UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DIRETORIA DE ENSINO E DUCAÇÃO PROFISSIONAL
CURSO SUPERIOR DE TECNOLOGIA EM DESENVOLVIMENTO
DE SISTEMAS DE INFORMAÇÃO

FERNANDO HENRIQUE DOS SANTOS GOMES

COMUNICAÇÃO BIDIRECIONAL ENTRE CLIENTE E SERVIDOR *WEB* COM AJAX REVERSO

TRABALHO DE DIPLOMAÇÃO

MEDIANEIRA
2011

FERNANDO HENRIQUE DOS SANTOS GOMES

COMUNICAÇÃO BIDIRECIONAL ENTRE CLIENTE E SERVIDOR *WEB* COM AJAX REVERSO

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia em Desenvolvimento de Sistemas de Informação CSTDSI – da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: *M.Sc*. Prof. Fernando Schütz.

MEDIANEIRA

2011

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Diretoria de Graduação e Educação Profissional
Coordenação do Curso Superior de Tecnologia em Desenvolvimento de Sistemas de Informação

TERMO DE APROVAÇÃO

Comunicação Bidirecional entre Cliente e Servidor *WEB* com Ajax Reverso

Por

Fernando Henrique dos Santos Gomes

Este Trabalho de Diplomação foi apresentado às 16:30h do dia 14 de junho de 2011 como requisito parcial para a obtenção do título de Tecnólogo no Curso de Graduação em Desenvolvimento de Sistemas de Informação, da Universidade Tecnológica Federal do Paraná (UTFPR). O acadêmico foi argüido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

Prof. *M.Sc* Fernando Schütz UTFPR – *Campus* Medianeira (Orientador)	Prof. Márcio Angelo Matté UTFPR – *Campus* Medianeira (Convidado)
Prof. *M.Sc*. Everton Coimbra de Araújo UTFPR – *Campus* Medianeira (Convidado)	Prof. Juliano Rodrigo Lamb UTFPR – *Campus* Medianeira (Responsável pelas atividades de TCC)

A folha de aprovação assinada encontra-se na Coordenação do Curso

Dedico este trabalho à minha namorada Cristina Huller,
aos meus pais, Adilson e Eliane Gomes, e a todos que,
de alguma forma, contribuíram para o seu desenvolvimento
e conclusão.

AGRADECIMENTOS

À Deus pelo dom da vida, pela fé e perseverança para vencer os obstáculos.

Aos meus pais, pela orientação, dedicação e incentivo desde o início até esta última fase do curso e durante toda minha vida.

À minha namorada que esteve sempre presente, me incentivando e dando forças para o termino das atividades do curso.

Ao meu professor orientador, que disponibilizou seu tempo e paciência para me orientar, que mesmo sem ter lecionado nas matérias que cursei me recebeu prontamente colocando-se a disposição mesmo em horário extra oficiais.

Agradeço também aos meus colegas de faculdade e de trabalho que acompanharam minha jornada. Em especial gostaria de agradecer aos meus chefes Nérison Leonhart e Márcio Luiz Scharam, diretores e sócios da Digitaldoc, pela compreensão e apoio.

Enfim, a todos que de alguma forma, direta ou indiretamente contribuíram para a realização deste trabalho, meu muito obrigado.

“Qualquer tecnologia suficientemente avançada é indistinta de magia”.

(Arthur C. Clarke)

RESUMO

GOMES, Fernando Henrique dos Santos. Comunicação Bidirecional entre Cliente e Server *WEB* com Ajax Reverso. 2011. 52 páginas. Trabalho de Conclusão de Curso (Graduação em Desenvolvimento de Sistemas). Universidade Tecnológica Federal do Paraná, Medianeira, 2011.

Este trabalho teve por objetivo desenvolver um estudo bibliográfico sobre técnicas e tecnologias que permitam a implementação do Ajax Reverso para comunicação bidirecional entre cliente e servidor em aplicações *WEB.* O desenvolvimento da tecnologia *WEB* está relacionado, entre outros fatores, a necessidade de simplificar a atualização e manutenção da aplicação, porém, algumas aplicações necessitam de atualização constante de informações como um *chat* ou pregão eletrônico e essa atualização constante, sem a requisição explicita do usuário pode ser provida utilizando um recurso denominado Ajax Reverso, também conhecido como "Tecnologia *Push*". Ao decorrer do trabalho são apresentadas as tecnologias e ferramentas utilizadas, bem como a explicação para cada uma das três técnicas Ajax Reverso. Foram estudadas duas possibilidades de tecnologias para implementar uma das técnicas em um exemplo, a nova versão da plataforma Java *WEB* – JEE6 (*Java Enterprise Edition 6*) e o *framework* Ajax DWR (*Direct WEB Remoting)*. Foi escolhido como exemplo uma aplicação de *chat*, onde, quando um usuário envia uma mensagem, todos os outros usuários conectados devem recebê-la.

Palavras-chave: *Ajax*, *comet*, reverso

ABSTRACT

GOMES, Fernando Henrique dos Santos. Bidirectional Communication Betwenn Client and Server *WEB* whit Reverse Ajax. 2011. 52 pages. Trabalho de Conclusão de Curso (Graduação em Desenvolvimento de Sistemas). Universidade Tecnológica Federal do Paraná, Medianeira, 2011.

This study aimed to goals a bibliographic study on techniques and technologies that enable implementation of the Reverse Ajax for bidirectional communication between client and server in *WEB* applications. The development of *WEB* technology is related, among other factors, the need to simplify the upgrade and maintenance of the application, however, some applications require constant updating of information as an Electronic Auction, *chat* , and this constant update, without requiring explicit user can be provided using a feature called reverse Ajax, also known as "Push Technology" Along this work be presented the technologies and tools used, as well as the explanation for each of the three techniques of reverse Ajax. We studied two possible technologies for implementing one of the techniques in one example, the new version of the Java platform *WEB* - JEE6 (*Java Enterprise Edition* 6) and the Ajax framework DWR (Direct *WEB* Remoting). Was chosen as an example *chat* application, where, when a user sends a message, all other connected users should receive it.

Keywords: *Ajax*, *comet, reverse*

ÍNDICE DE FIGURAS

ÍNDICE DE QUADROS

ÍNDICE DE TABELAS

LISTA DE SIGLAS

AJAX	*Asynchronous Javascript and XML*
API	*Application Programming Interface*
CSRF	*Cross-Site Request Forgery*
DHTML	*Dynamic Hyper Text Markup Language*
DOM	*Document Object Model*
DWR	*Dynamic WEB Routing*
EL	*Expression Language*
EJB	*Enterprise JavaBeans*
IBM	*Internacional Business Machines*
HTML	*Hyper Text Markup Language*
HTTP	*Hyper Text Transfer Protocol*
IDE	*Integrated Development Environment*
JEE	*Java Enterprise Edition*
JDBC	*Java Database Connectivity*
JME	*Java Micro Edition*
JNDI	*Java Naming and Directory Interface*
JSE	*Java Standart Edition*
JSF	*Java Server Faces*
JSP	*Java Server Pages*
JSR	*Java Specification Request*
JVM	*Java Virtual Machine*
PHP	*Hypertext Preprocessor*
TCP	*Transmission Control Protocol*
UTFPR	Universidade Tecnológica Federal do Paraná
XML	*eXtensible Markup Language*
XSLT	*eXtensible Stylesheet Language for Transformation*

SUMÁRIO

1 INTRODUÇÃO

Segundo Soares (2006), a Internet é atualmente, senão o mais importante, um dos principais meios de comunicação em todo o mundo. Desenvolver aplicações *WEB* exige a integração de linguagens e ferramentas visando desempenho, segurança e qualidade e cada vez mais escalabilidade.

Atualmente “Aplicação *WEB”* é o termo utilizado para designar, de forma geral, sistemas projetados para utilização através de um navegador, na Internet ou em redes privadas. Designa-se como um conjunto de programas que é executado em um servidor. O desenvolvimento da tecnologia *WEB* está relacionado, entre outros fatores, a necessidade de simplificar a atualização e manutenção da aplicação.

A arquitetura de sistemas distribuídos baseados na *WEB* não apresenta diferenças fundamentais em relação à de outros sistemas distribuídos (TANEMBAUM, 2007). A arquitetura para comunicação *WEB* consiste, tradicionalmente, em um navegador (cliente) realizando requisições de dados à um servidor *WEB*. As requisições em sua maioria são realizadas através do protocolo HTTP (*Hyper Text Transfer Protocol*), que permite a transferência de dados através da abordagem *Request/Response*. Nesta abordagem o cliente abre uma conexão HTTP com o servidor requisitando dados, após receber a requisição o servidor apenas fornece uma resposta e fecha a conexão.

Encontram-se aplicações *WEB* em todas as empresas, em todos os ramos de negócios, do mais simples ao mais complexo, de mais alto risco ou de maior desempenho e a forma mais comum encontrada é a conhecida como *Request/Response*.

1.1 CONTEXTUALIZAÇÃO

Existem tecnologias e técnicas que possibilitam uma abordagem diferente da tradicional, uma abordagem mais dinâmica e maleável. Uma comunicação reversa onde o servidor pode enviar dados aos clientes a ele conectados. Este recurso, que

descreve um estilo de Internet baseado na comunicação servidor cliente, onde a chamada de uma determinada operação é iniciada pelo servidor, é conhecido como "Tecnologia *Push*"[1].

Neste trabalho foram estudadas algumas técnicas e tecnologias que tornam possível essa comunicação, consolidando este estudo aplicou-se os conhecimentos desenvolvendo a implementação do recurso em uma situação real, baseado em um exemplo tipo *chat*, facilitando assim o entendimento da aplicação e a dinâmica envolvida.

1.2 OBJETIVO GERAL

Demonstrar o funcionamento da tecnologia Ajax Reverso através de um estudo experimental.

1.3 OBJETIVOS ESPECÍFICOS

Neste trabalho os objetivos específicos foram:

- Desenvolver um estudo sobre as existentes técnicas de programação que implementem a "Tecnologia *Push*";
- Analisar, projetar e desenvolver uma aplicação exemplo baseada em uma das técnicas estudadas demonstrando a usabilidade e importância deste recurso.
- Descrever o funcionamento da tecnologia implementada para o desenvolvimento da aplicação exemplo;
- Testar o funcionamento da aplicação exemplo.

[1] Denominação dada ao recurso de envio de dados pelo servidor

1.4 JUSTIFICATIVA

Para algumas aplicações, onde é necessária atualização constante de informações como um *Chat,* Pregão Eletrônico, jogos *Online* ou um editor colaborativo como o *Google Docs*, é interessante prover a comunicação do servidor com os clientes a ele conectados.

Esta implementação de conexões *WEB* permite atualizar os valores no cliente em um menor tempo possível, assim que estes sejam modificados, sem a necessidade de que o usuário da aplicação tenha que requisitar esta ação.

1.5 DIVISÃO DO TRABALHO

Este trabalho de conclusão de curso esta distribuído em quatro capítulos distintos: Introdução, Revisão Bibliográfica, Desenvolvimento da aplicação e Conclusão.

