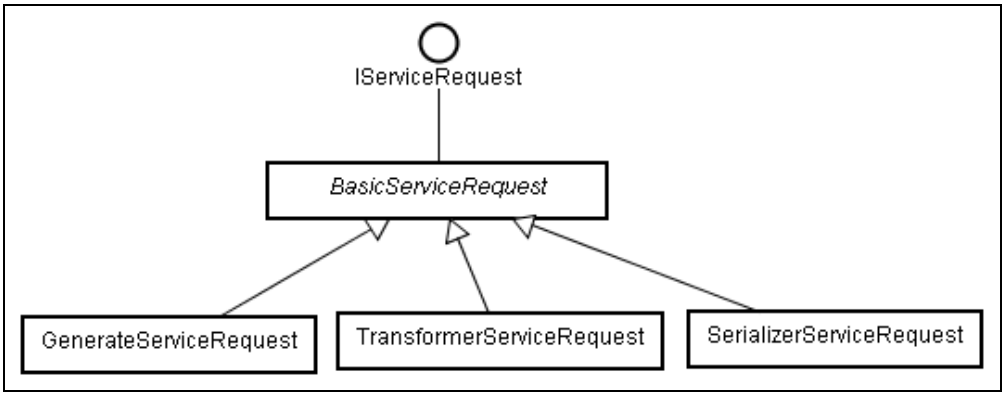
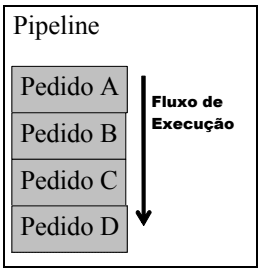


Figura 10– Hierarquia dos pedidos de serviço

4.3. *Pipeline de Execução*

Os pedidos de serviços são enviados para o Gestor agrupados em um *pipeline de execução*. O pipeline de execução é implementado como uma fila de execução onde os pedidos de serviços ficam armazenados em uma lista encadeada que é percorrida como uma fila (*FIFO*). Ao receber o *pipeline*, o Gestor o percorre invocando os serviços de acordo com os pedidos que estiverem contidos. Um *pipeline* também pode conter outros *pipelines* em sua estrutura, para que desvios na seqüência de execuções possam ser incluídos. Dessa forma é possível colocar componentes que dinamizem a execução dos serviços criando ramos (*branchs*) como *ifs* e *switchs*. A Figura 11 apresenta um desenho esquemático de um *pipeline* de execução, bem como sua interface Java.




```
public interface IPipeline extends
                          java.io.Serializable
{
  IServiceRequest next();
  boolean hasNext();
  void addServiceRequest(IServiceRequest
                              serviceRequest);
  String getSecurityLevelRequired();
  int getSize();
  void reset();
  IPipeline getCopy();
  boolean isCacheable();
}
```

Figura 11 – Representação de um *pipeline* de execução e sua Interface

A criação de um *pipeline* de execução é feita através da instanciação de objetos provenientes de classes que implementem as interfaces *IPipeline* e *IServiceRequest*. O Quadro 13 demonstra a criação de um pipeline de execução.

```
IPipeline pipeline = new DefaultPipeline();

pipeline.addServiceRequest(
      new GenerateServiceRequest("Service A"));
pipeline.addServiceRequest(
      new TransformerServiceRequest("Service B"));
pipeline.addServiceRequest(
      new SerializerServiceRequest("Service C"));
```

Quadro 13 – Exemplo da criação de um *pipeline* de execução

4.4. Registros

O XMLTalk necessita armazenar algumas informações sobre aplicações e serviços que estão disponíveis. Tais informações dizem respeito a questões como segurança das aplicações, localização dos serviços, entre outras. Essas informações ficam armazenadas em registros que devem ser concebidos de forma centralizada em um repositório que disponibilize acesso remoto. No *framework*, eles estão armazenados em tabelas *hash* que são acessadas somente pelo Gestor de serviços. Todo acesso aos registros deve ser feito por intermédio do Gestor.

As seções seguintes irão detalhar cada tipo de registro, bem como o mecanismo de disponiblização automática que auxilia na composição dinâmica dos sistemas desenvolvidos com o XMLTalk..

4.4.1. Registro de Serviços

Os serviços no XMLTalk devem ser registrados antes de serem utilizados. Para tanto, enviam ao registro, por intermédio do Gestor, algumas propriedades que tratam questões como localização e protocolo de comunicação.

