



UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E
DE COMPUTAÇÃO



Desenvolvimento de um middleware para comunicação via web services e sua aplicação em sistemas de aquisição de dados industriais

Rivaldo Rodrigues Machado Júnior

Orientador: Prof. Dr. Luiz Affonso Henderson Guedes de Oliveira

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e de Computação da UFRN (área de concentração: Engenharia de Computação) como parte dos requisitos para obtenção do título de Mestre em Ciências.

Natal, RN, janeiro de 2014

Catálogo da Publicação na Fonte. EMATER / Biblioteca Central

M149d

Machado Júnior, Rivaldo Rodrigues.

Desenvolvimento de um Middleware para Comunicação via Web Services e sua Aplicação em Sistemas de Aquisição de Dados Industriais / Rivaldo Rodrigues Machado Júnior. – Natal, RN, 2014.

88f.

Orientador: Luiz Affonso Henderson Guedes de Oliveira

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte.
Centro de Tecnologia. Programa de Pós-Graduação em Engenharia Elétrica e de Computação.

1. Comunicação industrial - Dissertação. 2. Padrão OPC – Dissertação. 3. Informática Industrial – Dissertação. 4. Programação Web – Dissertação I. Oliveira, Luiz Affonso Henderson Guedes de. II. Universidade Federal do Rio Grande do norte. III. Título.

RN/ EMATER/ BIBLIOTECA

CDU: 004.42:681.3

Desenvolvimento de um middleware para comunicação via web services e sua aplicação em sistemas de aquisição de dados industriais

Rivaldo Rodrigues Machado Júnior

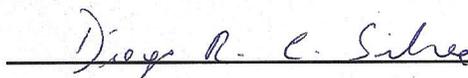
Dissertação de mestrado aprovada em 24 de Janeiro de 2014 pela banca examinadora
composta pelos seguintes membros:



Prof. Dr. Luiz Affonso H Guedes de Oliveira (Presidente) DCA/UFRN



Prof. Dr. Luiz Marcos Garcia Gonçalves (Interno) DCA/UFRN



Prof. Dr. Diego Rodrigo Cabral Silva (Externo ao Programa) ECT/UFRN



Prof. Dr. Fernando Menezes Matos (Externo a Instituição) UFPB

*“O insucesso é apenas uma
oportunidade para recomeçar de
novo com mais inteligência.”*

Henry Ford

*A Deus, cuja forte mão estava (e
estará) sempre estendida para me
levantar quando eu cair. Pelo
conforto nos momentos difíceis
dessa jornada e por estar sempre ao
meu lado, mesmo que por vezes eu
estivesse distante. Tudo o que tenho,
tudo o que sou, e o que vier a ser,
vem de ti Senhor.*

AGRADECIMENTOS

Aos meus pais, pelo apoio incondicional ao longo de toda minha vida.

Ao professor Luiz Affonso pelo incentivo e orientação acadêmica e pessoal.

À todos os professores e funcionários do Departamento de Engenharia de Computação e Automação, pelo aprendizado e pela ajuda.

Aos colegas de mestrado.

Ao Gustavo e Juliano pelo conhecimento e experiência de vida compartilhados.

À todos os demais companheiros do LII e da Logique.

RESUMO

O controle de processos industriais têm se tornado cada vez mais complexo devido à diversidade de equipamentos de chão-de-fábrica, exigência na qualidade e concorrência de mercado. Tal complexidade exige que, uma grande quantidade de dados seja tratada pelos três níveis de controle de processo: dispositivos de campo, sistemas de controle e software para gerenciamento. Utilizar de forma efetiva os dados presentes em cada um desses níveis é de fundamental importância para indústria.

Muitos dos sistemas computacionais industriais de hoje são compostos de sistemas distribuídos de software, escritos em uma grande variedade de linguagens de programação e desenvolvidos para plataformas específicas. Desta forma, cada vez mais, pequenas e grandes empresas aplicam um investimento significativo para manter ou até mesmo re-escrever seus sistemas para diferentes plataformas. Além disso, é raro que um sistema de software seja executado em completo isolamento. Na área de automação industrial é comum que sistemas de software interajam com outros sistemas em diferentes máquinas e até mesmo escritos em diferentes linguagens. Tendo isto em vista, interoperabilidade não é apenas um desafio a longo prazo, mas também uma exigência do contexto atual de produção de softwares industriais.

Este trabalho visa propor uma solução de *middleware* para comunicação de aplicações via web service, além de apresentar um estudo de caso aplicando a solução desenvolvida a um sistema integrado para captura de dados industriais, permitindo assim que tais dados sejam disponibilizados de maneira simplificada e independente de plataforma através da rede.

Palavras-chave: CyberOPC, OPC UA, REST, SOAP, Web Service, Sistemas distribuídos, Middleware

ABSTRACT

The control of industrial processes has become increasingly complex due to variety of factory devices, quality requirement and market competition. Such complexity requires a large amount of data to be treated by the three levels of process control: field devices, control systems and management softwares. To use data effectively in each one of these levels is extremely important to industry.

Many of today's industrial computer systems consist of distributed software systems written in a wide variety of programming languages and developed for specific platforms, so, even more companies apply a significant investment to maintain or even re-write their systems for different platforms. Furthermore, it is rare that a software system works in complete isolation. In industrial automation is common that, software had to interact with other systems on different machines and even written in different languages. Thus, interoperability is not just a long-term challenge, but also a current context requirement of industrial software production.

This work aims to propose a middleware solution for communication over web service and presents an user case applying the solution developed to an integrated system for industrial data capture , allowing such data to be available simplified and platform-independent across the network.

Keywords: CyberOPC, OPC UA, REST, SOAP, Web Service, Distributed Systems, Middleware

SUMÁRIO

Sumário	i
Lista de Figuras	iv
Lista de Tabelas	vi
Lista de Símbolos e Abreviaturas	vii
1 Introdução	1
1.1 Introdução	1
1.2 Sistema de aquisição de dados - BR-Coletor	3
1.2.1 Aspectos de robustez	5
1.2.2 Comunicação	5
1.3 Visão geral do documento	5
1.3.1 Objetivos	5
1.3.2 Estrutura	6
2 Tecnologias de comunicação de dados	8
2.1 Modelo de objetos distribuídos	8
2.1.1 Java RMI	9
2.2 Web Services	10
2.2.1 REST - <i>Representational State Transfer</i>	11
2.2.2 SOAP - <i>Simple Object Access Protocol</i>	12
2.3 Comunicação assíncrona e Java Servlets	14
2.3.1 Notificação do servidor (<i>Server push</i>)	15
2.3.2 Servlet 3.0	16

2.4	Soluções industriais	16
2.4.1	OPC	16
2.4.2	CyberOPC	18
3	Middleware - Arquitetura Proposta	20
3.1	Motivação	20
3.2	Proposta	22
3.2.1	Visão geral da arquitetura	22
3.2.2	Objetos Remotos	23
3.2.3	Estilo de comunicação remota - JSON-RPC	24
3.2.4	Módulos	25
4	Testes de Desempenho	40
4.1	Testes de Desempenho	40
4.1.1	Hardware utilizado	40
4.1.2	Cenário de teste	41
4.1.3	Resultados obtidos	42
5	Caso de uso - Sistema de Aquisição de Dados Industriais	45
5.1	Estado da arte	45
5.2	Arquitetura web aplicada ao BR-Collector	47
5.2.1	Servidor web do BR-Collector	48
5.2.2	Cliente web do BR-Collector	51
5.2.3	Cliente web C#	51
5.2.4	Aplicação cliente em C#	54
5.3	Supervisório para dispositivos com sistema operacional Android	56
6	Conclusão	58
	Apêndice A - BR-Collector-webAPI: ServiceRDA	60
	Apêndice B - Diagrama de classes do Middleware	63
.1	Diagrama de classes por pacote	64
.1.1	Client	64
.1.2	Domain	65
.1.3	Exception	65
.1.4	Server	66
.1.5	Typechecker	67
.1.6	Util	68

Apêndice C - Diagrama de classes de domínio	69
.2 Diagrama de classes trafegadas e suas representações em JSON	69
.2.1 Server	69
.2.2 TagItem	70
.2.3 TagInfo	71
.2.4 Callback RDA	72
.2.5 ProcessVariableSubscription	72
.2.6 TagItemGroup	73
.2.7 DataRDA	73
.2.8 GenericDataRDA	74
Apêndice D - Exemplo de requisições HTTP do BR-Collector	75
.3 Requisições e respostas via JSON-RPC	75
.3.1 Método listServer	75
.3.2 Método connectServer	75
.3.3 Método supportsHierarchialListing	75
.3.4 Método browseNode	76
.3.5 Método searchTagItems	76
.3.6 Método getTagInfos	76
.3.7 Método createSubscription	76
.3.8 Método addItemToSubscription	77
.3.9 Método createGroup	77
.3.10 Método addItemToGroup	77
.3.11 Método writeValues	77
.3.12 Método getCurrentValues	77
Apêndice E - Diagrama de classes do cliente C#	79
.4 Diagrama de classes por pacote	80
.4.1 JSONRPC	80
.4.2 Util	80
.4.3 WebClient	81
Apêndice F - BR-Collector-webAPI: Cliente C#	82
Referências Bibliográficas	86

LISTA DE FIGURAS

1.1	Arquitetura de comunicação OPC.	3
1.2	Arquitetura do BR-Collector.	4
1.3	Interface de configuração do BR-Collector.	6
2.1	Arquitetura Web Service baseado em SOAP.	12
2.2	Aplicações OPC UA.	18
3.1	Camada de comunicação.	21
3.2	Arquitetura geral.	23
3.3	Sequencia de validação de interface.	26
3.4	Fluxo do solicitante.	28
3.5	Proxy do cliente.	29
3.6	Fluxo de requisição síncrona.	32
3.7	Fluxo de requisição síncrona.	34
3.8	Fluxo de invocação.	35
3.9	Callback para comunicação assíncrona.	37
3.10	Fluxo de execução assíncrona do cliente.	38
3.11	Fluxo de execução assíncrona do servidor.	39
4.1	Diagrama de classe – Server.	41
4.2	Diagrama de classe – TagItem.	42
4.3	Comparação JETY x RMI.	43
4.4	Tempo médio durante os testes.	44
4.5	Tempo médio por requisição.	44
5.1	Componentes do <i>Server Toolkit</i>	46