Na introdução o leitor é contextualizado sobre o problema a ser resolvido e a solução proposta, além de conhecer os objetivos e a justificativa do trabalho. Esta breve sessão descreve rapidamente o propósito e funcionamento de aplicações *WEB,* e um novo conceito para comunicação das informações, o Ajax Reverso. Em seguida a Revisão Bibliográfica trata de apresentar, ou, reapresentar técnicas ferramentas e tecnologias que são necessárias para o entendimento e desenvolvimento do trabalho. O capítulo 3, Desenvolvimento da aplicação, traz todo o conteúdo principal do problema, é neste capítulo que o problema da comunicação entre o servidor e cliente *WEB* é trabalhado no intuito de desenvolver e apresentar uma solução. Ainda no capitulo 3, é demonstrado o caminho percorrido para a resolução e apresentados os resultados finais. Ao final tem-se a conclusão, que traz o aprendizado ganho com o desenvolvimento e término do trabalho e sugestões para trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Para o desenvolvimento da aplicação servidora, responsável por receber as requisições *WEB* e enviar as atualizações necessárias aos clientes conectados, foi escolhida a Plataforma JEE. Esta edição da plataforma Java difere um pouco das outras por adicionar bibliotecas que permitam a implementação de aplicações distribuídas executando em um servidor de aplicações. Esta plataforma é considerada um padrão de desenvolvimento já contém uma série de especificações e containers necessários para que a aplicação desenvolvida seja considerada compatível.

A plataforma Java, juntamente com a linguagem Java, compilada nativamente formão um conjunto ideal para o desenvolvimento de uma aplicação servidora com arquitetura robusta.

A IDE (*Integrated Development Environment*) escolhida como base do desenvolvimento foi o Eclipse IDE em sua versão *Helios* (3.6). Como container da aplicação no servidor será utlizado o *Apache Tomcat* em sua versão 7.0.14, que já implementa os novos requisitos definidos pela JEE 6.

Para trabalhar as requisições Ajax foi escolhido o *DWR*, um *framework* que se propõe a facilitar o uso do Ajax na plataforma Java. Este que é considerado pela comunidade um dos mais robustos *frameworks javascript* do mercado, usa de mecanismos que permitem que scripts client/side invoquem métodos em objetos Java residentes no servidor. Em sua versão atual, 3.0, o *DWR* apresenta recursos que possibilitam a implementação do “Ajax Reverso”.

2.1 PLATAFORMA JAVA

Segundo FLANAGAN (2000), desde 1995 quando lançada pela Sun Microsystems a linguagem e plataforma Java tomaram de assalto o mundo da programação. Em sua definição David Flanagan descreve a importância de se fazer a distinção entre a linguagem de programação Java, a Java Virtual Machine(JVM) e a plataforma Java.

A linguagem de programação Java é a linguagem na qual os aplicativos Java (incluindo os componentes applets, servlets e JavaBeans) são escritos. É uma linguagem orientada a objetos e foi desenvolvida paralelamente a plataforma com uma equipe chefiada por James Gosling. É uma linguagem de grande portabilidade já que sua compilação é feita usando o *bytecode*, tem sintaxe fácil, semelhante a de C/C++ e principalmente C#. Suporta processamento distribuído, *multi-threading*, tratamento claro a exceções e tem como um de seus pontos fortes o alto poder de reuso de código.

A plataforma Java é distinta tanto da linguagem quanto da JVM, é o conjunto predefinido de classes Java que existe em cada instalação Java; estas classes estão disponíveis para uso por todos os programas Java.

Pode-se observar o funcionamento da plataforma com base na Figura 01.

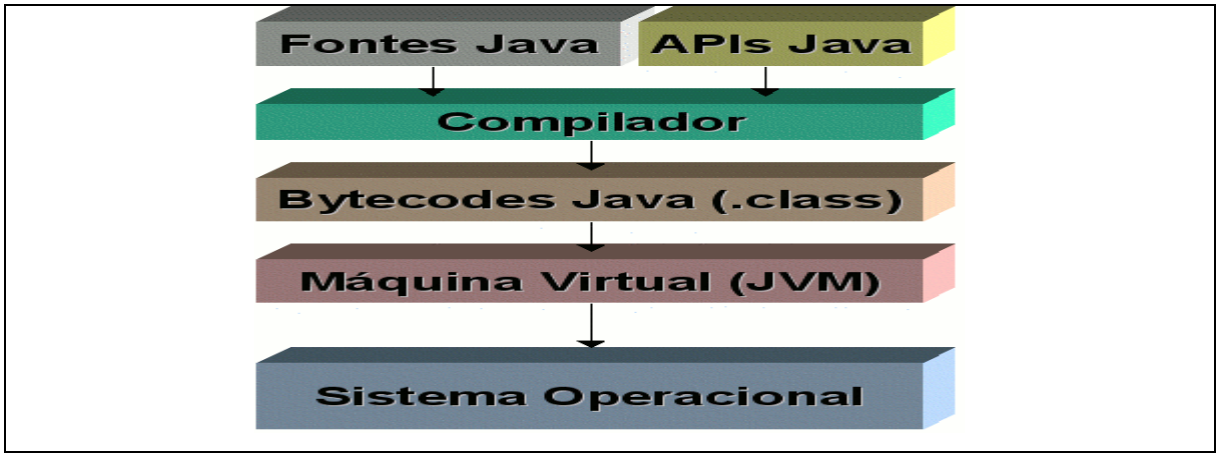

Figura 01 Ilustração do funcionamento da plataforma Java.

Fonte: MICROSYSTEMS, SUN (2009)

Segundo a ORACLE (2011), atual empresa proprietária e mantenedora da tecnologia Java, a plataforma Java subdivide-se em quatro diferentes plataformas, que basicamente, consistem na Java Virtual Machine e um conjunto de APIs (Application Programming Interface). São elas:

- Java Platform, Standard Edition (JSE)
- Java Platform, Enterprise Edition (JEE)
- Java Platform, Micro Edition (JME)
- Java FX

2.2 JAVA EE

Segundo a ORACLE (2011), a plataforma *Java Enterprise Edition* (JEE) é construída com base na plataforma *Java Standart Edition* (JSE), porém a JEE fornece uma API e um ambiente que propicia o desenvolvimento e execução em larga escala, utilizando várias camadas, com escalabilidade e segurança para aplicações de rede.

A plataforma Java EE foi projetada para ajudar os desenvolvedores a criar em larga escala, multi-camada e escalável, aplicações de rede confiáveis e seguras. A abreviação dada pela ORACLE (2011) para tais aplicações é "aplicações corporativas", assim chamadas porque são projetadas para resolver os problemas enfrentados por grandes empresas. Os benefícios de um aplicativo corporativo são úteis, até mesmo essenciais, para os desenvolvedores individuais e pequenas empresas em um mundo cada vez mais em rede.

Tipicamente, aplicações multi-camadas tem uma camada cliente, uma camada intermediária e a camada de dados. A camada do cliente consiste em um programa cliente que faz pedidos para a camada intermediária. A função da camada intermediária de negócios é de lidar com as solicitações do cliente e dados de aplicativos do processo, armazenando-o em um repositório de dados permanente na camada de dados. O desenvolvimento de aplicações Java EE concentra-se na camada intermediária para fazer o gerenciamento de aplicativos empresariais mais fáceis, mais robustos e seguros.

Os aplicativos clientes acessam um servidor Java EE fazem solicitações ao servidor, que por sua vez, processa os pedidos e retorna uma resposta de volta para o cliente. Muitos tipos diferentes de aplicações podem ser clientes Java EE, e não necessariamente estas aplicações são aplicações Java. O cliente pode ser um navegador web, um aplicativo independente, ou outros servidores, e executado em uma máquina diferente do servidor

A camada web consiste de componentes que lidam com a interação entre clientes e a camada de negócios. Suas principais tarefas são as seguintes:

- Gerar dinamicamente conteúdo em vários formatos para o cliente.
- Coletar a entrada dos usuários e retornar os resultados apropriados a partir dos componentes na camada de negócios.

- Controlar o fluxo de telas ou páginas no cliente.
- Manter o estado de dados para uma sessão do usuário.
- Executar alguma lógica básica e manter dados temporariamente em componentes *JavaBeans*.

As seguintes tecnologias Java EE são usadas na camada web em aplicações Java EE.

Tecnologia	**Propósito**
Servlets	Classes com a linguagem Java de programação que processam dinamicamente solicitações e geram respostas, geralmente para páginas HTML
Java Server Faces	Estrutura de componentes de interface com usuário para aplicações *WEB* que permite incluir componentes (como campos e botões) em uma página, converter e validar dados na interface, salvar dados para as aplicações do lado do servidor, e manter o estado dos componentes.
Facelets	Aplicações *Facelets* são um tipo de aplicativos JSF que usam páginas XHTML ao invés de páginas JSP.
Expression Language	Um conjunto de tags padrão utilizado em JSP e páginas *Facelets* para se referir aos componentes Java EE.
Java *Server Pages*	Baseado em texto de documentos são compiladas em *servlets* e definem como conteúdo dinâmico pode ser adicionado a páginas estáticas, como páginas HTML.
Standard Tag Library	A biblioteca de tag que encapsula um núcleo de funcionalidades comuns para páginas JSP
JavaBeans	Objetos que funcionam como armazenamento de dados temporários para as páginas de uma aplicação

Tabela 01 Tecnologias da utilizadas pela camada *WEB* da plataforma JEE

Fonte: Adaptado de ORACLE (2011)

A camada de negócios é composta de componentes que fornecem a lógica de negócios para uma aplicação. Lógica de negócios é o código que fornece funcionalidade para um domínio particular, como o setor financeiro, ou um site de

comércio eletrônico. Conforme descrito pela ORACLE (2011), em um aplicativo corporativo adequadamente projetado, a funcionalidade do núcleo existe nos componentes da camada de negócios.

2.2.1 Servidores Java EE

Um servidor Java EE é uma aplicação de servidor que implementa a plataforma e estabelece o padrão Java EE serviços. Estes servidores são às vezes chamados de servidores de aplicação, porque permitem prover dados de aplicativos para clientes, bem como servidores web prover páginas web para os navegadores. Os chamados servidores de aplicação realizam a integração de vários tipos de componentes de aplicativos que correspondem às camadas em uma aplicação multi-camadas. O servidor também fornece serviços para estes componentes na forma de um recipiente.

Containers são a interface entre o componente e a funcionalidade de baixo nível fornecidos pela plataforma para apoiar esse componente. A funcionalidade do recipiente é definida pela plataforma, e é diferente para cada tipo de componente. No entanto, o servidor permite que os diferentes tipos de componentes trabalhem em conjunto para fornecer funcionalidade de um aplicativo corporativo.

Um componente *WEB* pode ser um *Servlet*, um *Facelet* ou uma página JSP. O container *WEB* é a interface entre os componentes *WEB* e o servidor. Este container gerencia o ciclo de vida dos componentes, despacha pedidos de componentes da aplicação, e fornece interfaces de dados de contexto, tais como informações sobre a solicitação atual.

O recipiente de cliente do aplicativo é a interface entre os clientes de aplicativos JEE, que são aplicações JSE utilizando componentes do servidor JEE, e o servidor. O aplicativo cliente é executado na máquina do cliente, e é o *gateway* entre o aplicativo cliente e os componentes do Java EE servidor que o cliente utiliza.