Um serviço, ao ser disponibilizado, deve se comunicar diretamente com o Gestor passando suas propriedades agrupadas em um descritor de serviços (*IServiceDescriptor*). O descritor é na verdade um container para um conjunto de propriedades pré-definidas. Além disso, cada serviço deve fornecer um identificador único que passa a ser associado com o objeto de registro para futuras consultas.

O descritor de serviços pode ser implementado de diversas formas, bastando para isso implementar uma interface específica (*IServiceDescriptor*). Para facilitar o registro de serviços, foi criado um formato de arquivo XML que auxilia a criação dos descritores de serviços. Esse arquivo é denominado *SSIDL (Simple Service Interface Description Language*), cujo formato se propõe a ser uma versão altamente simplificada da WSDL descrita anteriormente. Basicamente, o SSIDL espelha a interface de descrição de serviços (*IServiceDescriptor*), onde cada nó de informação é associado diretamente a um método. O Quadro 14 exemplifica um arquivo SSIDL.

```
<service id="File">
  <location>
    <URI>FileService</URI>
    <protocol>RMI</protocol>
  </location>
  <description>Serviço de Arquivos XML fixos</description>
  <input>
    <attributes>
      <attribute name="base-url" type="string">
        <values>/c:/jir2003/content/</values>
```

```
        <description>Path base de onde serão procurados
               os arquivos</description>
      </attribute>
      <attribute name="file-name" type="string" />
    </attributes>
  </input>
  <output>
    <data-type>String</data-type>
    <data-mapper>org.xmltalk.data.StringMapper</data-mapper>
  </output>
</service>
```

Quadro 14 – Exemplo de Arquivo SSIDL

O arquivo possui basicamente o identificador único do serviço (atributo *id*) e depois três áreas: *location*, *input* e *output*.

A primeira área (*location*) diz respeito à localização do serviço e qual protocolo deve ser usado na comunicação entre o Gestor e o serviço.

Já a segunda (*input)* expõe quais atributos devem ser enviados para os serviços bem como seus valores *default* e descrição de uso. A formalização dos atributos de entrada de um serviço é totalmente opcional, podendo ser deixada em aberto. Nesse caso, cada cliente deve saber como se comunicar com o serviço. Porém, através da formalização dos atributos no descritor é possível criar clientes dinâmicos e, por exemplo, criar interfaces gráficas para testar o uso dos serviços de forma mais amigável e independente.

Finalmente, a área da saída (*output*) é obrigatória e diz respeito a que tipo de dado o serviço irá gerar. Como os serviços podem gerar qualquer tipo de dado pois sua interface, como será visto mais à frente, retorna um vetor de bytes, é imprescindível que o mapeador desses dados seja definido (na seção 4.8 será apresentado o conceito de mapeadores). Nesta seção, o tipo do dado Java também é descrito.

Em geral os arquivos de descrição de serviços SSIDL são empacotados em arquivos do tipo JAR (*Java Archive*) juntamente com a implementação do respectivo serviço. Com isso a implementação e a descrição de interface compõem um arquivo único facilitando sua manutenção. No caso do uso de *proxies*, onde uma única implementação gera vários serviços distintos (por exemplo, um *proxy* para serviços implementados em *javascript*), os arquivos

SSIDL são disponibilizados separadamente e não necessitam estar acompanhados da implementação do serviço.

Assim como em toda implementação do *framework*, o registro de serviços é altamente flexível, possibilitando criar um outro formato de arquivo para se descrever um serviço, bastando para isso se implementar a interface de descrição de serviços, criar um *deployer* especifico e ligá-lo ao mecanismo de disponibilização.

4.4.2. Registro de Aplicações

O registro de aplicações funciona de maneira muito similar ao registro de serviços. O *framework* disponibiliza uma interface que descreve as propriedades de uma aplicação (*IApplicationDescriptor*) e um leitor para um documento XML (*Application Descriptor*) que implementa esta interface. O registro de aplicações, assim como o registro de serviços, é acessado através do Gestor de serviços. As propriedades de uma aplicação dizem respeito ao nível de segurança da aplicação e a configuração de alguns diretórios. O Quadro 15 exemplifica um arquivo de descrição de aplicações.

```
<application id="site">
  <description>Site da Ideais</description>
  <sitemap>file:/C:/XMLTalk/sitemaps/site.sitemap</sitemap>
  <static-file-dir>file:/C:/static</static-file-dir>
  <security>
    <user-factory>
        org.xmltalk.security.basic.BasicUserFactory
    </user-factory>
    <ticket-factory>
        org.xmltalk.security.basic.BasicTicketFactory
    </ticket-factory>
    <default-security-level>1</default-security-level>
  </security>
</application>
```

Quadro 15 – Exemplo de um arquivo descritor de aplicações

Diferentemente do descritor de serviços, o descritor de aplicações não deve ser registrado em conjunto com alguma implementação especifica. Tal arquivo é disponibilizado isoladamente sem a necessidade de estar empacotado em um JAR.