5.2	Fluxo para coleta de dados de variáveis de processo no BR-Collector.	47
5.3	Modelo da arquitetura proposta.	48
5.4	Diagrama das classes JsonRequest e JsonResponse.	53
5.5	Aplicativo Industrial Supervisor no Modo de Execução.	56
1	Classes do pacote Client.	64
2	Classes do pacote Domain.	65
3	Classes do pacote Exception.	65
4	Classes do pacote Server.	66
5	Classes do pacote Typechecker.	67
6	Classes do pacote Util.	68
7	Classe Server	69
8	Classe TagItem	70
9	Classe TagInfo	71
10	Classe CallbackRDA	72
11	Classe ProcessVariableSubscription	72
12	Classe TagItemGroup	73
13	Classe DataRDA	73
14	Classe GenericDataRDA	74
15	Classes do pacote JSONRPC.	80
16	Classes do pacote Util.	80
17	Classes do pacote WebClient.	81

LISTA DE TABELAS

2.1	Métodos HTTP e operações CRUD.	11
3.1	Parâmetros do cabeçalho HTTP.	33
4.1	Resultado do tempo médio por requisições.	43

LISTA DE SÍMBOLOS E ABREVIATURAS

API	Application Programming Interface
CENPES	Centro de Pesquisas da Petrobras
COM	Component Object Model
DCOM	Distributed Component Object Model
DDE	Dynamic Data Exchange
DLL	Dynamic Link Library
EPS	Enterprise Production Systems
ERP	Enterprise Resource Planning
JDK	Java Development Kit
JNA	Java Native Access
JNI	Java Native Interface
JVM	Java Virtual Machine
MES	Manufacturing execution system
OLE	Object Linking and Embedding
OPC	OLE for Process Control
PIMS	Plant Information Management System

SCADA	Supervisory Control and Data Acquisition
SDCD	Sistema Digital de Controle Distribuído
SO	Sistema Operacional
TCP	Transmission Control Protocol
UFRN	Universidade Federal do Rio Grande do Norte
UML	Unified Modeling Language
XML	eXtensible Markup Language

INTRODUÇÃO

1.1 Introdução

Com o passar dos anos, os processos industriais foram ficando cada vez mais complexos exigindo assim uma evolução dos sistemas de automação. Com isso, surge a necessidade de utilizar soluções de software para comunicação com dispositivos (*drivers*) bem como aplicações para supervisão, controle, calibração e configuração remota de instrumentos de campo. Tal necessidade, impulsionou ainda a criação de aplicações para gerenciamento de grandes quantidades de informações obtidas do campo e transmitidas para as salas de controle, bem como a geração de informações úteis para outros setores da empresa [Gutierrez & Pan 2008].

Os sistemas de gerência de informação industrial, que são englobados com o termo geral de *Enterprise Production Systems* (EPS), onde estão incluídos os *Plant Information Management System* (PIMS) e os *Manufacturing Execution Systems* (MES), atuam como um intermediário entre o chão-de-fábrica e os sistemas corporativos de gestão da planta *Enterprise Resource Planning* (ERP), responsáveis pela transformação desses dados em informações de negócio. Apesar de serem geralmente responsáveis pela coleta e disponibilização de dados do chão-de-fábrica, eles desempenham diferentes papéis nesse contexto [Carvalho 2004].

Neste cenário em constante desenvolvimento, um empecilho para a reutilização de módulos de software tem sido a incompatibilidade entre interfaces de comunicação em equipamentos industriais, gerando um gasto adicional de tempo e dinheiro para adaptar o software a diferentes interfaces.

Muitos dos sistemas computacionais empresariais de hoje são compostos por sistemas distribuídos que, muitas vezes, efetuam operações que são críticas para um determinado negócio. Devido a isso, cada vez mais e mais sistemas utilizam-se de soluções de *middleware* de objetos distribuídos, a fim de prover aos mesmos um melhor desempenho, confiabilidade e interoperabilidade.

Na década de 1990 sugeriram várias abordagens comercialmente bem-sucedidas e populares de objetos distribuídos como, por exemplo, o *Common Object Request Broker Architecture* (CORBA), promovido pelo *Object Management Group* (OMG) e o *Common Object Model* (COM) da Microsoft. O CORBA foi projetado especificamente para resolver a inerente heterogeneidade das redes de computação de negócios, onde há uma grande variedade de tipos de máquinas, sistemas operacionais, linguagens de programação e estilos de aplicação que devem coexistir e cooperar. Já o COM, por outro lado, foi construído para suportar aplicações orientadas a componentes que executam no sistema operacional Microsoft Windows. Hoje, o COM tem sido substituído pelo seu sucessor (.NET), enquanto o CORBA continua sendo utilizado como uma arquitetura para a construção e implantação de sistemas heterogêneos em escala empresarial, bem como sistemas de tempo real e integrado [Völter et al. 2005]. Apesar do fato de que, o CORBA e o COM foram projetados para fins fundamentalmente diferentes, eles compartilham uma série de semelhanças. Estas semelhanças vão desde noções básicas, incluindo aplicações de objetos remotos, clientes e servidores, *proxies*, empacotadores, comunicação síncrona e assíncrona, descrições de interfaces e gerenciamento de ciclo de vida. Não é de surpreender, no entanto, que essas semelhanças possam ser encontradas em outras tecnologias e abordagens, incluindo o .NET, Java 2 Enterprise Edition [Oracle 2013a], e até mesmo em web services.

No âmbito industrial, como solução para o problema de interoperabilidade, surge a idéia de padronização das interfaces de comunicações proposta pela *OPC Foundation* [OPC Foundation 2013]. A *OPC Foundation* é uma organização cujo objetivo principal é prover a comunicação entre aplicações industriais. Para isso, a fundação mantém um conjunto de especificações que visam padronizar a comunicação entre diversos dispositivos de fabricantes diferentes e até mesmo situados em diferentes redes industriais.

O padrão OPC consegue unir diferentes áreas da empresa com objetivo de disponibilizar dados integrando os diversos níveis arquiteturais da mesma, de forma a gerar dados de forma homogênea (vide figura 1.1). Assim, os níveis arquiteturais mais altos tem acesso a dados de todos os níveis da indústria de forma mais precisa.

Conhecida como OPC UA (*Unified Architecture*), a última evolução do OPC pretende unificar todas as especificações anteriores, além de permitir uma implementação livre de linguagem. O OPC UA baseia-se na tecnologia web service, provendo assim uma forma

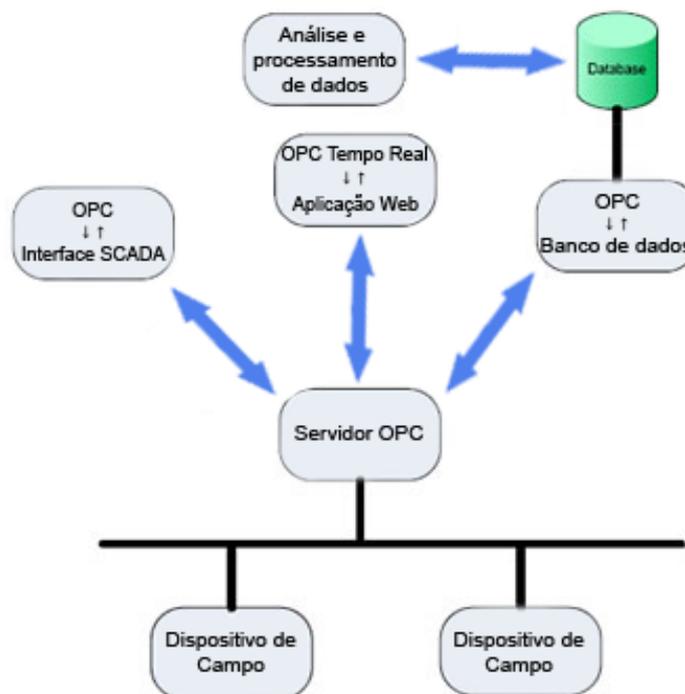


Figura 1.1: Arquitetura de comunicação OPC.

mais eficaz de comunicação vertical e independente de plataforma, em contraste com as tecnologias utilizadas nas especificações anteriores que restringiam o desenvolvimento à linguagem de programação que permitiam o acesso direto ao sistema operacional Windows.

Apesar do grande avanço obtido pela padronização proposta pela plataforma OPC, ainda se faz necessário um investimento considerável para desenvolvimento e manutenção de sistemas ERP, uma vez que, por se tratar de um padrão que visa unificar todos os níveis da indústria, muitas das definições apresentadas na especificação OPC UA, atualmente composta por onze partes, são confusas e até mesmo desnecessárias para a camada de aplicação.

1.2 Sistema de aquisição de dados - BR-Coletor

Visando aumentar a eficiência, precisão e eficácia dos procedimentos de diagnósticos sobre operações de processos industriais, é de suma importância a integração das informações de processo (alarmes, eventos, variáveis contínuas, etc.) com informações sobre o processo em si (aspectos de projeto, conhecimento sobre a operação dos processos, histórico de anomalias e históricos de manutenção de equipamentos). Tal cenário se

mostra fortemente presente em processos complexos, como a operação de sistemas de produção de petróleo e gás a longas distâncias onde o uso de soluções via software tem se mostrado cada vez mais presente na área de automação, para efetuar tarefas tais como visualização, arquivamento ou controle de dados [Carvalho et al. 2008].

Com a crescente aplicação de técnicas de automação em plantas industriais, cada vez mais se torna necessário o tratamento do grande volume de informações provenientes de diversas fontes de aquisição de dados. A integração dessas informações aos sistemas corporativos, possibilita a otimização da gerência, refletindo positivamente nos aspectos financeiros [Leitão 2008].

Nesse contexto, surge a ferramenta BR-Coletor, desenvolvida através de uma parceria entre a UFRN e a Petrobras. O BR-Collector é um produto criado para integrar a captura de dados em um ambiente industrial e disponibilizá-los de forma simplificada. Atualmente o BR-Collector suporta captura de dados de processo histórico e em tempo real, captura de dados de alarmes e eventos históricos e em tempo real.