O container EJB é a interface entre beans corporativos, que fornecem a lógica de negócios, e o servidor. Este container é executado no servidor e gerencia a execução dos *Beans* de um aplicativo corporativo.

2.2.2 Java EE 6 e Servlets 3.0

Servlet é um componente da *API Java Servlet*, presente na plataforma JEE, que tem basicamente a finalidade de processar dinamicamente requisições e respostas, proporcionando dessa maneira novos recursos aos servidores.

Um *Servlet* pode ser considerado um módulo que estende a funcionalidade de um servidor *WEB*, através de módulos de aplicação implementados em Java. Essa API disponibiliza ao programador da linguagem Java uma interface para o servidor *WEB*. Os *Servlets* normalmente utilizam o protocolo HTTP, apesar de não serem restritos a ele.

Em novembro de 2009 a Sun, antiga proprietária e mantenedora da tecnologia, lançou a versão 6 da JEE. Esta versão trouxe um grande número de mudanças significativas nas especificações que definem a plataforma.

A especificação JSR (Java Specification Request) 315, que define a versão 3.0 dos *Servlets*, é uma das mais significativas mudanças da versão da JEE6 segundo a opinião do Dr Xinyu Liu (LIU, 2009), arquiteto Java certificado pela SUN que até o ano da publicação do artigo trabalhava na *HealtCare Corporation* e na *IT Consulting*. No artigo Liu explica em detalhes o porquê considera que o processamento assíncrono é fundamental em aplicações colaborativas, multi-usuário que definem a *WEB* 2.0.

Uma grande melhoria no padrão HTTP 1.1, também implementada pela JSR 315 são as conexões persistentes. Na antiga HTTP 1.0, a conexão entre um cliente *WEB* e o servidor era fechada após um ciclo de requisição e resposta. No HTTP 1.1, uma conexão é mantida viva e reutilizada para múltiplas solicitações. Conexões persistentes reduzem perceptivelmente a demora de resposta, isso porque o cliente não precisa abrir uma nova conexão TCP (*Transmission Control Protocol)* a cada solicitação.

A JSR 315 também traz melhorias como maior facilidade de configuração, facilidade de integração com outras bibliotecas e *frameworks*, e melhorias internas na API já existente.

2.3 ECLIPSE

O Eclipse IDE foi inicialmente desenvolvido pela empresa IBM (*Internacional Business Machines*), que logo após sua primeira versão doou-o como software livre para a comunidade Java. O projeto teve um gasto inicial de 40 milhões de dólares e hoje é o IDE Java mais utilizado no mundo. Desde 2004 o Eclipse é mantido por uma organização sem fins lucrativos a Eclipse Foundation, criada especificamente para isto. De acordo com a Eclipse Foundation (2009), atualmente a fundação Eclipse oferece 4 serviços à comunidade:

- Infra-estrutura de TI (Tecnologia da Informação);
- Gestão de Propriedade Intelectual;
- Processo de Desenvolvimento;
- Desenvolvimento do Ecossistema.

Seu núcleo está arquitetado para a utilização de *plug-ins*, adicionando novas funcionalidades ao ambiente de desenvolvimento. Atualmente, a comunidade conta com inúmeros *plug-ins* para muitas funcionalidades que não são nativas da IDE, como por exemplo *plug-ins* para desenvolvimento gráfico tanto *desktop*, quanto *web*.

O conjunto de bibliotecas nativas, somado aos *plug-ins* desenvolvidos pela comunidade tornam o Eclipse um bom ambiente de desenvolvimento Java atualmente. A versatilidade e facilidade para desenvolver aplicações, módulos e até novos *plug-ins* é um grande destaque. A Figura 02 ilustra o ambiente de desenvolvimento da IDE Eclipse.

O projeto eclipse também conta com uma plataforma de desenvolvimento a partir da qual você pode criar a sua aplicação Swing. Muitas das funcionalidades de manipulação de janelas e ações já estão implementadas e muitos módulos básicos de aplicações já estão desenvolvidos. A própria IDE é desenvolvida a partir desta plataforma, a RCP (*Rich Client Plataform*).

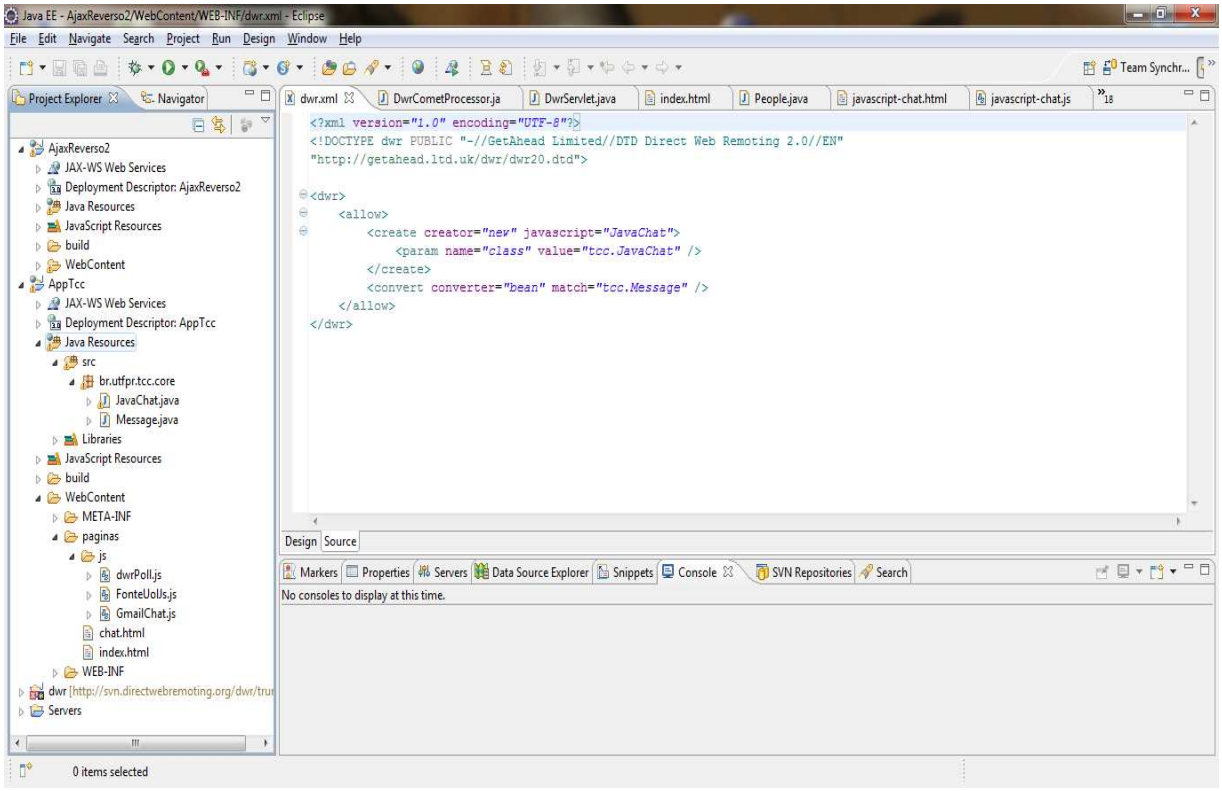

Figura 02 Visualização do ambiente de produção com a IDE Eclipse.

2.4 SERVIDORES DE APLICAÇÃO JEE6

Um servidor Java EE é uma aplicação de servidor que implementa toda ou parte da plataforma estabelecendo o padrão Java EE de serviços. Serão listados neste trabalho os três servidores de aplicação *WEB* que foram estudados para realizar a implementação do objeto de exemplo. São eles: Jetty (por ter sido um dos pioneiros na comunicação assíncrona), GlassFish (por ter sido desenvolvido pela própria Sun) e Tomcat (por ser popular e ter fácil integração com a IDE Eclipse).

2.4.1 *Jetty*

É um servidor *HTTP* e *Servlet* Container totalmente escrito em Java. É o grande concorrente do Tomcat que ficou famoso por ter sido utilizado como o *Servlet*

container do JBoss antigamente. Segundo JETTY (2011), a grande vantagem com relação ao Tomcat é a sua fácil configuração. Ele também foi o pioneiro a usar I/O assíncrono para aguentar uma carga maior de usuários simultâneos sem depender da estratégia *thread-per-connection*.

O servidor *WEB Jetty* fornece um servidor HTTP e *Servlet* container capaz de servir conteúdo estático e dinâmico a partir de um instanciações autônomo ou incorporado. O servidor e seus componentes são hospedados atualmente pela fundação eclipse. O projeto *Jetty* inclui:

- Conexão HTTP assincrona
- *Container* de *Servlets*
- Servidor *Web Sockets*
- Cliente HTTP assíncrono
- OSGi, JNDI, JMX, JASPI, suporte AJP

Jetty sempre foi, dentre os principais concorrentes, o servidor de aplicações WEB mais avançado na questão de comunicação assíncrona. Podemos observar este avanço analisando a Tabela 02 que traz o histórico de versões do servidor e as versões do Java, HTTP, *Servlet* e JSP utilizadas, e onde podemos visualizar a utilização do protocolo HTTP 1.1 em sua especificação atual, desde a versão 4.x.

Jetty	**Java**	**HTTP**	**Servlet**	**JSP**
8.x	1.6	HTTP/1.1 RFC2616	3.0	2.1
7.x	1.5	HTTP/1.1 RFC2616	2.5	2.1
6.x	1.4-1.5	HTTP/1.1 RFC2616	2.5	2.0
5.x	1.2-1.5	HTTP/1.1 RFC2616	2.4	2.0
4.x	1.2	HTTP/1.1 RFC2616	2.3	1.2
3.x	1.2	HTTP/1.1 RFC2068	2.2	1.1
2.x	1.1	HTTP/1.0 RFC1945	2.1	1.0
1.x	1.0	HTTP/1.0 RFC1945		

Tabela 02 Histórico de versões do servidor *Jetty*

Fonte: Adaptado de JETTY (2011)

2.4.2 *Glassfish* 3.1

GlassFish Server 3.1 é o sucessor do versões anteriores 3.0.x com um tempo de execução modular, flexível e baseado no padrão OSGi. Neste versão o container implementa todas as especificações da plataforma JEE6 alem de contar com recursos de *cluster* com a administração centralizada de vários clusters e alta disponibilidade de componentes *stateful*.

Segundo GLASSFISH (2011), a nova versão é o mais rápido servidor de aplicativos *open source* que oferece recursos avançados como controle de versão de aplicativos, recursos de escopo do aplicativo, e integração com as ferramentas de desenvolvimento NetBeans 7.0, Eclipse e outros.

O projeto do *Glassfish* foi lançado pela Sun em junho de 2005 e atualmente se encontra na terceira versão. Pode rodar aplicação não somente em Java, mas também em outras linguagens *WEB* como .NET e PHP. Por ser um produto criado pela e atualmente mantido pela Oracle, tem 100% de compatibilidade com a tecnologia Java. Atualmente, os seus defensores afirmam que o *Glassfish* é mais produtivo, pois consegue ter as funcionalidades do Jboss e um tempo de inicialização comparável a um *Tomcat*, além de uma excelente integração com *IDE's* como o NetBeans, que tem uma parte de seu ambiente de integração no modo visual.