Como os serviços, as aplicações também devem possuir um identificador único que é fornecido pelo descritor da aplicação para que se possa fazer uma referência futura.

4.4.3. Mecanismo de disponibilização automática

No XMLTalk é possível que os serviços sejam disponibilizados e registrados automaticamente. Isso é possível devido ao mecanismo de disponiblização automática. Este mecanismo monitora alguns diretórios do servidor, a espera de arquivos específicos definidos pelas suas extensões (ex. .jar). Ao receber esses arquivos, ele dispara um *deployer* que se encarrega de disponibilizar e registrar o serviço, se comunicando diretamente com o Gestor de serviços.

Como a especificação J2EE não padroniza o mecanismo de disponibilização para todos servidores de aplicação, é necessária que a implementação permita a criação de novos *deployers* que possam ser incorporados ao XMLTalk. Atualmente, o *framework* está apto a executar a disponibilização automática em servidores de aplicação *JBoss* [JBoss, 2003]; porém, para que outros servidores possam se beneficiar do mecanismo de disponibilização automática, basta que se implemente a interface descrita no Quadro 16.

```
public interface IServerDeployer
{
    public void deploy(URL jarFile) throws DeployException;
    public void undeploy(URL jarFiLe) throws DeployException;
}
```

Quadro 16 – Interface de disponibilização automática

4.5. Sitemap

O *sitemap* é o mecanismo que proporciona um maior dinamismo e reduz significativamente a complexidade da elaboração de aplicações que utilizem o XMLTalk. Na prática ele é implementado como um documento XML que descreve um ou mais *pipelines* de execução. O formato de arquivo implementado para este trabalho é baseado no formato utilizado no *framework* Apache Coocon [Apache Cocoon, 2003].

Através dos arquivos que descrevem os *pipelines*, é possível criar aplicações de forma declarativa e alterá-las em tempo de execução. Para melhor exemplificar, o Quadro 17 exibe um trecho de um arquivo XML de um *sitemap*.

```
<matcher pattern="/bookSearch" cacheable="no">
<generate service="BookSearch"/>
      <transformer service="XSLTransformer"
                   file-name="bookResults.xsl"/>
      <serializer service="StringSerializer" type="html"/>
</matcher>
```

Quadro 17 – Trecho de um arquivo de sitemap

O trecho apresentado no Quadro 17, durante a inicialização da aplicação, é traduzido para um descritor de *pipeline* de execução, com chamadas a três serviços distintos (*BookSearch*, *XSLTransformer* e *StringSerializer*). Os atributos XML de cada nó que representam pedidos de serviço passam a ser propriedades dos pedidos (atribuídas com o método *setProperty* da interface *IServiceRequest*). Além disso, cada serviço é invocado com uma implementação diferente do pedido de serviço (*IServiceRequest*), que carrega uma informação semântica distinta (vide seção 4.2). O Quadro 18 mostra a relação entre os nós XMLs e os pedidos de serviço específicos:

```
<generate />     -  GenerateServiceRequest
<transformer />  -  TransformerServiceRequest
<serializer />   -  SerializerServiceRequest
```

Quadro 18 – Relação entre nó XML e classe que implementa o pedido de serviço.

Os arquivos de *sitemaps* são monitorados pelo framework e caso sejam alterados em tempo de execução da aplicação, a tradução para o *pipeline* é automaticamente refeita.

O nó *matcher*, que envolve o *pipeline* de execução, é traduzido para uma estrutura que armazena um *pipeline* e é unicamente identificada em pelo atributo *pattern*. O *framework* recebe requisições, por exemplo um requisição HTTP, e extrai do URL requerido a parte relativa a identificação do *pipeline*. Por exemplo, se o URL for http://localhost/apl/teste.html, a identificação usada para encontrar o *pipeline* requerido é teste.html. Esta identificação é então usada pelo *sitemap* para selecionar, através de um casamento de padrão (*pattern* matching) entre o atributo *pattern* e a identificação extraída da URL o *pipeline* correspondente. Além disso, no *matcher* existem algumas propriedades comuns a todos elementos do *pipeline*, como por exemplo, se o resultado de sua execução deve ou não ser armazenado em *cache*.