Concebido de maneira modular, o BR-Collector possui uma arquitetura dividida em *slots* e *drivers*. Os *drivers* são responsáveis por implementar os detalhes de cada protocolo de comunicação e realiza interface com o BR-Collector através dos *slots*. Os *slots*, por sua vez, têm como função padronizar a forma como o BR-Collector entende cada tipo de dado. Atualmente são disponibilizados quatro *slots* e quatro *drivers* conforme demonstra a figura 1.2.



Figura 1.2: Arquitetura do BR-Collector.

À medida que necessário, novos *drivers* podem ser acoplados a essa arquitetura sem qualquer alteração nas camadas superiores.

Além da integração de dados, o BR-Collector permite realizar uma série de otimizações de comunicação de dados. O BR-Collector administra, por exemplo, todas as conexões realizando a operação de junção. Ou seja, se um aplicativo está solicitando a leitura de uma determinada variável e outro desejar esse mesmo dado, O BR-Collector

não enviará uma nova requisição de leitura às camadas inferiores diminuindo o tráfego de rede quando várias aplicações observam um mesmo dado.

1.2.1 Aspectos de robustez

Visando evitar ao máximo a perda de dados, o BR-Collector incorpora uma lógica de redundância. Ao configurar o BR-Collector para captura de dados, o usuário pode optar por adicionar um servidor secundário o qual entrará em ação no momento em que o primário ficar indisponível, superando assim eventuais falhas de operação e evitando perda de dados. Tal operação ocorre de forma totalmente transparente ao usuário, ou seja, toda a configuração pré-existente no servidor primário é transferida para o secundário automaticamente. O sistema se encarrega ainda de chavear novamente para o servidor primário tão logo este volte à ativa.

A figura 1.3 apresenta a interface do BR-Collector configurada para aquisição de dados de histórico e em tempo real provenientes de diversos servidores configurados de forma redundante. Através desta tela, o BR-Collector fornece ainda uma série de informações úteis como o status de cada servidor, número de requisições com falha, data da última requisição com sucesso entre outras.

1.2.2 Comunicação

Atualmente o BR-Collector utiliza a tecnologia *Java Remote Method Invocation* (RMI) como forma de comunicação com as máquinas clientes. Assim, para se ter acesso a todas as funcionalidades disponibilizadas pelo BR-Collector é necessário que a aplicação cliente seja implementada também em Java, e realize primeiramente o processo de conexão com o RMI. Após a conexão ter sido realizada o cliente recebe uma referência para a interface *Request* onde estão todos os métodos de acesso as funcionalidades do BR-Collector. Desse ponto em diante o acesso a todos os dados se dará através de chamadas de métodos da classe *Request*. Para o usuário, a partir desse momento, a comunicação remota entre a máquina cliente e servidora é totalmente abstraída.

1.3 Visão geral do documento

1.3.1 Objetivos

O objetivo deste trabalho é propor uma solução ao desafio de comunicação proposto provendo uma API que possibilite o fácil desenvolvimento de aplicações distribuídas que

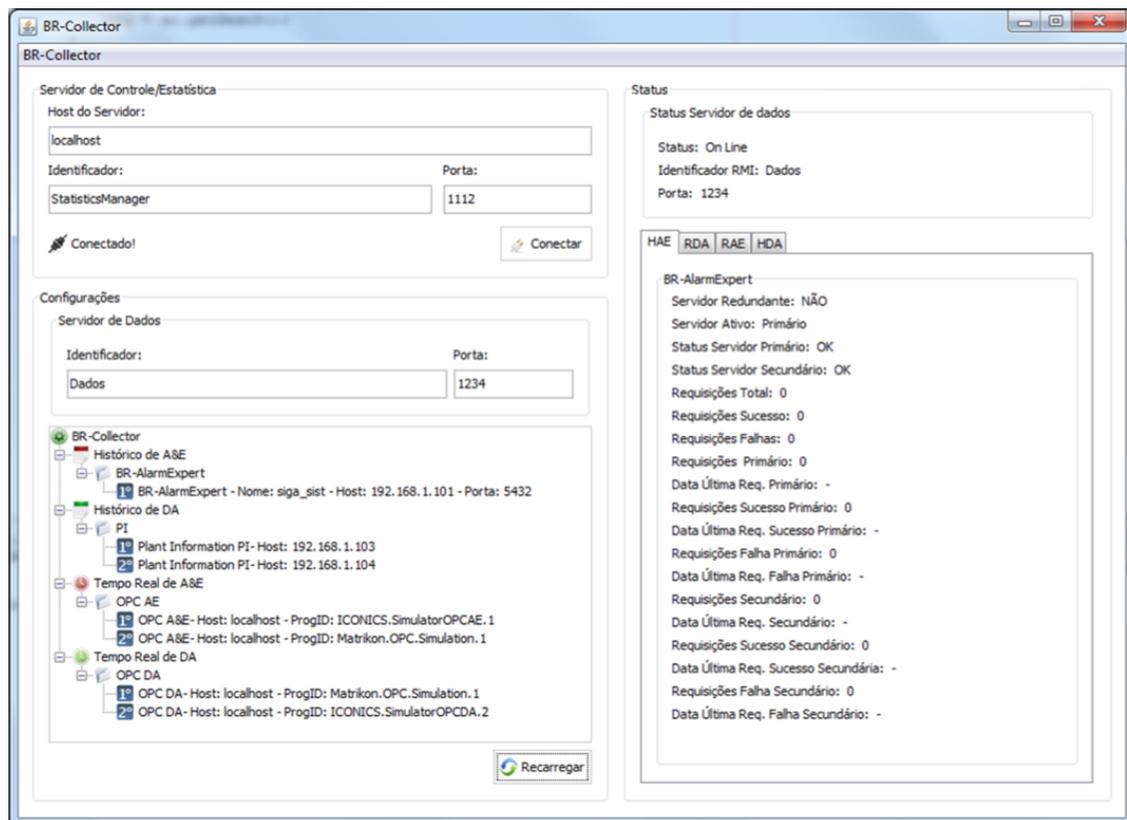


Figura 1.3: Interface de configuração do BR-Collector.

utilizam-se da tecnologia web service. Tal solução consiste em um *middleware* para comunicação com serviços web via Java Servlet que se destaca das demais soluções presentes no mercado por possibilitar a comunicação assíncrona entre cliente e o servidor, atributo este de extrema importância para aplicações industriais. Este trabalho visa ainda efetuar um estudo de caso aplicando a solução desenvolvida à ferramenta BR-Collector a fim de permitir o desenvolvimento de aplicações industriais que possibilitem a aquisição de dados provenientes de diversas fontes de forma simplificada e independente de plataforma.

1.3.2 Estrutura

A dissertação é estruturada da seguinte forma:

No capítulo 2 são apresentados os protocolos para comunicação de dados, com ênfase na solução web service, bem como as soluções para aquisição de dados industriais presentes no mercado, apresentando as principais características de cada tecnologia.

No capítulo 3 são descritos os detalhes da implementação da solução proposta bem como o uso das tecnologias envolvidas e utilizadas no desenvolvimento do *middleware*.

No capítulo 4 são apresentados os teste de performance realizado sobre o *middleware* desenvolvido.

O capítulo 5 apresenta um caso de uso do *middleware* desenvolvido, em forma de um modelo de arquitetura para o BR-Collector, que visa prover uma forma de aquisição de dados industriais em diferentes plataformas.

Por fim, o capítulo 6 apresenta as considerações finais deste trabalho.

TECNOLOGIAS DE COMUNICAÇÃO DE DADOS

2.1 Modelo de objetos distribuídos

O fantástico crescimento da Web aliado à alta velocidade de acesso à rede contribuíram para levar a computação distribuída a uma posição de destaque no atual cenário da área de processamento de dados. A fim de simplificar a programação em rede e aplicar uma arquitetura baseada em componentes de software, dois modelos de objetos distribuídos foram propostos como padrões, sendo largamente utilizados até os dias atuais. São eles: o COM/DCOM [Microsoft 2013] e CORBA [OMG 2013].

Criada pela Microsoft, a tecnologia COM tem por função definir componentes que possam ser reutilizados por diversas aplicações. Este princípio é utilizado, por exemplo, quando se insere uma tabela do Microsoft Excel no Microsoft Word, possibilitando assim a troca de informações em tempo real [Leitão 2006].

A tecnologia COM permite que aplicações clientes possam acessar os serviços de uma aplicação servidora, que os implementa em forma de objetos, especificando interfaces (uma coleção de métodos ou funções e procedimentos) para a comunicação padronizada. Já a tecnologia DCOM (*Decentralized Component Object Model*), provê as mesmas funcionalidades do COM, porém, utiliza uma rede de comunicação de dados como meio de acesso [Cândido 2004].

Um objeto COM consiste em um código binário que pode ser utilizado por outras aplicações, seja qual for a linguagem de programação utilizada no desenvolvimento da

mesma. Uma vez criado, o objeto COM funciona como uma caixa preta, permitindo que o desenvolvedor os utilize sem conhecer os detalhes das implementações dos mesmos.

CORBA é um *framework* de objetos distribuídos proposto por um grupo, formado por mais de 700 empresas, conhecido como OMG. O núcleo da arquitetura CORBA é o ORB (*Object Request Broker*) que atua permitindo que objetos interajam uns com os outros de forma transparente, estejam eles localizados em uma mesma máquina ou mesmo remotamente [Vinoski 1997]. Os ORBs manipulam a transformação das estruturas de dados internas aos processos de/para uma sequência de bytes que, por fim, é transmitida pela rede. Este processo é chamado de triagem ou serialização.

Um objeto CORBA é representado por uma interface com um conjunto de métodos. O cliente de um objeto CORBA adquire o seu objeto de referência e usa-o como um identificador para fazer chamadas de métodos como se o objeto estivesse localizado no cliente. O ORB é responsável por todos os mecanismos necessários para encontrar a execução do objeto, prepará-lo para receber o pedido, comunicar o pedido a ele, e levar a resposta ao cliente.

Tanto a tecnologia COM/DCOM quanto a CORBA possuem uma estrutura de comunicação do tipo cliente/servidor. Para solicitar um serviço, um cliente invoca um método implementado por um objeto remoto que atua como o servidor. O serviço prestado pelo servidor é encapsulado em um objeto e sua interface é descrita por um arquivo IDL (*Interface Definition Language*). As interfaces definidas em um arquivo IDL servem como um contrato entre um servidor e seus clientes. Os clientes interagem com um servidor invocando métodos descritos na IDL de forma que, a implementação de objeto real permanece transparente para o cliente [Chung et al. 1998].