2.4.3 *Apache Tomcat* 7

O *Apache Tomcat* é um servidor feito puramente em Java para aplicações *WEB* que implementa parte das especificações da plataforma *Java Enterprise Edition*. Ele também pode comportar-se como um servidor *WEB* (HTTP) ou funcionar integrado a um servidor *WEB* dedicado (como o Apache ou o IIS).

Nascido no Projeto *Apache* Jakarta da *Apache Software Foundation*, o servidor foi oficialmente autorizado pela Sun como a implementação de referência para as tecnologias Java *Servlet* e *JavaServer Pages* (JSP). Ele cobre parte da especificação J2EE com tecnologias como *servlet* e JSP e tecnologias de apoio

relacionadas JNDI (*Java Naming and Directory Interface) Resources* e JDBC (*Java Database Connectivity) DataSources*, contudo, ele não implementa pacotes EJB (Enterprise JavaBeans). O *Tomcat* não deve ser confundido com o servidor *WEB* Apache, que é um servidor HTTP totalmente implementado em C.

Três anos após o lançamento do *Tomcat* 6 os desenvolvedores da *Apache* entregaram a versão 7 que implementa as novas especificações:

- Java Servlet 3.0
- Java Server Pages 2.2
- Expression Language (EL) 2.2.

Outra modificação vem das versões anteriores que exigiam que todas as alterações de configuração *WEB* das aplicações fossem realizadas no arquivo `web.xml`. A nova versão quebra as instruções de configuração em vários arquivos separando as configurações de segurança, novas conexões, parâmetros de inicialização e integração.

2.5 AJAX

Ajax é um acrônimo para definir *Asynchronous Javascript and* XML (*eXtensible Markup Language*) (Javascript e XML assíncrono). Segundo SOARES (2006), *Ajax* é muito mais que a junção de *JavaScript* e XML, é todo um conceito de navegação e atualização de páginas *WEB*.

Apesar de revolucionário Ajax não é algo novo, os navegadores implementam essa tecnologia desde o ano 2000. Soares também afirma que algumas partes descritas na definição de *Ajax* já foram denominadas de DHTML (*Dinamyc Hyper Text Markup Language)* e Script Remoto.

Desenvolvida inicialmente por Jesse James Garret, pesquisador e desenvolvedor na área de tecnologias *WEB*, co-fundador da *Adaptive Path* e também do *Information Architecture Institute,* a definição e utilização *Ajax* tem se popularizado e estimulado a construção de aplicações *WEB* mais dinâmicas e criativas.

O *Ajax* incorpora em sua especificação:

- Exposição e interação dinâmica usando o DOM (*Document Object Model*);
- Intercâmbio e manipulação de dados usando XML e XSLT (*eXtensible Stylesheet Language for Transformation*);
- Recuperação assíncrona de dados usando o objeto *XMLHttpRequest* e *XMLHttpResponse*;
- *JavaScript* fazendo a junção entre os elementos.

Com a popularização de aplicações que funcionam inteiramente na *WEB*, bem como com o aumento da velocidade das conexões banda larga, o problema da espera pelo envio e retorno da página inteira se tornou muito mais evidente para o usuário.

Se a Internet fosse re-projetada a partir do zero para aplicações, nas escalas e moldes atuais, não fariam que os usuários esperassem pelo carregamento completo das páginas. Uma vez que a interface está carregada, a interação do usuário não deve parar a cada requisição realizada ao servidor.

Numa aplicação *WEB* clássica baseada em páginas, o navegador não sabe nada sobre o que o utilizador está realmente realizando em suas ações conseqüentes. Todas essas informações são retidas no servidor *WEB*, tipicamente na sessão do utilizador.

Já em uma aplicação *Ajax*, parte da lógica da aplicação é movida para o navegador. Neste novo cenário um documento mais complexo é entregue ao navegador, uma grande proporção do qual é código *JavaScript*. O tráfego tem seu maior pico no início, com uma grande e complexa quantidade de dados sendo entregue de uma única vez, quando o usuário entra. Para uma aplicação simples, o tráfego cumulativo pode ser menor em uma aplicação sem Ajax, mas conforme o tamanho e a complexidade aumentam, as vantagens do uso do Ajax também aumentam.

Antigamente, para uma aplicação *WEB* clássica se comunicar com o servidor, era necessário clicar em um *hyperlink* ou submeter um formulário, e então aguardar, o que interrompia a interação do usuário. Atualmente o uso de Ajax possibilita a comunicação com o servidor em resposta a um movimento ou arraste

do mouse, ou até quando digita-se, habilitando o servidor a trabalhar juntamente com o usuário.

Uma aplicação *Ajax* é uma porção de código complexo que pode se comunicar dinâmica e eficientemente com o servidor enquanto o usuário continua com seu trabalho. Ela é claramente uma evolução das antigas aplicações baseadas apenas em páginas.

2.6 AJAX REVERSO

Para algumas aplicações, onde é necessário a atualização constante de informações como um *chat* ou Pregão Eletrônico, a clássica implementação *Request/Response* (onde o cliente faz a requisição e o servidor apenas fornece uma resposta) da *WEB* seria melhor prover a comunicação do servidor com os clientes a ele conectados.

Este recurso, que descreve um estilo de Internet baseado na comunicação bidirecional entre cliente e servidor, onde, após o contato inicial do cliente, o servidor pode realizar a chamada de uma determinada operação no cliente, é conhecido como “Tecnologia *Push*”. Com os benefícios trazidos pela tecnologia Ajax, foi possível desenvolver técnicas *WEB* que possibilitam emular essa comunicação reversa.

O “Ajax Reverso” é a técnica que permite juntar tecnologias tanto no lado cliente quanto no servidor para permitir que o servidor envie dados ao cliente, sem haver explicitamente uma requisição do mesmo. A implementação de Ajax reverso dá a impressão de que o servidor possa realmente invocar ações no cliente, o que não ocorre de fato, devido às restrições do protocolo HTTP.

De acordo com CRANE E MCCARTHY (2008), para entender o *Ajax* Reverso, primeiro é necessário entender o protocolo HTTP. Este protocolo de comunicação foi projetado para recuperação de documentos em servidores, conforme ilustrado na Figura 03, e como tal, ele tem duas importantes características:

- A comunicação entre o cliente e o servidor é sempre iniciado pelo cliente e nunca pelo servidor.

- As conexões entre o cliente e o servidor são transitórios, e o servidor não mantém qualquer informação de estado de longo prazo sobre o cliente. Pelo menos, esta foi a situação com a versão 1.0 da especificação.
-

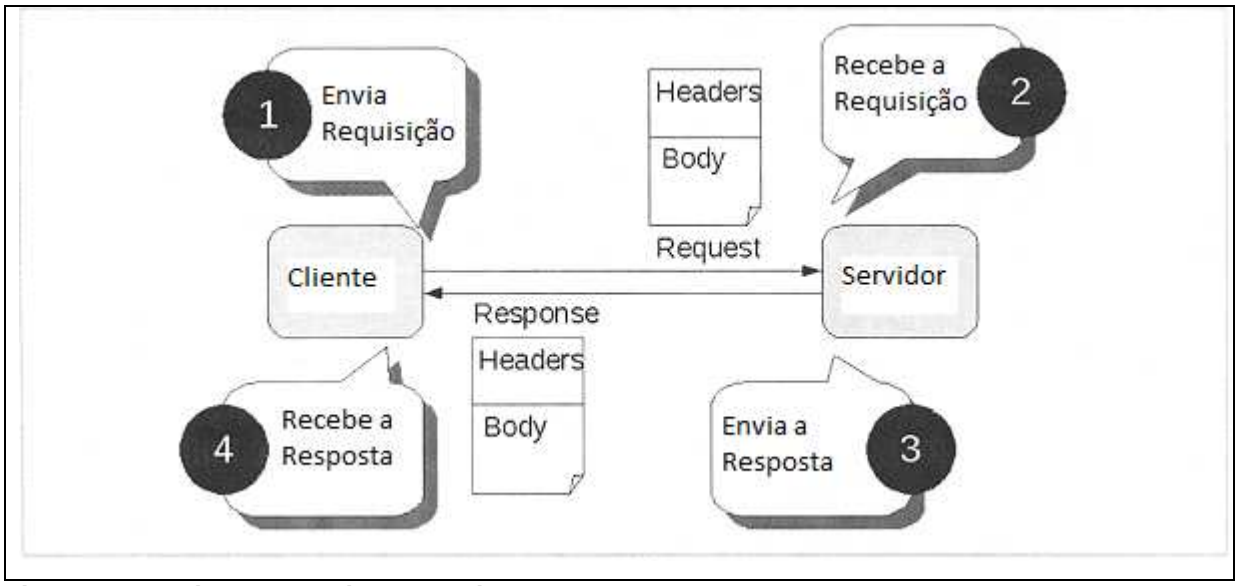

Figura 03 Funcionamento do protocolo *HTTP*

Fonte: Adaptado de CRANE E MCCARTHY (2008)

Existem três formas de se implementar o Ajax Reverso para emular essa funcionalidade de envio de dados pelo servidor: *Polling*; *Comet*; e *PiggyBack*.

2.6.1 Polling

É talvez a técnica de mais fácil implementação de Ajax reverso e também a que esteja menos perto da situação real de comunicação do servidor com o cliente. Consiste em realizar uma requisição do cliente para o servidor em um determinado intervalo de tempo. Assim que o servidor recebe a requisição de atualização, é realizado o envio das informações pendentes, desde o ultimo envio (AMADEI, 2007).

Com a realização do *polling* pode haver requisições desnecessárias, por não haver conteúdo a ser enviado ao servidor, e também, o cliente pode estar visualizando dados desatualizados, o que seria aceitável em muitas aplicações.

Em uma implementação mais simplista desta técnica é possível atualizar todos os dados da página, em cada requisição do *poll,* deixando o usuário descobrir o que foi atualizado. Nesta abordagem, a maioria dos dados seria redundante e acabaria por pesar desnecessariamente o trafego de dados. Para aprimorar esta técnica otimizando seu desempenho CRANE E MCCARTHY (2008) sugere duas abordagens:

- Atribuir um número de versão para cada entidade, e incrementá-lo quando atualizar. Se o cliente informa ao servidor a versão atual de cada elemento que ele conhece, o servidor pode comparar a versão e enviar apenas as entidades mais recentes.
- Atribuir um `timestamp` para cada entidade, e enviar o tempo da última atualização a cada pedido de atualizações. O servidor pode enviar os dados das entidades atualizadas desde a última chamada.

Pode-se observar o comportamento da técnica *polling* com base na Figura 04, onde o cliente realiza o *poll* em um intervalo de tempo, e o servidor envia a atualização, caso houver.