Os *patterns* podem conter em sua definição caracteres curingas (nesta implementação foi usado o caractere interrogação) para serem substituídos durante a execução por uma outra seqüência de caracteres qualquer. O trecho de *sitemap* mostrado no Quadro 19 cria um documento HTML através da transformação de um arquivo XML dinâmico (dependente da requisição) e um documento XSL.

```
<matcher pattern="/?.html" cacheable="no">
    <generate service="File" file-name="{1}.xml" />
    <transformer service="XSLTransformer"
                 file-name="transformer.xsl"/>
    <serializer service="StringSerializer" type="html"/>
</matcher>
```

Quadro 19 – Trecho de sitemap com caracter curinga

Cada caractere curinga é indexado (seqüência de números inteiros iniciada em um), e a seqüência de caracteres que casou com o padrão do atributo *pattern* pode ser usada, através desse índice, para o preenchimento dos atributos enviados aos serviços. No exemplo introduzido no Quadro 19, se a string com o pedido fosse ‘index.html’, o atributo *file-name* do serviço *File* seria automaticamente ‘index.xml’. É possível que existam quantos caracteres curingas sejam necessários, desde que não estejam em seqüência direta em um mesmo padrão.

O mecanismo de inclusão de caracteres curinga facilita a criação de *pipelines,* pois reduz o trabalho de replicar seqüências de pedidos que executem tarefas similares, se diferenciando apenas por alguns atributos dos pedidos de serviço.

Finalmente, é importante ressaltar que o *sitemap* é um conceito, e não somente um formato de arquivo. A arquitetura em torno do *sitemap* foi implementada de maneira que outros formatos de descritores de *pipelines* possam ser criados e incluídos como parte integrante do framework, bastando para isso implementar a interface *ISitemap* mostrada no Quadro 20.

```
public interface ISitemap extends Serializable {
  public void load() throws SitemapException;
  public boolean update() throws SitemapException;
  public IMatcher getMatcher(String pattern);
  public IApplicationDescriptor getApplication();
  public List getMatchers();
}
```

Quadro 20 – Interface *ISitemap*

4.6. Serviços e *Proxies* para serviços

No XMLTalk os serviços podem ser implementados de formas distintas, mas sempre se comunicando diretamente com o Gestor. Na prática, se um serviço é implementado por um componente que não se comunica pelo protocolo RMI-IIOP, então deve existir um *proxy* (representante local do serviço que se comunica usando o protocolo definido) que executa a tarefa de dialogar diretamente com o Gestor.

Todos serviços devem implementar uma interface (*IService*) que possui apenas um método. O Quadro 21 apresenta esta interface.

```
public interface IService
{
  public byte[] execute(ITicket ticket,
                        IServiceRequest serviceRequest,
                        RequestParameters requestParameters,
                        DataContext ctx)
     throws RemoteException, ServiceException;
```

```
}
```

Quadro 21 – Interface de um serviço

A único método da interface (*execute*) é, propositalmente, o mais genérico possível, e por isso possui como valor de retorno um vetor de bytes. Com isso os serviços podem retornar qualquer tipo de dado que seja serializável em um vetor de bytes.

Já os parâmetros de entrada são bem definidos e importantes para a construção dos serviços. Devido à importância do método *execute* para o funcionamento do XMLTalk, os parâmetros estão descritos a seguir:

- ticket – Objeto relacionado ao esquema de segurança. Com esse objeto é possível realizar uma autorização aos recursos específicos de cada serviço. Quando um serviço é invocado, o usuário do serviço já foi devidamente autenticado pelo esquema de segurança, como será visto na seção 4.7, restando apenas que o serviço cuide da autorização aos recursos.
- serviceRequest – Representa o pedido de serviço que gerou a execução do serviço, como descrito na seção 4.2..
- requestParameters – Este objeto contém os parâmetros específicos da requisição sendo feita no momento. No caso de uma chamada http, por exemplo, este objeto contém os parâmetros de *POST* ou do *GET* enviados ao servidor http.
- ctx – Contexto de dados. Este objeto contém todos os dados que foram gerados dentro de uma mesma requisição de um *pipeline*. Tipicamente os serviços são independentes e recebem valor nulo nesse parâmetro. Porém em alguns casos, como por exemplo *logs* e transformadores, esse objeto é enviado como parâmetro. O contexto de dados será mais bem detalhado na seção 4.8.1.