2.1.1 Java RMI

A tecnologia RMI é a resposta da linguagem de programação Java para o problema de comunicação distribuída. De forma semelhante ao que acontece no CORBA, a idéia básica do RMI é obter um conjunto de objetos colaboradores que se comuniquem através de uma rede.

O RMI possibilita que o desenvolvedor efetue a comunicação distribuída entre aplicações Java, onde métodos de objetos localizados em um servidor remoto podem ser chamados através de diferentes JVMs (*Java virtual machine*) [Oracle 2013b].

Para invocar um método RMI é necessário criar um objeto denominado *stub* que funciona como um *proxy* que encapsula o pedido do cliente em um pacote de bytes. Tal pedido é composto por:

- Um identificador do objeto remoto a ser usado;

- Um número de operação para descrever o método a ser invocado;
- Um conjunto de parâmetros serializados.

Para o servidor receber a informação, ele cria um objeto *skeleton*, o qual executa as seguintes funções:

- Decodifica os parâmetros;
- Chama o método desejado;
- Recebe o valor de retorno e serializa o mesmo;
- Retorna o valor codificado para o cliente.

A partir da separação do conceito de interface e implementação da linguagem Java, o RMI permite que a interface e a respectiva implementação se localizem em JVM's diferentes. O RMI torna possível que uma determinada aplicação cliente adquira uma interface (que define o comportamento) referente a uma classe (que contém a implementação) em uma JVM diferente. A comunicação entre interface e a implementação é assegurada pelo RMI via TCP/IP.

2.2 Web Services

Apesar de eficientes, as antigas tecnologias de modelos de objetos distribuídos apresentam um alto grau de dificuldade de implementação, bem como um custo elevado para manutenção de aplicações baseadas nas mesmas [Hochgurtel 2003]. Em virtude disto, empresas como Microsoft, IBM e Sun Microsystems têm cada vez mais investido nos serviços web como a próxima grande tecnologia para permitir a fácil criação de objetos remotos por parte dos desenvolvedores.

Como o nome sugere, um serviço web é um tipo de aplicação web, ou seja, uma aplicação normalmente entregue sobre o protocolo HTTP (*Hyper Text Transport Protocol*). Um serviço web é, portanto, uma aplicação distribuída, cujos componentes podem ser implementados e executados em dispositivos distintos como por exemplo PCs, *handhelds* e outros dispositivos.

Em geral, os serviços web podem ser divididos basicamente em dois grupos: os serviços baseados em SOAP (*Simple Object Access Protocol*) e os REST (*Representational State Transfer*). Apesar da separação, a distinção entre esses grupos não é nítida, pois, como será explanado nas sessões subsequentes, um serviço baseado em SOAP entregue sobre HTTP é um caso especial dos serviços REST.

2.2.1 REST - *Representational State Transfer*

O Termo REST apareceu pela primeira vez descrito em Fielding (2000). O REST por si só não representa uma arquitetura, mas sim um conjunto de restrições que, quando aplicadas na concepção de um sistema, cria um estilo de arquitetura de software. Um sistema no estilo REST é denominado RESTful e possui as seguintes características:

- Deve ser um sistema cliente-servidor.
- Tem que ser independente de estado, ou seja, cada requisição deverá ser independente das outras.
- Tem que suportar um sistema de cache, a infraestrutura de rede deve suportar caches em diferentes níveis.
- Tem que ser uniformemente acessível, cada recurso deve ter um endereço exclusivo e um ponto de acesso válido.
- Tem que ser em camadas e deve suportar escalabilidade.

Um sistema RESTful pode ser implementado em qualquer arquitetura de rede disponível de forma que, não se faz necessário a criação de novas tecnologias ou protocolos de rede.

Enquanto o SOAP é um protocolo de mensagens, o REST é um estilo de arquitetura de software para sistemas hipermídia distribuídos denominados recursos. Um recurso RESTful é qualquer coisa que possa ser endereçável pela web, isto é, sistemas em que o texto, gráficos, áudio e outras mídias são armazenadas em uma rede e interconectadas através de hiperlinks podendo ser acessados e transferidos entre clientes e servidores.

O núcleo da abordagem REST consiste na percepção de que, apesar termo transporte em seu nome, o HTTP consiste em uma API e não um simples protocolo de transporte. Desta forma, o HTTP possui métodos bem definidos que correspondem as operações CRUD (*Create, Read, Update, Delete*). Cada requisição HTTP inclui um dos métodos apresentados na tabela 2.1 para indicar qual a operação CRUD que deve ser realizada sobre o recurso.

Método HTTP	Operação
POST	Cria um novo recurso a partir dos dados requisitados
GET	Lê um recurso
PUT	Atualiza um recurso a partir dos dados requisitados
DELETE	Remove um recurso

Tabela 2.1: Métodos HTTP e operações CRUD.

Em termos gerais, um cliente RESTful emite um pedido que envolve um recurso, como por exemplo, um pedido de leitura. Se este pedido for bem sucedido, uma representação do recurso é transferido do servidor que hospeda o recurso para o cliente que emitiu o pedido [Kalin 2009].

2.2.2 SOAP - *Simple Object Access Protocol*

SOAP é um protocolo para troca de informações estruturadas em uma plataforma descentralizada e distribuída. Dentre as suas características destacam-se o formato de mensagem baseado em XML (*eXtensible Markup Language*) e a utilização do protocolo HTTP para negociação e transmissão de mensagens. SOAP pode formar a camada base de uma pilha de protocolos de web services, fornecendo um framework de mensagens básico sob o qual os serviços web podem ser construídos. Este protocolo baseado em XML consiste de três partes: um envelope, que define o que está na mensagem e como processá-la, um conjunto de regras codificadas para expressar instâncias dos tipos de dados definidos na aplicação e uma convenção para representar chamadas de procedimentos e respostas [W3C 2012].

Um exemplo de cenário típico é apresentado na figura 2.1 onde a biblioteca do cliente SOAP solicita um serviço enviando uma mensagem SOAP ao servidor que por sua vez envia outra mensagem SOAP com a resposta do serviço correspondente.

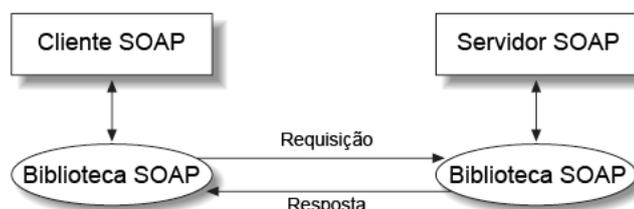


Figura 2.1: Arquitetura Web Service baseado em SOAP.

Em um serviço web baseado em SOAP, um cliente geralmente faz uma chamada de procedimento remoto ao servidor solicitando uma operação do serviço web. Essas operações requisição/resposta sobre mensagens SOAP padronizadas, permitem que o cliente e o servidor sejam escritos em diferentes linguagens de programação. No exemplo do código 2.1 um cliente Perl gera uma requisição HTTP que consiste em uma mensagem formatada cujo corpo é uma mensagem SOAP.

Código 2.1: Requisição HTTP com mensagem SOAP

```
1 POST http://127.0.0.1:9876/ts HTTP/ 1.1
```

```

2 Accept: text/xml
3 Accept: multipart/*
4 Accept: application/soap
5 User-Agent: SOAP::Lite/Perl/0.69
6 Content-Length: 434
7 Content-Type: text/xml; charset=utf-8
8 SOAPAction: ""
9
10 <?xml version="1.0" encoding="UTF-8"?>
11 <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
12     <S:Header/>
13     <S:Body>
14         <ns2:welcome xmlns:ns2="http://welcomesoap.teste.com/">
15             <name>Teste</name>
16         </ns2:welcome>
17     </S:Body>
18 </S:Envelope>

```

Como é possível visualizar no exemplo acima, o método POST é utilizado ao invés de uma requisição GET, pois, apenas um pedido POST tem um corpo, o que se torna necessário para encapsular a mensagem SOAP comumente chamada de envelope SOAP. No exemplo, o corpo SOAP contém um único elemento, correspondente ao método *welcome* requisitado como serviço.

No lado do servidor, há uma biblioteca responsável por processar a solicitação HTTP, extrair o envelope SOAP, determinar a operação solicitada e, em seguida, gerar a mensagem SOAP contendo o resultado da operação para que a mesma seja enviada de volta ao cliente.

Código 2.2: Resposta HTTP com mensagem SOAP

```

1 HTTP/1.1 200 OK
2 Content-Length: 323
3 Content-Type: text/xml; charset=utf-8
4 Client-Peer: 127.0.0.1:9876
5 Client-Response-Num: 1
6
7 <?xml version="1.0" encoding="UTF-8"?>
8 <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
9     <S:Body>
10         <ns2:welcomeResponse xmlns:ns2="http://welcomesoap.teste.com/
11             /">
12             <return>Hello Teste!</return>
13         </ns2:welcomeResponse>
14     </S:Body>

```

2.3 Comunicação assíncrona e Java Servlets

Antes da especificação HTTP 1.1, a comunicação web funcionava de forma síncrona. O servidor mantinha um número limitado de *sockets* a espera de possíveis requisições e, à medida que chegava uma nova requisição, era atribuído um *thread* para executar a tarefa requisitada. Neste modelo de sistema, o servidor detém um conjunto finito de *threads* disponíveis de forma que, uma vez que todas as *threads* estejam ocupadas, as novas requisições são colocadas em uma fila de espera até que uma *thread* termine a sua execução atual.

O processo de abrir e fechar uma conexão com *sockets* é uma operação custosa, influenciando bastante no tempo de espera e latência da comunicação, principalmente para o caso de múltiplas requisições. Na especificação HTTP 1.1 foi introduzido o modelo de conexão *keep-alive*, permitindo assim que um cliente mantenha a conexão aberta até que todos os recursos tenham sido recuperados [Chetty 2010]. Utilizando este novo conceito e visando desenvolver servidores web mais escaláveis, é comum por parte dos desenvolvedores a utilização de técnicas conhecidas como: *thread* por conexão e *thread* por requisição.