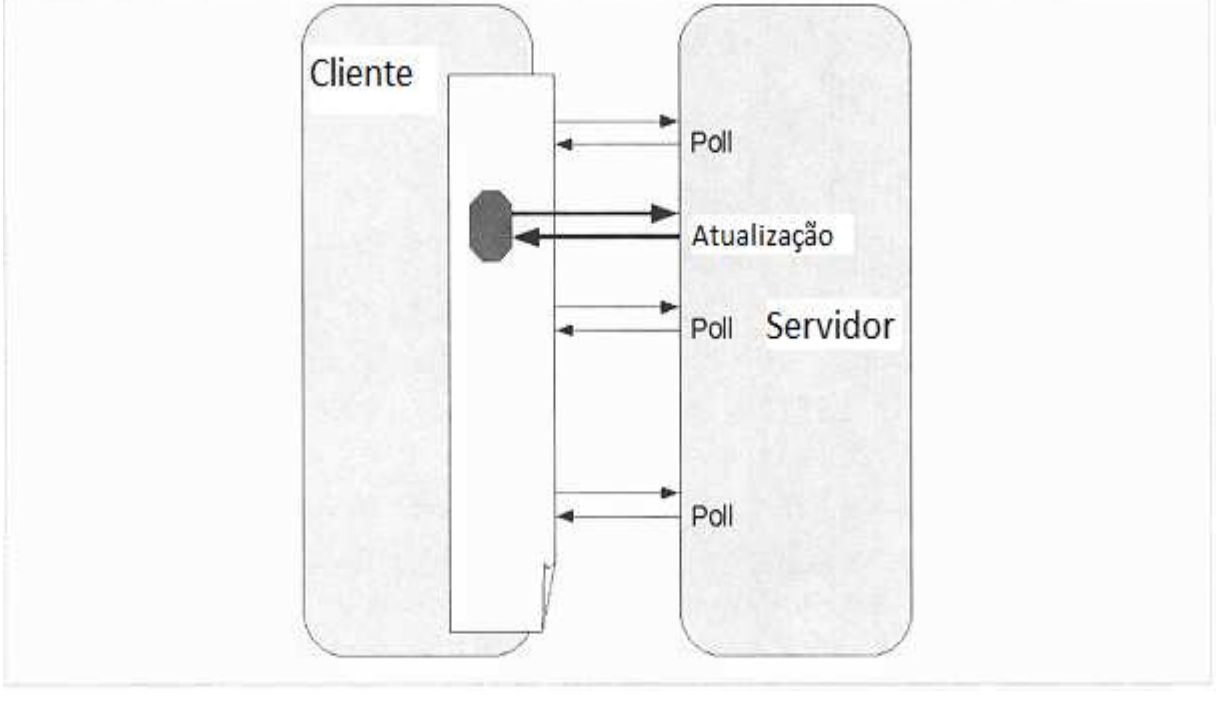

Figura 04 Representação da comunicação reversa utilizando a técnica *polling*

Fonte: Adaptado de CRANE E MCCARTHY (2008)

2.6.2 Comet

Comet é a técnica que mais se aproxima da real comunicação bi-direcional. Após a estação cliente realizar a primeira chamada à página da aplicação, o servidor realiza a resposta, porém, ao invés da implementação que seria normal ao HTTP, mantêm a conexão ativa, possibilitando futuramente que o servidor possa enviar atualizações sem que sejam explicitamente requisitadas. Este comportamento descrito pela técnica *Comet* pode ser representado através da

CRANE E MCCARTHY (2008) lembra que, em uma chamada HTTP normal para o servidor, a resposta é concluída logo após a chegada do pedido, e a chegada da resposta geralmente é tratada como um único evento no lado do cliente. Com o *Comet*, o fluxo de dados de resposta é mantido aberto por um tempo significativamente mais longo e, normalmente, são enviados vários pedaços de dados de volta na resposta, cada um destes sendo tratado como um evento distinto no cliente

Este termo teve seu cunho em meados de 2006, e foi dado pelo Engenheiro de software Alex Russell, co-fundador do framework *DWR*, ao postar em seu blog algo como: “O novo termo foi uma brincadeira com o Ajax (Ajax e *Comet* são dois produtos de limpeza de uso doméstico muito comuns nos EUA)” (RUSSELL, 2006).

Porém, cada conexão aberta consome uma *Thread Servlet*, o que acaba por exceder o limite de uso de memória do servidor rapidamente. Em poucas palavras, as *threads* ficam ociosas à espera de atualizações para serem enviadas e, ao menos antes da chegada da JSR 315 da JEE6, este estado de ócio x uso de memória acaba saindo caro de mais.

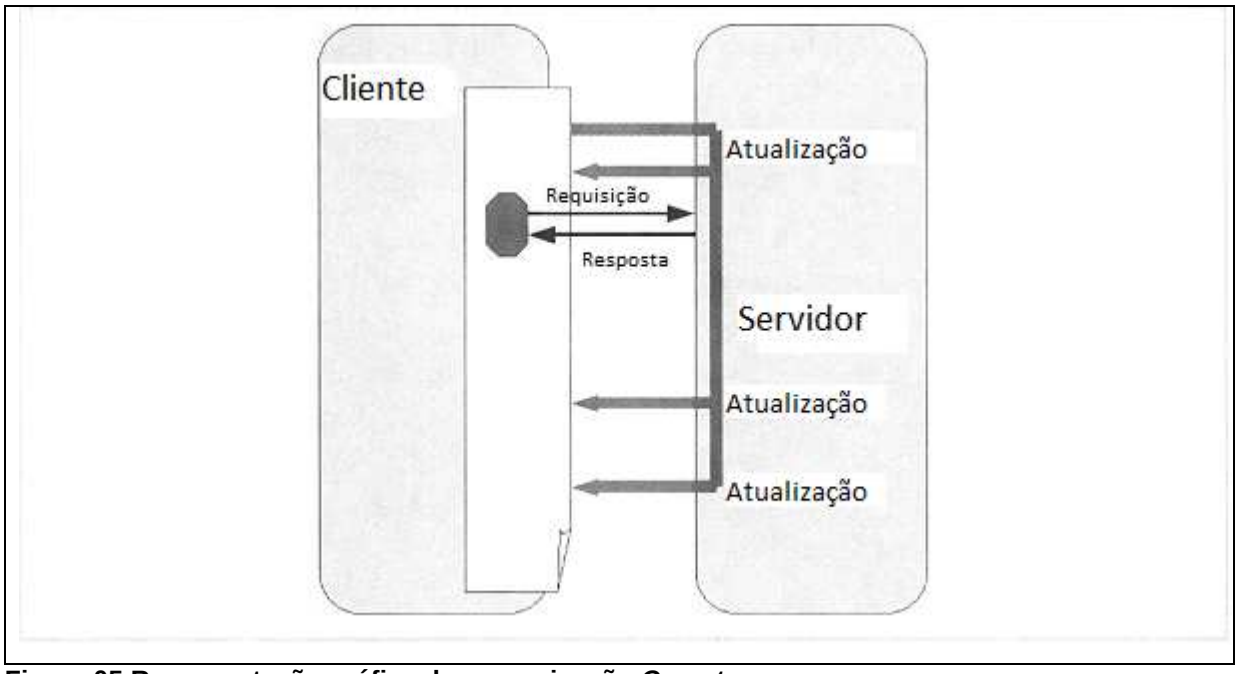

Figura 05 Representação gráfica da comunicação *Comet*
Fonte: Adaptado de CRANE E MCCARTHY (2008)

2.6.3 *PiggyBack*

PiggyBack significa algo como “pegar uma carona”. A técnica considerada oportunista aproveita uma requisição explicita realizada pelo cliente para enviar as atualizações pendentes, ou seja, além da informação requisitada o servidor realiza o envio de todas as informações que foram atualizadas desde a última requisição. (AMADEI, 2007).

Entre as três técnicas apresentadas, esta tende a ser a mais complexa e a que provê menor interatividade do servidor com o cliente. Mesmo sendo mais complexa e com menor resposta da parte servidora, “pegar uma carona” na requisição tem a vantagem de adicionar menos overhead no trafego da aplicação.

É possível utilizar uma combinação de *Polling* com *PiggyBack* para otimizar o trafego gerado pela aplicação e o tempo que os dados ficam desatualizados no cliente. Como podemos observar na Figura 06 o *poll* no lado cliente é inicializado, para a obtenção das atualizações, e no intervalo entre uma e outra requisição deste *poll* o cliente pode realizar outra requisição qualquer. O servidor irá aproveitar esta requisição explícita do cliente para enviar a atualização dos dados, reiniciando a

temporalidade do *poll*, uma vez que a atualização gerada pelo mesmo acabou de acontecer. Sendo assim, o tempo de resposta ao cliente fica mais rápido e dinâmico.

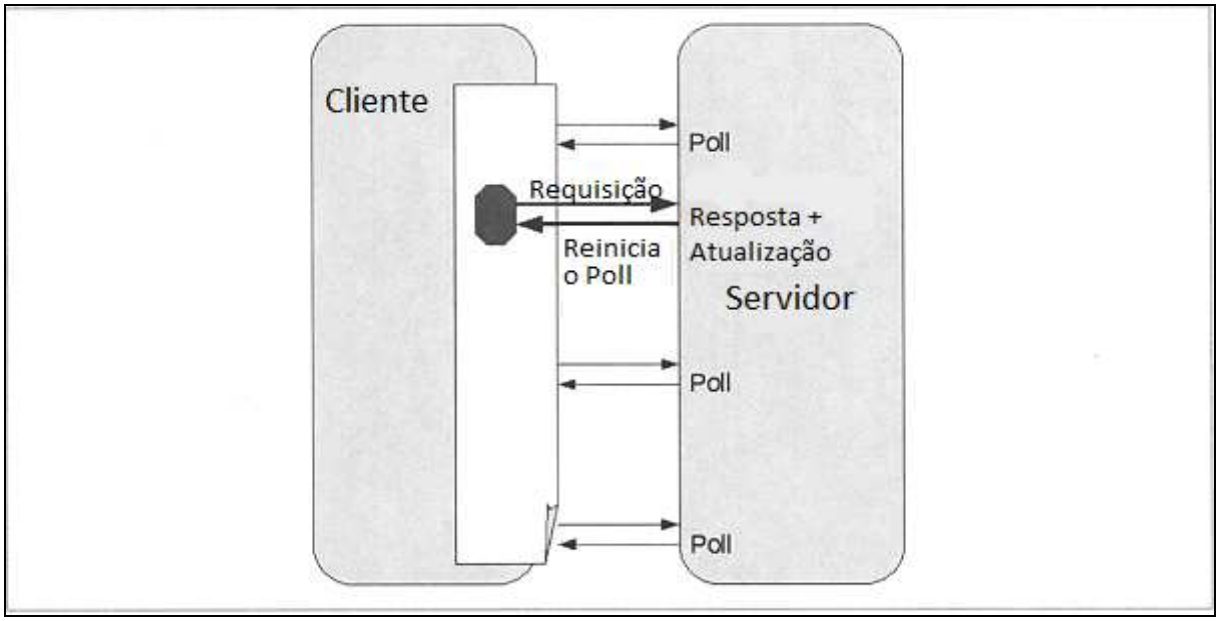

Figura 06 Utilização de duas técnicas em conjunto para otimizar implementação
Fonte: Adaptado de CRANE E MCCARTHY (2008).