Os serviços que se comunicam diretamente com o gestor implementam diretamente esta interface e são EJBs do tipo *Session Bean Stateless*. Já os serviços que são acessados através de um *proxy* podem ser implementados de qualquer maneira, sem a necessidade de implementar essa interface. Nesse caso o

proxy é obrigado a implementar esta interface. Com isso o acesso dos clientes aos serviços se dá de maneira transparente, como é o objetivo do *framework*.

4.7. Segurança

A segurança é um fator muito importante para aplicações no mundo real, principalmente em sistemas distribuídos. Sistemas distribuídos devem trafegar seus dados através de redes de comunicação não necessariamente seguras, tipicamente utilizando um protocolo como TCP/IP que é sabido não ser seguro [Bellovin, 1989]. Em geral aplicações distribuídas devem implementar a sua própria camada de segurança visando proteger a integridade de seus dados e evitando que estes possam ser rastreados através da rede. No caso de sistemas onde os dados enviados atravessam as redes locais (LANs) atingindo as redes de longa distância (WANs), o problema se torna ainda mais grave, pois é possível que terceiros tenham acesso aos pacotes contendo os dados.

O XMLTalk fornece uma API de segurança que pode ser usada para aplicações com finalidades distintas. Algumas aplicações não necessitam de alta segurança e preferem não ser penalizadas pelo uso de complicados mecanismos, enquanto outras são obrigatoriamente seguras. Com a API criada é possível resolver de forma simplificada os dois casos. No projeto da API também foi levado em consideração que todos componentes do *framework* não possuem estado de sessão, ficando assim impedidos de armazenar alguma informação sobre o cliente que esteja acessando os serviços. Tal incumbência fica a cargo dos próprios clientes como será mais bem detalhado na seção 4.7.1.1.

Outro fator importante que foi considerado durante o projeto do esquema de segurança do XMLTalk é o fato que cada aplicação possui um usuário que pode ser descrito com um conjunto de dados distinto. Uma certa aplicação pode considerar que um usuário possui endereço e telefone e uma outra pode precisar apenas do CPF. Para todas só é necessário que tenham ao menos uma identificação única e uma senha, tipicamente um login e uma senha. O provedor desses dados também é livre, podendo ser desde um banco de dados passando por um serviço de diretórios (LDAP) ou até mesmo um documento XML.

4.7.1. API

A API de segurança é constituída por quatro interfaces Java que as aplicações devem implementar. O XMLTalk já vem com algumas implementações básicas prontas que podem ser usadas em distintos tipos de aplicações, porém para sistemas mas específicos uma ou mais dessas interfaces devem ser implementadas. A Figura 12 apresenta as interfaces que compõem a API e as implementações existentes.

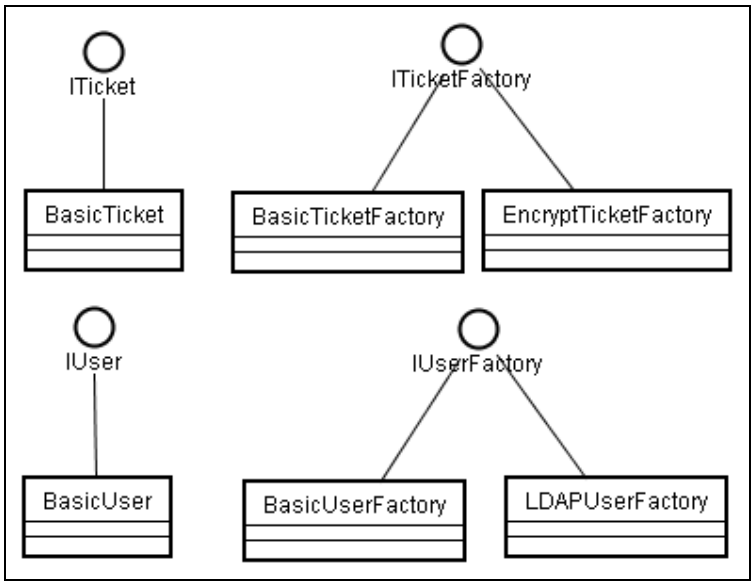


Figura 12– Interfaces e classes da API de segurança

4.7.1.1. Tíquetes

O conceito fundamental desta API é baseado nos tíquetes *kerberos* [Neuman et al, 1994]. O cliente deve efetuar um login no servidor para receber como resultado deste login um tíquete que deve ser reapresentado nas requisições subseqüentes. Com base nas informações contidas neste tíquete é que serviços autorizam ou não o acesso a determinados recursos. Durante a conversação, o servidor não armazena informação alguma sobre o cliente e seu estado na conversação. Informações desse tipo como, por exemplo, se o cliente está logado ou não, ficam armazenadas no tíquete enviado ao cliente.