A técnica *thread* por conexão é baseada em conexões persistentes HTTP 1.1. Sob esta estratégia, cada conexão HTTP entre o cliente e o servidor é associada a uma *thread* do lado do servidor gerenciado por um *pool* de *threads*. Quando uma conexão é fechada, o recurso da *thread* é reciclado e está pronto para servir outras tarefas. Dependendo da configuração esta abordagem pode acarretar em um elevado número de conexões simultâneas resultando em um consumo de memória diretamente proporcional ao número de conexões HTTP. Já servidores configurados com um número fixo de *threads* pode sofrer de *starvation*, ou seja, novos pedidos do clientes pode demorar a serem executados ou até mesmo serem rejeitados uma vez que todas as *threads* no *pool* estejam sendo utilizadas.

Já a técnica de *thread* por requisição, consiste na utilização do recurso de entrada e saída não-bloqueantes introduzido na versão quatro da API Java. Desta forma, uma conexão HTTP persistente não exige que uma *thread* esteja constantemente associada à mesma. As *threads* podem ser alocadas para conexões somente quando os pedidos estão sendo processados. Quando uma conexão se torna ociosa, a *thread* pode ser reciclada, e a conexão colocada em um sistema de espera definido para detectar novas solicitações sem consumir uma *thread*. Este modelo, permite que os servidores web pos-

sem lidar com um número cada vez maior de conexões com um número fixo de *threads*. Tal técnica é implementada na maioria dos servidores web atuais, como por exemplo o Tomcat, Jetty, GlassFish (Grizzly), WebLogic e WebSphere [JavaWorld 2009].

2.3.1 Notificação do servidor (*Server push*)

O protocolo HTTP é um protocolo de requisição/resposta. Um cliente envia uma mensagem de requisição a um servidor, e o servidor responde com uma mensagem de resposta. O servidor não pode iniciar uma conexão com um cliente ou enviar uma mensagem inesperada para o mesmo. Este aspecto do protocolo HTTP aparentemente torna impossível o envio assíncrono de dados por parte do servidor, no entanto, várias técnicas foram criadas para contornar esta restrição. São elas:

- *Service streaming* - Permite que o servidor envie uma mensagem ao cliente quando ocorre um evento, sem um pedido explícito por parte do cliente. Na implementação real, o que ocorre é que, o cliente inicia uma conexão com o servidor através de um pedido, em seguida, a resposta é retornada em pedaços cada vez que um evento do lado do servidor ocorre. Os pedaços podem ser interpretados por JavaScript do lado do cliente. A resposta dura por um período indeterminado.
- *Long polling* - Também conhecida como *polling* assíncrono esta técnica funciona como um streaming onde um cliente se inscreve em um canal de conexão com o servidor enviando um pedido, este por sua vez mantém o pedido e aguarda um evento acontecer. Uma vez que o evento ocorre (ou depois de um tempo limite pré-definido), uma mensagem de resposta completa é enviada para o cliente. Após receber a resposta, o cliente envia imediatamente um novo pedido desta forma o servidor tem quase sempre um pedido pendente que pode ser usado para entregar os dados em resposta a um evento do lado do servidor.
- *Piggyback Passivo* - Quando o servidor tem uma atualização para enviar ao cliente, ele aguarda a próxima vez que o navegador faz uma solicitação e, em seguida, envia a sua atualização, juntamente com a resposta que o navegador estava esperando.

O serviço de *streaming* e *long polling* implementados com Ajax, são conhecidos como Comet, ou Ajax reverso. A ideia consiste no envio de dados do servidor para o navegador sem que o cliente solicite. Embora a lógica pareça simples, a implementação de aplicações Web no estilo Comet, tanto nos navegadores como em servidores, é algo que só se tornou viável nos últimos anos [JavaWorld 2008].

2.3.2 Servlet 3.0

As servlets consistem em classes Java que possuem a capacidade de gerar conteúdo HTML. O nome servlet vem da ideia de um pequeno servidor cujo objetivo é receber chamadas HTTP, processá-las e devolver uma resposta ao cliente [Oracle 2013c]. Desta forma, cada servlet seria responsável por ler dados da requisição do cliente e responder com outros dados (uma página HTML, uma imagem GIF e etc).

A implementação de servlets com suporte assíncrono nas versões anteriores a especificação 3.0, eram feitas utilizando-se de bibliotecas de terceiros que proviam suporte Comet . A partir da especificação servlet 3.0 é possível criar servlets que possam operar tanto em modo síncrono como assíncrono e que sejam portáteis a qualquer servidor de aplicações compatíveis (Tomcat 7 ou GlassFish 3.x, etc).

Em servlets síncronos, uma *thread* é utilizada para manipular a requisição HTTP do cliente se mantendo ocupada durante todo o processamento do pedido. Já as servlets assíncronas permitem que a *thread* fique livre para ser utilizada novamente tão logo a requisição seja entregue enquanto a tarefa requisitada será executada em algum outro segmento do lado do servidor. Uma vez que a tarefa foi concluída e os resultados estão prontos, a servlet aloca uma nova *thread* para lidar com estes resultados e envia-los de volta para o cliente.

A comunicação assíncrona a partir da especificação servlet 3.0 segue os seguintes procedimentos:

- O cliente efetua uma requisição HTTP a um determinado servlet.
- A requisição é então tratada por um dos métodos disponibilizados pela servlet (doGet, doPost, service, entre outros).
- O método cria um *AsyncContext* (chamando *startAsync*).
- O método deverá criar uma *thread* para tratar a requisição. Esta *thread* terá uma referência ao *AsyncContext* para que seja possível notificar o cliente quando os resultados estiverem disponíveis.
- O método requisitado deverá então terminar a sua execução enquanto o tratamento assíncrono é processado em segundo plano.

2.4 Soluções industriais

2.4.1 OPC

Derivada da tecnologia COM/DCOM da Microsoft, o OPC é um padrão aberto que visa unificar a comunicação permitindo a fácil integração de diversos sistemas, desde ins-

trumentos de campo até os sistemas de gerenciamento e de gestão corporativa permitindo assim a integração dos dados de toda a empresa, desde equipamentos de chão-de-fábrica aos sistemas ERP e setores corporativos [Hadlich 2006].

O OPC é um conjunto comum de interfaces, métodos e propriedades de comunicação que estabelece regras de uso dos componentes COM, agregados dentro de uma especificação padronizada e aberta para acesso público possibilitando o desenvolvimento de aplicações OPC a partir das especificações contidas no web site da *OPC Foundation* [Pudá 2008].

Inicialmente, o OPC surgiu como uma simples resposta aos *drivers* de comunicação proprietários para equipamentos de chão-de-fábrica e acabou por se tornar um padrão altamente difundido na indústria, possibilitando a criação de uma camada de comunicação padronizada que integra facilmente todas as informações industriais [Nogueira 2009]. Basicamente, o OPC estabelece as regras para que sejam desenvolvidos sistemas com interfaces padrões para comunicação dos dispositivos de campo (controladores, sensores, etc.) com sistemas de monitoração, supervisão e gerenciamento (SCADA, MES, ERP, etc.) [Fonseca 2002].

Dentre as especificações definidas pelo padrão OPC, as mais comuns são:

- **OPC Data Access Specification** - Definição da interface para leitura e escrita de variáveis de processo em tempo real;
- **OPC Alarms and Events Specification** - Definição da interface para monitoração de eventos;
- **OPC Historical Data Access Specification** - Definição da interface para utilização de dados de históricos;
- **OPC and XML Specification** - Integração entre OPC e XML para aplicação via web.
- **OPC Unified Architecture** - Novo conjunto de especificações que visa prover uma arquitetura multiplataforma livre da tecnologia COM.

A investida mais atual da *OPC Foundation*, a especificação OPC UA, é baseada em padrões como TCP/IP, HTTP, SOAP e XML marcando assim a transição do DCOM para uma arquitetura orientada a serviços (SOA). Tal modificação utiliza como base o WSDL (*Web Service Description Language*) que consiste em um padrão baseado em XML para descrever o serviço permitindo que estes sejam convertidos para COM ou outros protocolos de serviço web. Desta forma, o OPC UA estende grandiosamente as fronteiras do padrão OPC tornando-o acessível pela internet e independente de plataforma ou sistemas operacionais [Schleipen 2008]. Essa nova especificação busca cobrir toda gama de

aplicações suportada pelas especificação anteriores, conforme pode ser visto na figura 2.2.

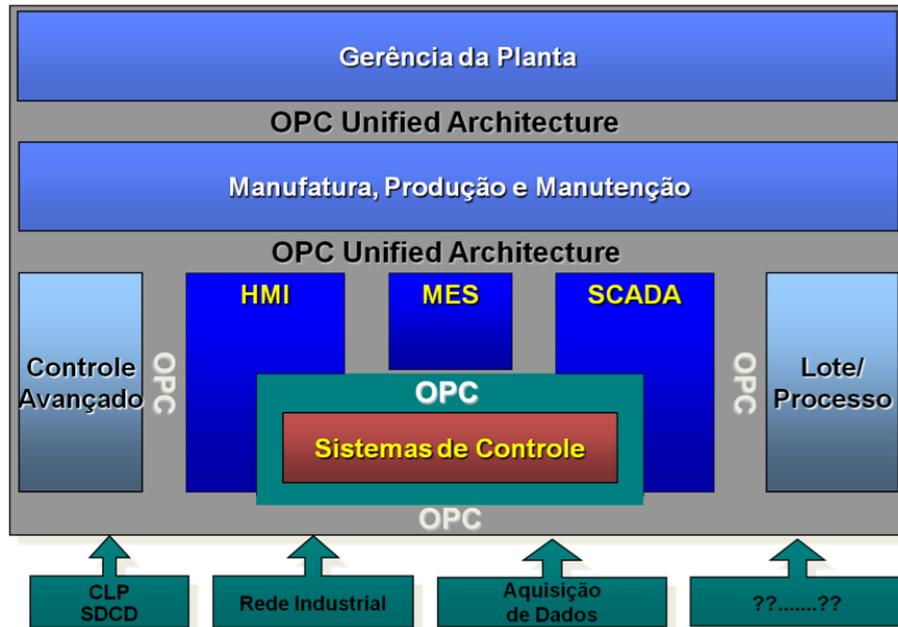


Figura 2.2: Aplicações OPC UA.