2.7 DWR

O *framework DWR* tem como principal proposta facilitar a utilização do Ajax na plataforma Java. O *DWR* provê mecanismos que permitem que o código *JavaScript* cliente possa executar chamadas “diretas” as classes Java do servidor.

O *framework* possui duas partes:

- Java Servlet recebendo e respondendo as requisições Ajax
- *JavaScript* realizando as requisições Ajax e atualizando os dados no navegador.

O *Servlet* do *DWR* gera automaticamente código *JavaScript* baseado nas classes mapeadas no servidor. Através destes códigos é possível invocar os métodos dos objetos registrados.

O *DWR* fornece ainda um conjunto de utilitários que facilitam a realização de várias tarefas comuns, como por exemplo popular um combo de elementos ou preencher uma tabela.

A partir da versão 2.0 o *framework* incorporou funcionalidades que implementam as três principais técnicas do Ajax Reverso: *Polling*, *PiggyBack* e *Comet*. Ao implementar esta última, ele se utiliza basicamente de duas estratégias:

- *Streaming*: o servidor abre uma única conexão persistente e, ao enviar dados a mesma não é fechada.
- *Long polling*; o servidor abre uma conexão persistente que é mantida aberta até que o servidor tenha dados para enviar. Após o envio dos dados o servidor encerra a conexão e imediatamente o navegador abre uma nova.

O *DWR* pode ser facilmente integrado a outros frameworks *WEB* como JSF (*Java Server Faces*) e EJB e sua configuração é realizada no descritor `web.xml` da aplicação.

3 MATERIAS E MÉTODOS

As pesquisas podem ser classificadas por diferentes critérios, tais como: área do conhecimento, finalidade, objetivos gerais e métodos empregados. Esta pesquisa segundo a sua finalidade classifica-se em pesquisa aplicada, pois é voltada à aquisição de conhecimentos com vistas à aplicação numa situação específica. Segundo objetivo geral classifica-se como pesquisa explicativa, as pesquisas explicativas tem como finalidade explicar a razão, o porquê das coisas. E segundo métodos empregados trata-se de uma pesquisa bibliográfica e experimental (GIL, 2010).

Portanto, como metodologia para estudar o funcionamento do *Ajax* Reverso para à comunicação bidirecional entre servidor e cliente em uma aplicação Java *WEB*, foi utilizado neste trabalho o desenvolvimento de um estudo bibliográfico sobre técnicas e tecnologias que abordam o *Ajax* Reverso*,* bem como sua implementação utilizando como exemplo um *chat.* A partir das referências de bibliográficas estudadas, listou-se vários códigos necessários para o desenvolvimento da comunicação, descrevendo-a passo a passo.

Uma aplicação *WEB* para comunicação instantânea (seja um *chat* ou um mensageiro *WEB*) necessita dos dados no mesmo instante em que os mesmos são gerados. No caso do *chat* ainda é necessário que vários clientes conectados recebam o texto enviado por um destes. Este cenário é perfeito para aplicar a técnica de *Ajax* Reverso *Comet*, onde, a cada novo usuário conectado ou a cada mensagem nova enviada, todos os outros clientes são avisados do evento pelo servidor.

3.1 REFERENCIAIS DE APLICAÇÕES

Existem atualmente aplicações *WEB* conceituadas que utilizam-se de *Comet* para prover a comunicação *WEB* do servidor com seus clientes. Pode-se citar como exemplo o *chat* UOL (Quadro 01), Gmail e Hotmail e GoogleDocs, sendo que estes

três últimos não há provas do uso de *Comet* (códigos criptografados, apenas supõe-se pelo comportamento da aplicação.

```
parent.getCometURL = function() {
    return "http://" + DOM_INITIAL_LETTER + Math.floor(Math.random() *
MAX_DOM) + ".bpm.uol.com.br:8080/comet-chat/action";
}
```

Quadro 01 Código JavaScript incluido no site de *chat* do Uol (UOL, 2011)

3.2 ANÁLISE

Depois de definido o tema e o escopo da aplicação, foi iniciada a análise modelagem da mesma. O primeiro passo da modelagem foi a descrição dos requisitos mínimos a serem desenvolvidos, listados na Tabela 03.

ID	DESCRIÇÃO
01	O usuário deverá se conectar à uma página de chat
02	A página de chat deve permitir que o usuário envie mensagens através de um campo de texto.
03	O sistema deve armazenar as mensagens enviadas pelos usuários.
04	O sistema deve atualizar as mensagens enviadas à todos os usuários conectados.

Tabela 03 Lista de requisitos

Em seguida foram elaborados o diagrama com a visão geral da solução, representado pela Figura 07 e o diagrama de sequência que pode ser observado na Figura 08.

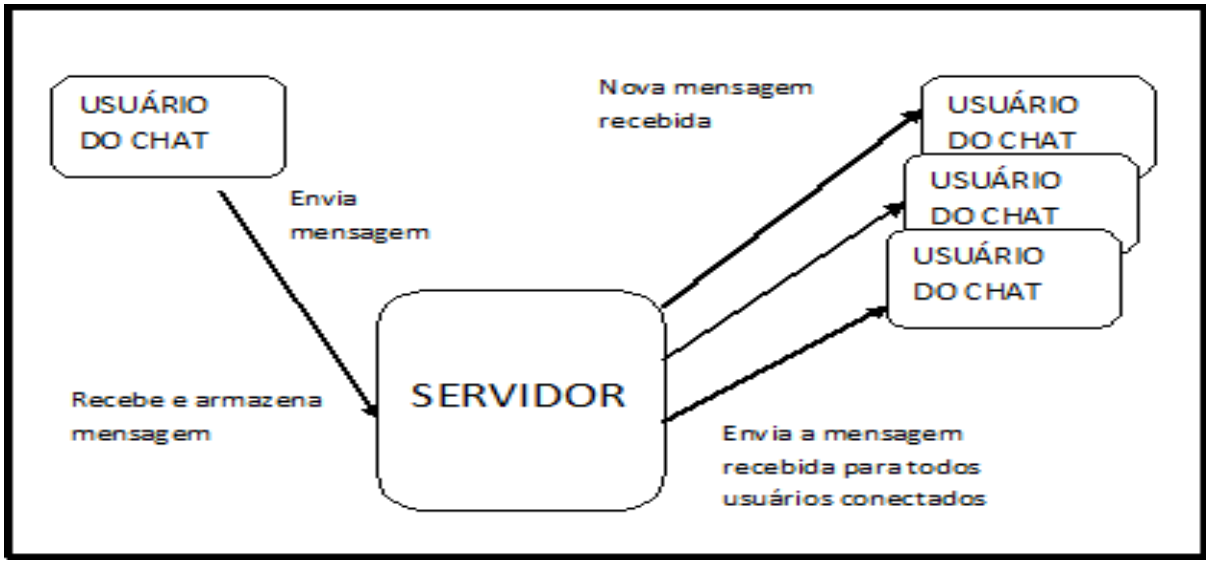

Figura 07 Diagrama de visão geral

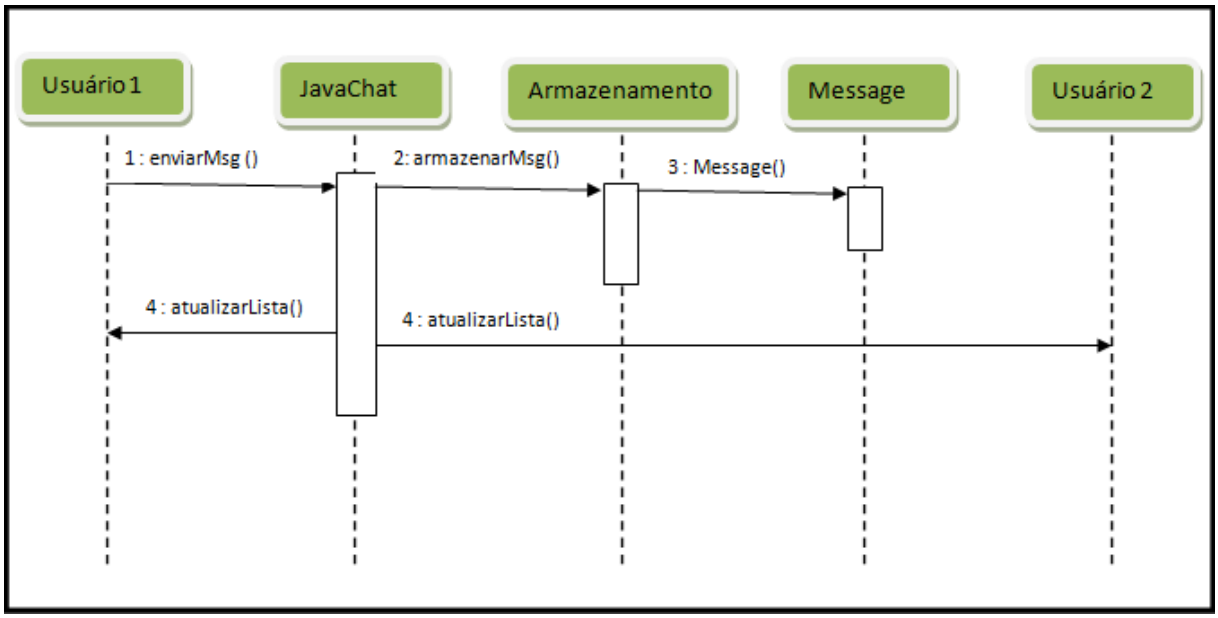

Figura 08 Diagrama de sequência

Por fim, as classes a serem implementadas no servidor para a implementação dos requisitos descritos foram mapeadas gerando o Diagrama de Classes, representado pela Figura 09.

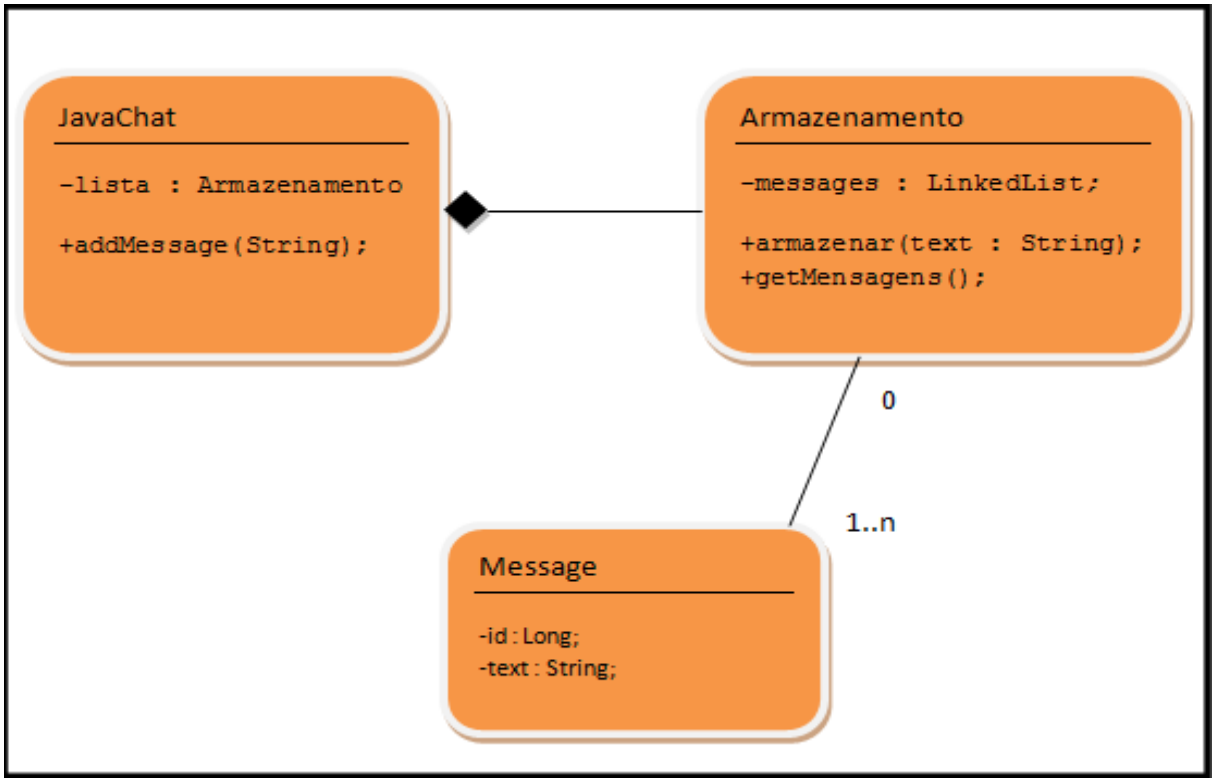

Figura 09 Diagrama de classes

Terminando a etapa de análise foram feitas as instalações e configurações necessárias para dar início às fases de desenvolvimento e testes.