A interface *ITicket* , que é introduzida no Quadro 22, representa um tíquete do XMLTalk. Ela possui alguns métodos responsáveis por armazenar informações sobre o tíquete que poderão ser utilizadas na autorização posterior. Essas informações vão desde o usuário que conseguiu acesso ao ticket até a hora em que o tíquete foi expedido.

```
public interface ITicket extends Serializable{
    public String getIpAddress();
    public Date getIssuedTime();
    public String getSecurityLevelGranted();
    public IUser getUser();
    public String getAplId();
    public Serializable getAttribute(String name);
    public void setAttribute(String name, Serializable obj);
}
```

Quadro 22 – Interface de um tíquete

O XMLTalk já vem com uma implementação desta interface para ser usada nas aplicações denominada *BasicTicket*, que simplesmente espelha diretamente os métodos da interface. Na grande maioria das aplicações esta classe se mostra suficiente.

Além da interface *ITicket*, existe também a *ITicketFactory*, apresentada no Quadro 23, que é a entidade responsável por emitir os tíquetes. A fábrica de tíquetes, como são chamadas as classes que implementam tal interface, também tem a importante função de verificar se um determinado nível de segurança requerido pelo recurso é compatível com o nível de segurança obtido pelo tíquete. Um exemplo disso é se uma determinada aplicação utilizar números inteiros para se referir a uma hierarquia de acesso aos recursos. Então, um recurso de nível 3 precisa de pelo menos um acesso nível 3 para poder ser acessado. A responsabilidade de tal verificação é da fábrica de tíquetes, que somente deve responder se um determinado nível pode ter acesso ao recurso ou não, retornando um booleano. Em uma outra aplicação, o acesso ao recurso pode requerer uma consulta a um banco de dados que mapeia as funções de cada usuário com o acesso aos recursos requeridos. Com isso a semântica de autorização aos recursos fica totalmente genérica podendo ser modificada de acordo com a aplicação. Os

níveis de segurança são mapeados em strings, mas as fábricas podem usar essas strings da maneira que lhes convir.

Além disso, a fábrica de tíquetes tem a responsabilidade de criptografar os tíquetes que serão emitidos por ela. Se por exemplo uma aplicação necessitar alta segurança, é possível que os tíquetes sejam criptografados, antes de serem enviados aos clientes para que armazenem localmente. Assim os clientes não conseguiriam forjar o conteúdo do tíquete de forma alguma, utilizando-o apenas como objeto de acesso ao gestor de serviços e aos serviços requisitados.

```
public interface ITicketFactory{
    public ITicket createTicket(IUser user, String applId,
                      Map otherFields);
    public byte[] encryptTicket(ITicket ticket)
                    throws TicketFactoryException;
    public ITicket decryptTicket(byte[] encryptedData)
                    throws TicketFactoryException;
    public boolean checkSecurityLevel(String n1, String n2);
    public void setProperties(Map properties);
}
```

Quadro 23 – Interface de uma fábrica de tíquetes

Existem, como parte integrante do *framework*, duas implementações básicas da fábrica de tíquetes. Uma delas, denominada *BasicTicketFactory*, não criptografa o tíquete (apenas o serializa para um vetor de bytes) e libera o acesso a todo e qualquer recurso requerido. É usada em aplicações que não requerem segurança. A outra, *EncryptTicketFactory*, como o próprio nome diz, criptografa os tíquetes usando o algoritmo DES com uma chave privada fixa, e utiliza números inteiros para hierarquizar o acesso aos recursos. Para outras aplicações é possível que seja necessária a criação de outras fábricas como, por exemplo, uma que utiliza as permissões de um banco de dados para gerenciar o acesso aos recursos.

4.7.1.2. Usuários

No XMLTalk um cliente ou usuário é representado por uma interface básica que expõe alguns dos seus atributos fundamentais como login e função (*role*). A extensão desta interface para o uso de outras propriedades é totalmente livre, ficando a cargo do desenvolvedor da aplicação. Assim como a interface *ITicket, IUser* que está descrita pelo Quadro 24, vem com uma implementação básica que espelha a interface denominada *BasicUser* e deve ser usada em sistemas onde as informações detalhadas sobre usuário não são essenciais.

```
public interface IUser extends java.io.Serializable{
    public String getLogin();
    public String getRoleId();
    public void setRoleId(String roleId);
    public boolean getIsLogged();
    public void setIsLogged(boolean isLogged);
    public String getWorkgroupId();
    public void setWorkgroupId(String workgroupId);
}
```