A especificação *OPC UA Specification: Part 6 – Mapping. Release Candidate 0.93* (2006) define atualmente dois tipos de mapeamentos: *UA Web Services* e *Native UA*. No primeiro caso, são utilizadas especificações WSDL bem como o protocolo SOAP para troca de informações. Já o segundo mapeamento utiliza apenas uma rede binária simples com mecanismos de segurança integrados.

A codificação dos dados pode ser feita tanto em XML como em binário. A codificação binária especifica a serialização de dados em uma seqüência de bytes o que torna a transmissão de dados mais rápida, uma vez que, o tamanho da mensagem é menor do que o obtido pela codificação XML. Por outro lado, a codificação XML permite a criação de clientes genéricos bastando assim interpretar os dados da mensagem SOAP.

O protocolo do mapeamento *Web Service UA* é o SOAP/HTTP(S), enquanto o mapeamento UA nativo normalmente é executado diretamente sobre TCP/IP.

2.4.2 CyberOPC

Baseado em tecnologias abertas, o CyberOPC é uma alternativa à especificação OPC XML-DA 1.0. O CyberOPC adota padrões abertos para codificação de dados e chamadas de processos industriais remotos provendo, assim, interoperabilidade a sistemas de supervisão e aquisição de dados (SCADA). O objetivo do projeto CyberOPC é definir um

padrão aberto para troca de dados industriais com base em tecnologias que visam minimizar o atraso de acesso no monitoramento contínuo de dados usando *streaming* de AJAX reverso [Torrise 2010]. Nesta abordagem, o envio de dados é totalmente assíncrono seja a partir do servidor para o cliente como do cliente para o servidor. Através deste paradigma, o CyberOPC promete minimizar a latência na entrega de dados para o cliente diminuindo o número de requisições enviadas ao servidor.

O CyberOPC, assim como o OPC-XML e OPC-UA, utiliza-se do HTTP como protocolo de transporte, no entanto, essas tecnologias diferem entre si na forma de encapsulamento do comando. Ao passo que, as especificações OPC utilizam serviços web baseado em SOAP, o que significa que, o comando relacionado ao serviço é encapsulado no cabeçalho do envelope HTTP, o CyberOPC adota o protocolo JSON-RPC para troca de mensagens sobre o corpo do envelope HTTP [Torrise 2011], conforme demonstra o código 2.3.

Tanto o XML como o JSON são formas eficientes e independente de linguagem para troca de mensagens de texto. Uma vez que, o JSON utiliza apenas dois tipos de estrutura de dados: uma coleção de pares nome/valor e uma lista ordenada de valores, se torna fácil a conversão de dados de/para o JSON uma vez que, quase todas as linguagem de programação dispõe de estruturas semelhantes.

Código 2.3: Requisição de leitura CyberOPC via JSON

```
1 POST /EEEHOME HTTP/1.1
2 User-Agent: CyberOPC Client Library
3 Host: EEEHOME
4 Accept: application/json
5 Connection: Keep-Alive
6
7 { "method": "Read", "params": { ItemList: ["Simulator.devsim1.SetPoint
   " ] }, "id": 3 }
```

MIDDLEWARE - ARQUITETURA PROPOSTA

3.1 Motivação

A chamada de recursos remotos através de um serviço web permite que aplicações clientes chamem determinadas funcionalidades encontradas remotamente, compartilhando dados e efetuando processamento em outros computadores. Apesar da programação e desenvolvimento de servidores web terem se tornado cada vez mais fácil, ela põe o desenvolvedor frente a um novo paradigma, a computação de objetos distribuídos.

Sistemas distribuídos podem ser construídos diretamente em protocolos de baixo nível. Por exemplo, a comunicação pode ser baseada em *sockets* TCP/IP permitindo assim que as aplicações possam enviar mensagens entre si. Tal abordagem, no entanto, faz com que os desenvolvedores tenham que lidar com comunicação de baixo nível, de forma que, tais sistemas em geral apresentem problemas como:

- Mais susceptível a erros da parte do desenvolvedor;
- Dificuldade em manter e aplicar mudanças ao sistema;
- Não possuem transparência na comunicação distribuída.

Desta forma, é necessário que o desenvolvedor se preocupe, não só com a criação dos serviços, mas também em definir um protocolo comum para comunicação entre o cliente e o servidor, além de codificar os dados que serão trafegados.

Uma solução para este problema consiste em introduzir uma camada adicional conhecida como *middleware* de comunicação, ou simplesmente *middleware*, que abstrai a heterogeneidade das plataformas utilizadas e provê transparência na comunicação para

os desenvolvedores. Tal transparência consiste em tornar as chamadas remotas o mais parecido possível a chamadas locais.

A figura 3.1 mostra como o *middleware* esconde os serviços de rede e outros detalhes de comunicação remota da aplicação cliente.

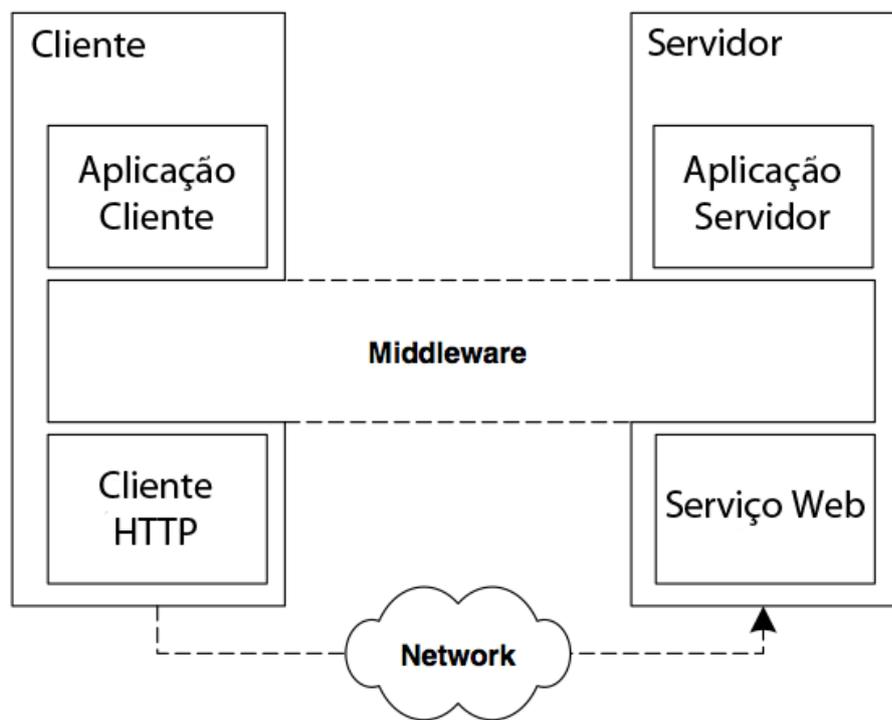


Figura 3.1: Camada de comunicação.

O *middleware* é uma camada de software adicional que fica entre a comunicação de rede oferecida pelo SO e a camada de aplicação, que hospeda o cliente e servidor de aplicações. Devido à camada adicional, o *middleware* esconde a heterogeneidade entre as plataformas onde se encontram a aplicação cliente e o servidor. Seja qual for a forma de comunicação utilizada, o desenvolvedor deve ser protegido dos detalhes de como a respectiva implementação do *middleware* realmente se comunica através da rede. Por exemplo, uma implementação do *middleware* pode cuidar de manipulação de conexão, transmissão de mensagens e serialização, mas não tenta esconder as condições de erro específicas introduzidas pelo fato de que o servidor está localizado remotamente.

Como resultado deste trabalho, foi desenvolvido o *middleware* JETTY, que visa prover aos desenvolvedores uma forma fácil e transparente de comunicação com web services, através da API Servlet 3.0 da plataforma Java. Tal proposta se destaca das demais opções presentes no mercado por permitir a comunicação assíncrona entre o servidor e a aplicação cliente, característica esta que se mostra de extrema importância para apli-

cações na área de automação industrial, onde o servidor precisa enviar periodicamente informações ao cliente, como por exemplo, dados de alarmes e variáveis de processo.

3.2 Proposta

A arquitetura proposta baseia-se no importante princípio de que: a definição do comportamento e a implementação do comportamento são conceitos separados. As linguagens de programação orientadas a objetos permitem que o código que define o comportamento e o código que o implementa permanecerem separados, isto é feito através do conceito de interface. Desta forma, ao utilizar o *middleware* JETTY, a definição dos serviços que serão providos remotamente é codificada usando uma interface comum ao cliente e servidor. Tal interface define como um ou mais métodos devem ser chamados sem especificar detalhes de como os mesmos funcionam internamente.

A implementação do serviço remoto é codificada em uma classe do lado do servidor, que passa a ser conhecida como objeto remoto. Logo, as interfaces definem o comportamento e as classes definem a implementação. A classe que implementa o comportamento roda do lado do servidor, ao passo que, a classe que roda no cliente atua como um *proxy* para o serviço remoto conforme será explicado com mais detalhes futuramente.

3.2.1 Visão geral da arquitetura

O *middleware* desenvolvido oferece toda uma infra-estrutura para que o cliente possa se comunicar com objetos remotos. O cliente invoca uma operação a um objeto local e espera que ela seja encaminhada ao objeto remoto. Para que isso aconteça, existem vários blocos da implementação trabalhando de forma sequencial.

A figura 3.2 demonstra esta interação entre os blocos que compõem o *middleware* bem como suas interações entre si.

Inicialmente, existe a figura do **Proxy do cliente** que consiste em um objeto local que oferece a mesma interface (validada pelo **Descritor de interface**) do objeto remoto. Todas as chamadas de métodos ao *proxy* do cliente são capturadas e repassadas ao solicitante.