3.3 INSTALAÇÕES E CONFIGURAÇÕES

Para todas as requisições o *DWR* realiza uma verificação de segurança CSRF (*Cross-Site Request Forgery*) através do uso de *cookies*. A partir do Tomcat 7 os cookies de sessão vem, por padrão, configurados como *HttpOnly*. Essa configuração impede que os cookies sejam acessados através de chamadas *Javascript*, causando erro no funcionamento do *DWR* que, sem o *cookie* enviado, terá a validação de segurança sempre retornando negativa.

Para contornar este problema podemos desativar a verificação do *DWR* ou configurar o arquivo de contexto do tomcat: `conf/context.xml` como no Quadro 02, para enviar os *cookies* normalmente.

```
<Context useHttpOnly=”false” >
```

Quadro 02 Configuração no contexto do container

Como citado acima, para o funcionamento do *framework DWR*, é necessário que o *servlet* do *DWR* seja configurado no arquivo `web.xml` da aplicação. Esta configuração, como mostrada no Quadro 03, define o “DwrServlet” como sendo receptor de todas as requisições feitas ao path `/dwr/*`. Também pode-se notar no quadro o parâmetro de inicialização “*pollAndCometEnable*” como verdadeiro. Este parâmetro de inicialização ativa a comunicação *Comet*.

```
<servlet>
<display-name>DWR Servlet</display-name>
<servlet-name>dwr-invoker</servlet-name>
<servlet-class>org.direct*WEB*remoting.servlet.DwrServlet</servlet-class>
<init-param>
<param-name>debug</param-name>
<param-value>true</param-value>
</init-param>
<init-param>
<param-name>pollAndCometEnabled</param-name>
<param-value>true</param-value>
</init-param>
</servlet>

<servlet-mapping>
<servlet-name>dwr-invoker</servlet-name>
<url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

Quadro 03 Configuração do Servlet *DWR* no *WEB*.xml

4 TESTES E RESULTADOS

A idéia inicial deste projeto foi de construir uma aplicação exemplo utilizando o framework *DWR* para prover as funções do Ajax Reverso e podendo melhorar o desempenho e consumo de recursos do servidor utilizando as novas especificações da JEE6.

4.1 JEE6 X *DWR*

Após estudar melhor o funcionamento interno do *framework* e as especificações da JEE6 foi identificado que não seria possível aproveitar os benefícios de ambos.

O *DWR* utiliza um *servlet* interno que deve ser configurado no `web.xml` da aplicação. Por este *servlet* é que transitaram todas as requisições Ajax realizadas pelo *framework* no lado cliente mantendo o controle das sessões clientes conectadas ao servidor.

A versão mais atual *DWR*, 3.0 RC1, foi lançada em meados de março de 2009, oito meses antes de terem sidos incorporados e anunciados as novas especificações da JSR 689 que contemplam não só, mas com importante atenção, as requisições assíncronas.

Para utilizar este recurso assíncrono, incluído nativamente na versão 6 da plataforma, seria necessário manipular as requisições em um *Servlet* próprio, que teria um código semelhante ao presente no Quadro 04.

Neste trecho de código, assim que o cliente requisita a página principal do *chat*, o servidor inicia o contexto assíncrono, mantendo a conexão em uma lista de clientes conectados.

```
private Queue<AsyncContext> clientes = new
                ConcurrentLinkedQueue<AsyncContext>();

protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                throws ServletException, IOException {
  AsyncContext asc = req.startAsync();
  asc.setTimeout(3000000);
  clientes.add(asc);
}
```

Quadro 04 Trecho de código implementado o contexto assíncrono com a JEE6

Um detalhe interessante na abordagem com os recursos da JEE6, é o mapeamento e configuração do *Servlet*, que pode ser feito através de anotações, mais uma novidade inserida nesta versão da plataforma. Este mapeamento com anotações pode ser observado analisando o Quadro 05, dando especial atenção ao parâmetro `asyncSupported`, o qual habilita o *Servlet* a trabalhar com o contexto assíncrono.

```
@WebServlet(urlPatterns={"/batepapo"}, asyncSupported=true)
public class ChatServlet extends HttpServlet {
....
}
```

Quadro 05 Mapeamento e configuração do *ChatServlet* permitindo o suporte assíncrono

Após o processo de “registro” dos usuários conectados junto ao servidor, assim que um destes envie uma mensagem o servidor iniciará a rotina de atualização da mensagem para todos conectados. Para isto, o servidor se utiliza da lista de clientes conectados, percorrendo um a um, acessando a conexão que foi estabelecida inicialmente e escrevendo a mensagem recebida. Seria possível implementar esta função em uma thread, que ficaria responsável por manter os clientes atualizados. Este função poderia ser algo como os códigos representados no Quadro 06.

```
while (true) {
  final String menssagem = getUltimaMensagem();
  for (final AsyncContext asc : clientes) {
    public void run() {
      PrintWriter writer = asc.getResponse().getWriter();
      writer.println(menssagem);
      writer.flush();
....
}
```

Quadro 06 Função responsável por enviar uma mensagem a todos usuário conectados

Apesar da eficácia da nova tecnologia foi escolhido o *framework DWR* por desempenhar as funções de *Ajax* Reverso propostas com facilidade, atendendo os objetivos do trabalho.

4.2 INÍCIO DA CONVERSAÇÃO PELO CLIENTE

Sempre que uma aplicação necessitar utilizar da comunicação remota com o *DWR*, ao carregar a página que utilizará a comunicação, que deverá incluir os *JavaScripts* do *framework*, a mesma deve chamar a função *setActiveReverseAjax* passando *true* como parâmetro, como demonstrado no Quadro 07. A chamada a esse método é que permitira o registro da sessão como conectada ao servidor.

```
<script type='text/javascript' src='/AppTcc/dwr/engine.js'></script>
<script type='text/javascript' src='/AppTcc/dwr/util.js'></script>

<body onload="dwr.engine.setActiveReverseAjax(true);">
```

Quadro 07 Código parcial da página índex.html

Para que o servidor possa se comunicar com o cliente, é necessário um *buffer* e para armazenar dados neste *buffer* o *DWR* criou o conceito de *ScriptingSession*. *ScriptingSession* seria algo como a sessão HTTP, esse script, cujo escopo esta associado a página corrente, é armazenado no servidor para posteriormente ser utilizado pela camada *JavaScript* do *DWR*.

Segundo os colaboradores do projeto *DWR* (DWR, 2011), sempre que um navegador carrega o script *"engine.js"* uma *SessionScript* é criada, e como pode-se visualizar nas listagens apresentadas nos Quadro 08 e Quadro 09, quando a função contida no Quadro 07 é acionada, o *DWR* ativa a estratégia *Comet* (*ActiveAjax*) enviando dados da mesma ao servidor. A partir deste momento o framework começa a manter este cliente conectado e armazenar esta conexão no servidor, junto dos outros navegadores conectados.

```
dwr.engine.setActiveReverseAjax = function(activeReverseAjax) {
  if (activeReverseAjax) {
    // Bail if we are already started
    if (dwr.engine._activeReverseAjax)
       return;
    // We always want a retry policy when reverse AJAX is enabled.
    dwr.engine._retryIntervals = dwr.engine._defaultRetryIntervals;
    dwr.engine._activeReverseAjax = true;
    dwr.engine._poll();
  }else {
    // Can we cancel an existing request?
    if (dwr.engine._activeReverseAjax && dwr.engine._pollBatch) {
      dwr.engine.transport.abort(dwr.engine._pollBatch);
    }
    dwr.engine._activeReverseAjax = false;
  }
};
```

Quadro 08 Código da função que ativa o AjaxReverso no *DWR*

```
dwr.engine._poll = function() {
  if (!dwr.engine._activeReverseAjax) {
    return;
  }
  dwr.engine._pollBatch = dwr.engine.batch.createPoll();
  dwr.engine.transport.send(dwr.engine._pollBatch);
};
```

Quadro 09 Código da função que define dados da estratégia e os envia para o servidor

Como citado na apresentação das tecnologias o *servlet* do *DWR* gera automaticamente um arquivo *JavaScript* para cada classe definida na configuração e através deste arquivo é possível realizar chamadas ao código como se fosse diretamente em Java. Com base nessa funcionalidade a página de *chat* inclui em seu conteúdo o arquivo "*JavaChat.js*", e o utiliza para enviar as mensagens do cliente conectado para o servidor. O código da página `chat.html` que implementa o lado cliente da aplicação é demonstrado com o Quadro 10.

```
<script type='text/javascript'
src='/AppTcc/dwr/interface/JavaChat.js'></script>

<script type="text/javascript">
function sendMessage() {
  JavaChat.addMessage(dwr.util.getValue("texto"));
}
</script>

</head>
<body onload="dwr.engine.setActiveReverseAjax(true);">

<h1>Java Chat</h1>

<p> Exemplo de uso do Ajax Reverso com um chat onde um conectado se
```

```
comunica com todos os outros. <p>

<p>
 Mensagem:
 <input id="texto" onkeypress="dwr.util.onReturn(event,sendMessage)" />
 <input type="button" value="Enviar" onclick="sendMessage()" />
</p>
<hr />
<h1 id="textApp"> </h1>
<ul id="listaMensagens" style="list-style-type: none;">
  <li>
    == -- Aqui serão listadas as mensagens enviadas por algum usuário no
chat -- ==.
  </li>
  <li>
     ==== --- Bem vindo --- ===
  </li>
</ul>
```

Quadro 10 Código parcial da página *chat*.html responsável por enviar e listar mensagens

4.3 COMUNICAÇÃO DO SERVIDOR PARA OS CLIENTES

Com a conexão ativa e “armazenada” no servidor, o próximo passo é fazer com que o servidor envie dados para todos os clientes a ele conectados. Utilizando as abstrações e os recursos do *DWR* esta tarefa parece trivial. A classe “*Browser*” traz várias possibilidades de se comunicação com as *ScriptSession* que estão ativas, uma delas, utilizada na aplicação exemplo, é a utilização do método “*whitCurrentPage(Runnable r)*”.

Quando invocado na situação acima citada, quando um usuário envia uma mensagem para todos conectados ao *chat*, o código presente no Quadro 11, que utiliza o método “*Browser.withCurrentPage*” terá efeito para todas as *ScriptSession* ativas. Na invocação do método é passada uma *Thread* que utiliza os recursos utilitários do *DWR* para atualizar a lista de mensagens.

As interfaces e recursos utilizados pelo *framework* são muito uteis e fáceis de utilizar. Para realizar a atualização dos dados no cliente é possível utilizar duas implementações diferentes, alterar diretamente o código da página ou realizar a chamada de uma função *JavaScript* da página para que esta altere.