Quadro 24 – Interface de um usuário

Como no caso dos tíquetes, os usuários devem ser criados através de uma fábrica de usuários, que são classes que implementam a interface *IUserFactory*, mostrada pelo Quadro 25. Uma fábrica de usuários tem como responsabilidade principal executar o processo de autenticação do usuário. A autenticação pode ser feita através de mecanismos distintos como um serviço de diretórios LDAP, um banco de dados ou até uma lista de usuários escrita diretamente no código; o importante é que a fábrica deve cuidar de validar ou não a seqüência de bytes enviada como senha no método *doLogin*.

```
public interface IUserFactory{
    public IUser getUserInstance();
    public IUser doLogin(String login, byte[] password);
    public void setProperties(Map properties);
}
```

Quadro 25 - Interface de uma fábrica de usuários

O XMLTalk fornece duas classes que implementam a fábrica de usuários. A *BasicUserFactory* instancia um usuário sem nenhuma informação e o considera sempre autenticado (logado). Deve ser usada para aplicações sem segurança. A outra fábrica, denominada *LDAPUserFactory,* trata de buscar e validar usuários em um serviço de diretórios LDAP.

4.7.1.3. O Processo de Login

Na prática o processo completo para uma requisição é feito em 5 passos. O diagrama de seqüência apresentado pela Figura 13 mostra com funciona o procedimento de login e uma chamada subseqüente para execução de um *pipeline* enviada ao gestor de serviços.

Cabe ressaltar que este processo não é feito a cada requisição feita ao gestor. Como o cliente recebe o ticket como resultado da operação inicial ele pode e deve armazená-lo para requisições subseqüentes. Além disso, o gestor de serviços utiliza um esquema de cache das fábricas para que não tenha que instanciar uma nova fábrica a cada processo de login (aplicações orientadas a serviços tipicamente têm muitos usuários).

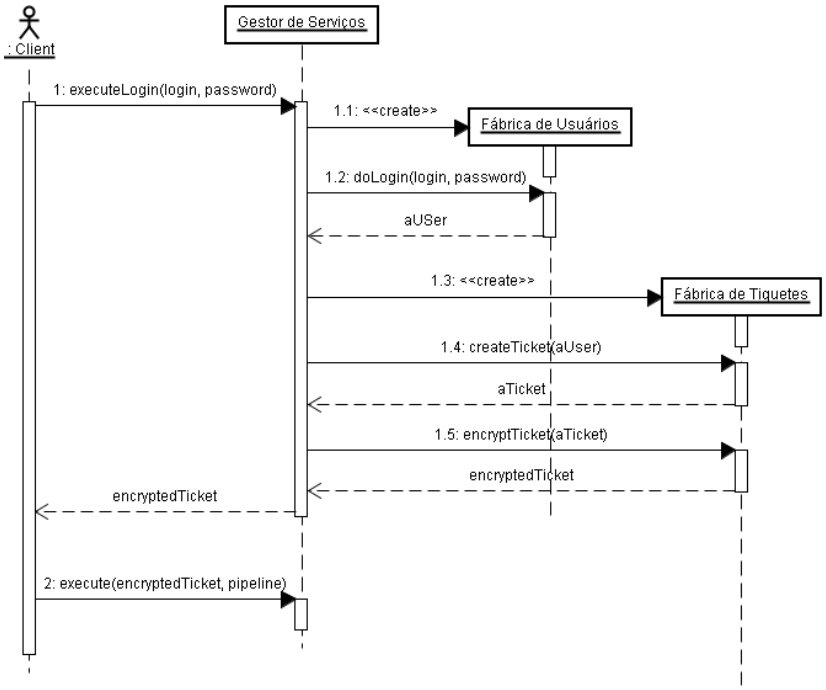


Figura 13 – Diagrama de seqüência para o processo de login

4.7.2. Configuração

A configuração de todo esquema de segurança é feita por aplicação, diretamente nos documentos XML que descrevem cada aplicação. As classes que implementam as fábricas de tíquetes e de usuários, devem estar descritas nesse documento para que possam ser devidamente instanciadas pelo Gestor de serviços. Esse arquivo já foi descrito anteriormente, mas o trecho relativo a segurança de uma aplicação é repetido pelo Quadro 26 para facilitar o entendimento.

```
<security>
  <user-factory>
      org.xmltalk.security.basic.BasicUserFactory
  </user-factory>
  <ticket-factory>
      org.xmltalk.security.basic.BasicTicketFactory
  </ticket-factory>
  <default-security-level>1</default-security-level>
</security>
```