O bloco **Solicitante** é responsável por construir uma chamada a partir dos dados enviados pelo cliente (parâmetros como a localização do objeto remoto, tipo do objeto, operação e argumentos). No lado do cliente, o solicitante transfere os dados coletados para o **Empacotador**, que tem por função serializar os dados de acordo com o estilo de comunicação JSON-RPC. Em seguida, o solicitante aciona o **Handler de requisição** do cliente para tratar a comunicação de rede. No lado do servidor, as chamadas remotas

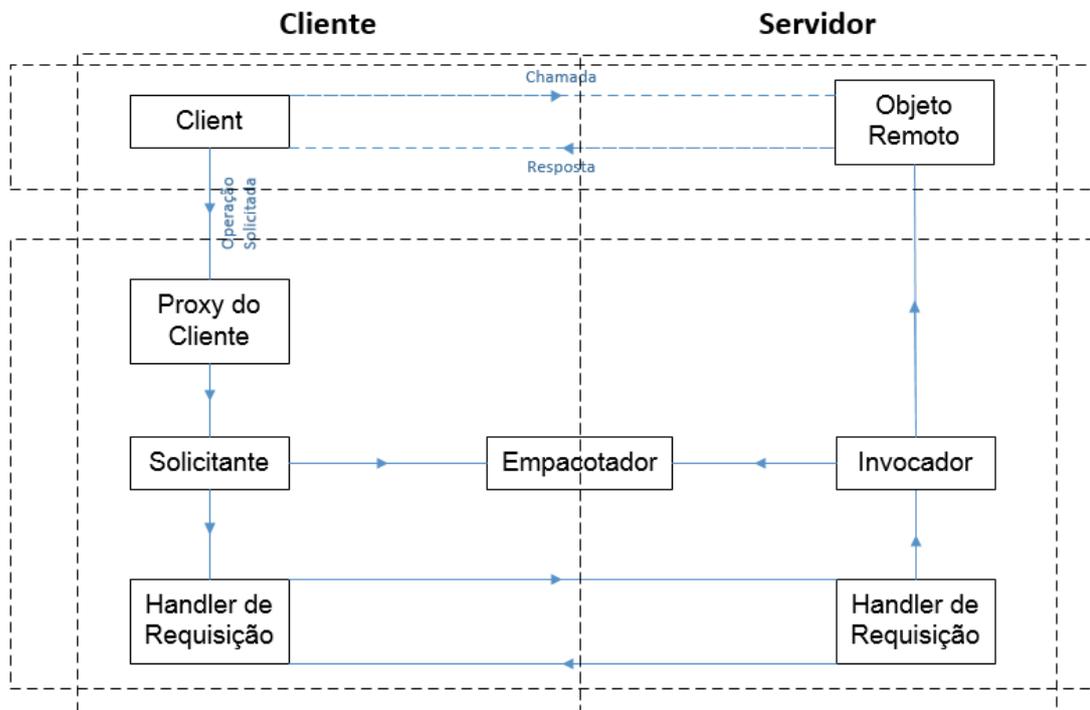


Figura 3.2: Arquitetura geral.

recebidas são tratadas pelo **Handler de requisição** do servidor, que recebe a mensagem e encaminha ao invocador.

O bloco que representa o **Invocador** tem por função decodificar a mensagem recebida, através do empacotador, e encaminha-la ao objeto remoto correspondente.

Caso ocorra alguma falha de comunicação, tanto por parte do cliente quanto do servidor, um erro é sinalizado.

Esses blocos serão detalhados nas sessões subsequentes.

3.2.2 Objetos Remotos

Em linhas gerais, uma aplicação servidora utilizando JETY cria um ou mais objetos remotos, torna sua referência acessível através de um identificador único e espera por chamadas do cliente. A aplicação cliente solicita a referência a um determinado objeto remoto presente no servidor, invoca um de seus métodos e aguarda uma resposta. O JETY se encarrega de prover todo o mecanismo para comunicação entre o cliente e o servidor de forma transparente.

A classe *JsonRpcServer* possibilita que o servidor adicione um ou mais objetos remotos através do método *addHandler* (descrito no código 3.1).

Código 3.1: Método addHandler

```
1 /**
2  * Add a new handler class to treat the client request.
3  * @param <T> the handler class type
4  * @param name a name to identify the handler
5  * @param handler an instance of the handle
6  * @param classes the interfaces implemented by the handler
7  * @throws RuntimeException erro while creating the handler
8  */
9 public <T> void addHandler(String name, T handler, Class<T>...
    classes)
```

O exemplo do código 3.2 demonstra esta facilidade, onde uma certa aplicação servidora adiciona um objeto *Calculator*, passando como parâmetros o identificador do objeto (calc), a implementação do objeto remoto (calcImpl) e por fim, a interface que representa o objeto (Calculator.class).

Código 3.2: Adicionando um objeto remoto

```
1 JsonRpcServer executor = new JsonRpcServer();
2
3 Calculator calcImpl = new SimpleCalculatorImpl();
4 executor.addHandler("calc", calcImpl, Calculator.class);
```

3.2.3 Estilo de comunicação remota - JSON-RPC

O estilo de comunicação remota define o formato das mensagens que serão enviadas entre o cliente e o servidor. No desenvolvimento deste trabalho, foi adotado o padrão RPC (*Remote Procedure Calls*) que visa deixar a requisição de um objeto remoto o mais parecido possível a uma chamada local.

No padrão RPC, o servidor fornece um conjunto bem definido de operações que o cliente pode invocar. Quando um cliente chama uma determinada operação do servidor, o processamento do cliente fica suspenso até o momento que o servidor processa a requisição e retorna os resultados. Para o desenvolvedor da aplicação cliente, essas operações são vistas quase como operações locais: normalmente contendo o nome da operação, os parâmetros necessários para execução da mesma, um tipo de retorno, e uma maneira de sinalizar erros. Em sistemas desenvolvidos a partir de linguagens de programação orientadas a objetos, a aplicação servidora fornece um conjunto de objetos remotos que disponibilizam operações para o cliente a partir de uma interface publica.

Nesse modelo, existe ainda a noção de identidade de objeto que possibilita o cliente indexar a requisição a um objeto remoto específico.

No desenvolvimento do JETY foi utilizada a especificação JSON-RPC que consiste em um protocolo simples que define alguns poucos tipos de dados e comandos para troca de mensagens codificadas em JSON [JSON-RCP 2013]. Em contraste ao XML-RPC ou ao SOAP, ele permite a comunicação bidirecional entre o serviço e o cliente, onde cada uma das pontas pode chamar a outra ou enviar notificações.

Uma requisição no estilo JSON-RPC consiste em uma chamada a um método provido por um sistema remoto. Ela deverá conter as seguintes propriedades:

- *method* - Uma *string* contendo o nome do método a ser chamado.
- *params* - Um *array* de objetos a ser passado como parâmetro ao método em questão.
- *id* - Um valor a ser utilizado para conciliar a resposta com a solicitação em questão.

A aplicação servidora deverá responder a solicitante com os seguintes parâmetros:

- *result* - O dado retornado pelo método chamado. Caso ocorra um erro o valor deverá ser null.
- *error* - Caso tenha ocorrido um erro, um código referente ao mesmo deverá ser retornado. Null caso contrário.
- *id* - O id da requisição ao qual a resposta pertence.

O código 3.3 demonstra um exemplo de requisição e resposta no padrão JSON-RPC 2.0.

Código 3.3: Exemplo de requisição e resposta em JSON-RPC 2.0

```
1 --> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
2 <-- {"jsonrpc": "2.0", "result": 19, "id": 1}
```

A escolha do JSON-RPC permite ainda a fácil criação de aplicações cliente por parte dos desenvolvedores que não desejam utilizar o *middleware* do lado do cliente, uma vez que, a mensagem trafegada consiste em um texto de fácil compreensão, além que que, existem diversas bibliotecas que possibilitam a conversão de classes para uma *string* codificada em JSON.

3.2.4 Módulos

Como dito anteriormente, o *middleware* aqui desenvolvido foi construído a partir de blocos básico de objetos distribuídos. O padrão adotado por este projeto é largamente utilizado

em soluções de *middleware*, conforme descrito em Völter et al. (2005) e visa não só padronizar a sua estrutura como prover uma arquitetura modular que permita a fácil modificação e manutenção da mesma. São eles:

Descritor de interface

Este módulo é responsável por validar as interfaces cadastradas tanto pelo cliente quanto pelo servidor, a fim de que, ao se efetuar uma requisição, o objeto remoto apropriado seja localizado e seu método devidamente executado.

O descritor de interfaces atua como um contrato entre a interface conhecida pelo cliente e a implementação do objeto remoto, fornecida pelo servidor. Seu fluxo de execução efetua uma checagem dos métodos e parâmetros de métodos (de forma recursiva, caso tais parâmetros sejam classes ou estruturas de dados), a fim de verificar se os mesmos são serializáveis pela biblioteca GSON.

O diagrama de sequência da figura 3.3 apresenta o processo de validação das interfaces requisitadas por uma aplicação cliente. Ao requisitar um acesso a um objeto remoto, o cliente invoca o método *getClassFromServer* da classe *JsonRpcClient* passando como parâmetro uma interface qualquer (tipo T) que será validada e futuramente retornada para o mesmo em forma de *proxy*.

Um processo semelhante ocorre do lado do servidor, quando o mesmo adiciona um determinado objeto remoto. Caso a interface requisitada não seja compatível com os critérios definidos pelo descritor, uma exceção do tipo *IllegalArgumentException* será lançada.

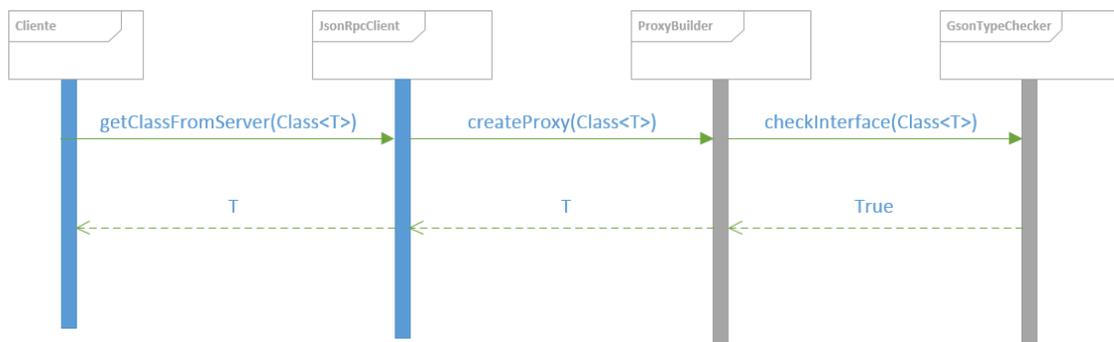


Figura 3.3: Sequencia de validação de interface.

As classes abaixo representam o módulo do descritor de interfaces.