```
Browser.withCurrentPage(new Runnable() {
   public void run() {
     Util.removeAllOptions("listaMensagens ");
     Util.addOptions("listaMensagens ", messages, "text");
   }
});
```

Quadro 11 Código parcial da classe Java*Chat* responsável por atualizar o HTML em todos os clientes conectados

```
Browser.withCurrentPage(new Runnable() {
   public void run() {
     ScriptSessions.addFunctionCall("receiveMessages", messages);
   }
});
```

Quadro 12 Código parcial da classe Java*Chat* responsável por invocar a função "receiveMessages()" em todos os clientes conectados

```
function receiveMessages(messages) {
  var chatlog = "";
  for (var data in messages) {
    chatlog = "<div>" + dwr.util.escapeHtml(messages[data].text)
    + </div>" + chatlog;
  }
  dwr.util.setValue("listaMensagens ", chatlog, { escapeHtml:false });
       }
```

Quadro 13 Código do método responsável por atualizar o HTML no cliente

Para alterar diretamente o conteúdo da página é preciso somente utilizar a interface *"Util"* do *DWR* ajustando o valor de um campo ou criando dinamicamente uma lista como demonstrado no Quadro 11. O *DWR* através da classe útil envia por texto as instruções com o conteúdo a ser executado no navegador e a parte *Client* interpreta o texto enviado e realiza as alterações por *JavaScript*.

O desenvolvedor pode optar pela segunda opção, deixando a responsabilidade das alterações de interface no lado *Client*, então é utilizado a interface "*ScriptSessions*" para realizar a chamada de uma função *JavaScript* que irá receber os dados atualizados passados como parâmetro e decidir como será realizada a alteração. A facilidade de codificação criada pode ser observada na Quadro 12 e Quadro 13.

O *framework DWR* facilita a comunicação Ajax do *JavaScript* com o código Java além de conter vários utilitários *JavaScript*. O mesmo ainda conta com tratamento de erros e até conversão de objetos *Java/Javascript*. Apesar de requerer uma boa manipulação de códigos *JavaScript*, o framework mostra inúmeras

vantagens se comparado ao que seria implementar uma aplicação Ajax pura ou uma aplicação *AjaxReverso* com JEE6 puro.

PiggyBack e *Polling* são estratégia a serem utilizadas quando não há necessidade de atualização das informações em tempo real. Pode se dar como exemplo aplicações como o email, no qual posso ser avisado que tenho novos emails de tempo em tempo ou até mesmo quando leio um dos emails da caixa de entrada, e o *Twitter* que também avisa o recebimento de novas mensagens regularmente ou quando realizo uma ação na página.

Para aplicações que necessitam da atualização de informações constantes como um *chat,* implementado neste trabalho, um Pregão Eletrônico, um visualizador de Cotações online, um jogo Online ou um editor colaborativo, a estratégia *Comet* é a mais indicada. Com a utilização de *Comet* a informação é atualizada para todos os clientes no mesmo momento em que foi alterada no servidor diminuindo o tempo de espera do utilizador e evitando trafego desnecessário, uma vez que só ocorre comunicação efetiva quando realmente é necessário.

Figura 10 Usuário conectado ao *chat*.

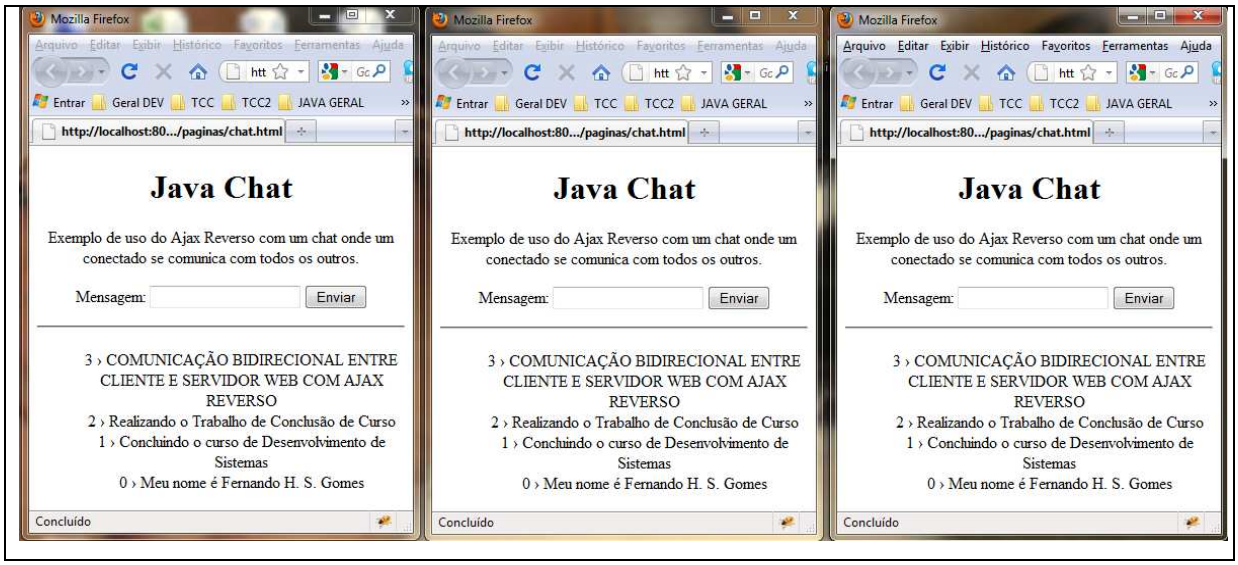

Figura 11 Atualização da informação para todos usuários conectados.

No exemplo criado neste trabalho, ao acessar a página de *chat*, visualizada na Figura 10, o usuário já esta conectado ao servidor e apto a receber novas mensagens, isso sem ter que atualizar a página nem ter que clicar em nenhum botão.

Quando ocorrer de um dos usuários conectados ao *chat* enviar uma mensagem, o servidor receberá a mesma e ficará responsável por “avisar” a todos sobre o novo evento, inclusive para o usuário que enviou a mensagem. A atualização da mensagem enviada por um dos usuários conectados em todos os navegadores conectados à página de chat da aplicação pode ser ilustrada com a Figura 11.

5 CONCLUSÃO

No trabalho foi estudado e apresentado a técnica *Ajax* Reverso e suas três técnicas de comunicação. O *Ajax* Reverso possui grande aplicabilidade nas aplicações que necessitam de trocar constantemente informações entre cliente e servidor. A utilização da solução reversa permite que o usuário receba a atualização da informação assim que esta ocorra e o mais importante, sem requisitar explicitamente a mesma para o servidor.

Com o desenvolvimento do trabalho foi possível aprender as técnicas de Ajax Reverso e entender o funcionamento do *DWR* e dos novos recursos da JEE6, constatando que o *DWR*, em sua versão 3.0, não pode usufruir de seus benefícios.

O exemplo utilizado neste trabalho para demonstrar a funcionalidade da aplicação denota a facilidade de uso do *framework DWR* ao implementar a técnica *Comet* provendo a “comunicação reversa” do servidor para com os clientes conectados.

Apesar de ser uma implementação anterior as especificações de contexto assíncrono o *DWR* demonstra ser bem robusto na forma como facilita a programação do Ajax Reverso, e ao mesmo tempo funcional.

5.1 TRABALHOS FUTUROS

Chegando nas etapas finais do projeto foi encontrada uma outra API que implementava a técnica *Comet* o *CometD* (*Comet Daily*). A princípio o framework parece utilizar a especificação JSR 567. Seria interessante um estudo detalhado do funcionamento do mesmo.

Outra situação que traria bons resultados seria a comparação de desempenho de uma aplicação implementada com *DWR* com a mesma aplicação implementada com os recursos da JEE6. Realizando testes com escalas variadas seria possível definir qual seria a melhor situação para utilizar uma ou outra solução.

REFERÊNCIAS

AMADEI, Cicero Daniel. Ajax Reverso, a Revolução. **Java Magazine**. Grajaú, ed. 49. a. VI, p.34-41, 2007;

CRANE, Dave; MCCARTHY, Phil. ***Comet and Reverse Ajax:*** *The Next-Generation Ajax* 2.0. Apress, 2008.

DWR. *Direct WEB Remoting – Easy* Ajax for Java. Disponível em <http://directwebremoting.org/dwr/index.html>. Acessado em 25 de Abril de 2011.

FLANAGAN, David. **JAVA: O Guia Essencial**.; tradução [da 3.ed. original] de Kátia Roque – Rio de Janeiro: Editora Campus, 2000.

GIL, Antonio Carlos. **Como elaborar projeto de pesquisa.** 5.ed. São Paulo: Atlas, 2010. 184p.

GLASSFISH, GlassFish Server. Disponível em <http://www.oracle.com/ technetwork/middleware/glassfish/overview/index.html>. Acessado em 25 de Abril de 2011.

JETTY, Jetty Server. Disponível em <http://www.eclipse.org/jetty/>. Acessado em 25 de Abril de 2011.

LUI, Xinyu. Solutions for Java Devlopers - Asynchronous processing support in Servlet 3.0. **Java World**, Fev. 2009. Disponível em <http://www.javaworld.com/javaworld/jw-02-2009/jw-02-servlet3.html/>. Acessado em 25 de Abril. 2011.

MICROSYSTEMS, SUN. **Tecnologia Java**. Disponível em <http://java.sun.com>. Acessado em 31 de Out. 2009.

ORACLE, Java. Disponível em < http://www.oracle.com/technetwork/java/index.html>. Acessado em 20 de Abril de 2011.

RUSSELL, Alex. Comet*: Low Latency Data for the Browser*. 2006. Disponível em <http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/>. Acessado em 29 de Abril de 2011.

SOARES, Wallace. **AJAX (Asynchronous JavaScript And XML)** Guia Prático. 2ª Edição. São Paulo: Érica, 2006.

TANEMBAUM, Andrew S. **SISTEMAS DISTRIBUÍDOS** princípios e paradigmas. 2ª Edição. São Paulo: Pearson, 2007.

UOL. BatePapo Uol. Disponível em <http://batepapo.uol.com.br/>. Acessado em 10 de maio de 2011.