Quadro 26 – Configuração do esquema de segurança

É importante ressaltar que existe o atributo denominado *default-security-level* que ainda não foi citado. Este atributo diz respeito ao nível de segurança *default* usado pelos *pipelines* de execução quando nenhum nível for explicitado, dizendo para o gestor de serviços, e conseqüentemente para a fabrica de tíquetes, que o cliente precisa ter acesso a pelo menos este nível de segurança para poder executar o *pipeline*

4.8. Outras funcionalidades

Esta seção irá descrever outras funcionalidades que o XMLTalk provê que não são tão essenciais para o funcionamento das aplicações. Estas outras funcionalidades serão descritas brevemente sem explicar muito detalhadamente cada uma delas.

4.8.1. Contexto de Dados

Todos dados gerados pelos serviços envolvidos em uma aplicação no XMLTalk são armazenados em um contexto de dados, que é válido para cada requisição. Esse contexto é enviado para serviços que necessitem dos dados gerados para funcionar, como transformadores e serializadores. O contexto é implementado como um vetor de objetos que armazenam informações como o dado que foi gerado pelo serviço, o identificador do serviço que gerou os dados e um mapa de atributos livres.

O contexto de dados é importante pois agrupa todos os dados gerados pelos serviços, de forma que estes possam ser usados mais tarde por outros serviços. Além disso, o contexto é válido por *Thread* de requisição, fazendo com que possam existir vários contextos simultâneos sendo um para cada cliente que requisitou a execução dos serviços.

4.8.2. Mapeadores de Dados

Todos os dados gerados pelos serviços do *framework* são vetores de bytes sem nenhuma representação semântica da informação contida nestes dados. Porém, existem alguns serviços, principalmente transformadores e serializadores, que necessitam ter uma representação mais concreta da informação para que possam exercer sua função. Para isso, cada serviço deve informar o seu mapeador de dados.

Um *mapeador de dados* é uma classe responsável por receber os dados em forma de um vetor de bytes e transformá-los em uma representação menos genérica como strings XML ou objetos Java, para que possam ser adequadamente usados.

Como os mapeadores são configurados por serviço, eles sabem exatamente qual o tipo de dado está contido no vetor de bytes que estão recebendo como argumento, e com isso conseguem reconstruí-lo para um formato inteligível por parte dos transformadores ou serializadores. A configuração dos mapeadores é

feita nos arquivos descritores de serviços (SSIDL) no trecho que descreve a saída de um serviço

O Quadro 27 configura um serviço para utilizar o mapeador de dados *StringMapper,* que trata o vetor de bytes retornado pelo serviço como uma string de caracteres.

```
<output>
   <data-type>String</data-type>
   <data-mapper>org.xmltalk.data.StringMapper</data-mapper>
 </output>
```

Quadro 27 – Configuração dos mapeadores de dados

4.8.3. Cache

Para que as aplicações possam obter um melhor desempenho, existe um mecanismo de cache implementado no XMLTalk. Esse mecanismo foi desenhado para não ficar na camada de negócios (tipicamente na camada Web), para que possa ser evitada a chamada remota para o Gestor de serviços O uso do mecanismo de cache aumenta significativamente o desempenho. Toda vez que o framework recebe uma requisição que seja passível de ser armazenada em cache, ele não executa os serviços, devolvendo imediatamente o resultado previamente armazenado. A indicação se um determinado *pipeline* é ou não passível de ser armazenado pelo mecanismo de cache é feita diretamente no sitemap da aplicação.

O mecanismo de cache é configurável podendo ter possibilidades distintas, como cache em disco, em memória ou ainda em banco de dados. Para se incorporar um novo mecanismo basta implementar uma interface específica.

O *framework* possui atualmente uma implementação de cache em memória (*RAMCacheSystem)* que utiliza o algoritmo LRU (*Last Recently Used*) para sua política de substituição de entradas no cache.

4.8.4. Instrumentação

Para que se tenha total controle das configurações do *framework* e uma instrumentação das partes importantes das aplicações, o *XMLTalk* prove uma interface com o mundo externo baseado na especificação JMX (*Java Management Extensions*) [JCP, 2003] da Sun. Através de um console JMX, é possível alterar configurações, como os diretórios que são monitorados pelos mecanismos de disponibilização automática, bem como instrumentar o *framework* para obter informações sobre os registros de serviços e de aplicações.