- *AbstractTypeChecker* – Uma interface que provê os métodos necessários para checar os tipos de dados.

- *GsonTypeChecker* – Implementação da interface *AbstractTypeChecker* que valida os tipos suportados pela biblioteca Gson.

Solicitante

O desenvolvimento de sistemas distribuídos apresenta um novo desafio aos programadores, dentre eles, podemos citar a integração de componentes heterogêneos em aplicações coerentes bem como o uso eficiente dos recursos de rede. A comunicação em rede é mais complexa do que a comunicação local uma vez que se faz necessário estabelecer conexões, tratar requisições e dados a serem transmitidos e lidar com um novo conjunto de possíveis erros inerentes a comunicação remota.

Invocar objetos remotos disponíveis em um serviço web requer que os parâmetros necessários para operação sejam coletados e empacotados em uma mensagem HTTP. Essas tarefas devem ser realizadas para cada objeto remoto a ser acessado pelo cliente, portanto, pode tornar-se tedioso para os desenvolvedores. O cliente tem que realizar várias operações recorrentes para cada requisição, dentre elas, o empacotamento da informação, o gerenciamento da conexão de rede, a transmissão da requisição, a manipulação do resultado da requisição e o tratamento de erros.

O solicitante recebe os parâmetros passados pelo cliente e delega a tarefa de empacotamento dos dados de acordo com o estilo de comunicação utilizado entre o cliente e servidor. Em seguida, ele se encarrega de enviar o pedido ao *handler* de requisição do cliente, receber a resposta e lançar exceções caso necessário.

O bloco solicitante é composto pela classe *InvokeHandler* e seu fluxo básico apresentado na figura 3.4. Ao receber o método a ser executado, bem como seus parâmetros, os dados são encaminhados as classes do empacotador para que seja criada a requisição JSON-RPC. Em seguida, é efetuada uma verificação a fim de detectar se o método deverá ser executado de forma síncrona ou assíncrona.

Caso o tipo de requisição seja assíncrona, é criado um proxy para que o servidor possa acessar o cliente, bem como um *listener* no lado do cliente, para que este escute as chamadas do servidor. Os detalhes da comunicação assíncrona serão discutidos mais a frente.

Por fim, o *InvokeHandler* recebe a resposta do servidor, encaminhada a ele pelo *handler* de requisição, e verifica se a mesma contém a resposta esperada pelo *proxy* ou uma mensagem de erro.

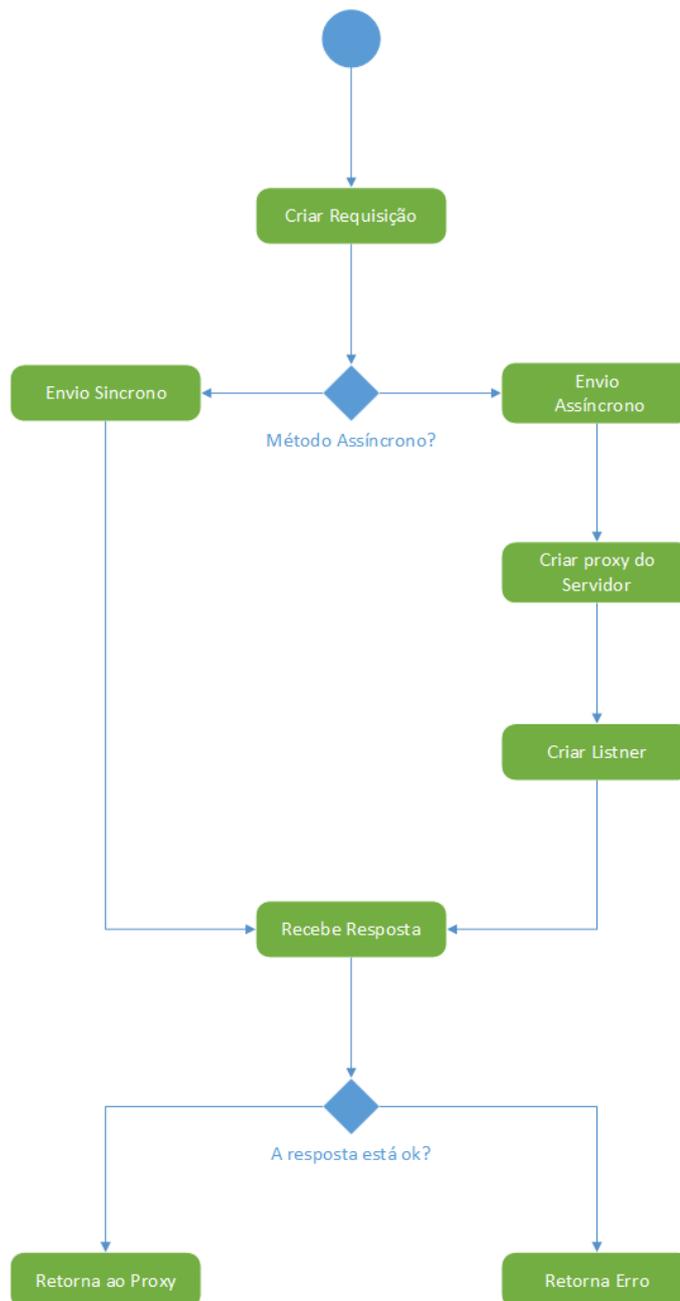


Figura 3.4: Fluxo do solicitante.

Proxy do cliente

O principal objetivo do *middleware* de objetos distribuídos é a facilidade de desenvolvimento de aplicações distribuídas. Os desenvolvedores não devem ser forçados a desistir de seu modo habitual de programação. No caso ideal, eles simplesmente invocam operações em objetos remotos como se fossem objetos locais.

Neste contexto, a implementação do *middleware* disponibiliza um *proxy* para acessar o objeto remoto. O *proxy* é fornecido como um objeto que espelha o objeto remoto, o nome das operações e seus argumentos. Ele intercepta uma requisição a um método local e encaminha a chamada ao solicitante que, por sua vez, constrói uma requisição remota a partir dos parâmetros e envia a chamada para o objeto remoto.

O *proxy* consiste em um objeto criado dinamicamente, em tempo de execução e específico para cada objeto remoto gerado a partir da descrição da interface do mesmo [Buschmann et al. 1996]. Para solicitar um serviço remoto, o cliente interage apenas com o *proxy* local. No momento em que o cliente chama um método disponível na interface, o *proxy* do cliente traduz o método local e seus parâmetros e desencadeia a requisição. Não obstante, o *proxy* do cliente recebe o resultado do serviço solicitado e entrega ao cliente usando um simples retorno para o método (vide figura 3.5).

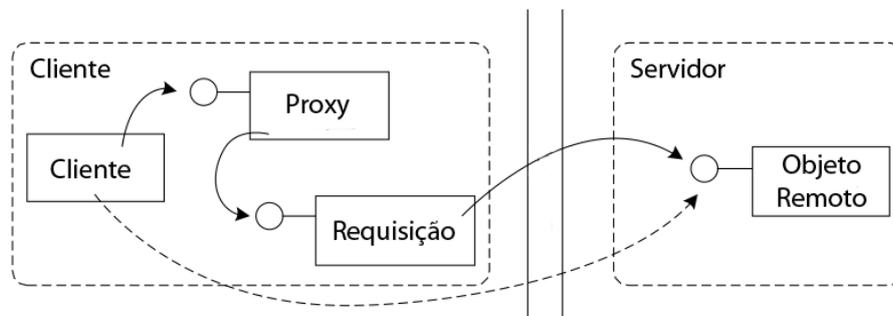


Figura 3.5: Proxy do cliente.

Como o *proxy* do cliente é específico para cada objeto remoto, ele é tipicamente gerado a partir da interface da classe que implementa o objeto em si, de forma que, tanto o servidor como o cliente deverão conhecer esta interface.

O cliente efetua a solicitação de um *proxy* a partir do método *getClassFromServer* da classe *JsonRpcClient*. Para isto, o cliente deverá passar como parâmetro um objeto *JsonRpcClientTransport* para acesso ao servidor, o nome que identifica o objeto remoto do lado do servidor e as interfaces que descrevem o objeto a ser acessado. O código 3.4 apresenta o cabeçalho, bem como a documentação, do método *getClassFromServer*.

Código 3.4: Método *getClassFromServer*

```

1 /**
2  * Get a proxy object to access the server object.
3  * @param <T> the object class type
4  * @param transport the transport to access the server
5  * @param handler the name of the server handle object
6  * @param classes the interfaces implemented by the object

```

```
7 * @return a proxy instance to the remote object
8 * @throws IllegalArgumentException erro while creating the proxy
9 */
10 public <T> T getClassFromServer(final JsonRpcClientTransport
    transport, final String handle, final Class<T>... classes)
    throws IllegalArgumentException
```

Empacotador

Para se executar chamadas remotas, as informações trafegadas necessitam ser transportadas pela rede através de um fluxo de bytes. É neste contexto que se aplica a figura do empacotador que fica responsável por transformar, tanto as mensagens de requisição e resposta, quanto os tipos de dados que serão utilizados como parâmetros e retornos dos métodos, em uma codificação que possa ser enviada pela rede.

Estruturas de dados e tipo definidos pelo usuário, normalmente tem referências a outros tipos, formando uma complexa hierarquia de objetos. No desenvolvimento do *middleware* proposto foi utilizado a biblioteca Gson, desenvolvida por um grupo do Google, a fim de possibilitar a conversão de objetos Java em uma *string* codificada em JSON e vice versa, provendo assim, uma enorme flexibilidade aos desenvolvedores no que diz respeito aos tipos de dados suportados [Google 2013].

Ao interagir com o *proxy*, o cliente efetua uma chamada a um método de forma semelhante a apresentada no exemplo do código 3.5. Tal solicitação, terá então sua requisição, bem como sua resposta, encapsulada no estilo JSON-RPC, conforme o código 3.6.

Código 3.5: Exemplo de chamada de método

```
1 Double res = calc.sum(2.0, 3.0);
```

Código 3.6: Requisição e resposta no estilo JSON-RPC

```
1 --> "{\"jsonrpc\":\"2.0\",\"id\":1279562108,\"method\":\"calc.sum\",\"params\":[2.0,3.0]}\"
2 <--  "{\"jsonrpc\":\"2.0\",\"id\":1279562108,\"result\":5.0}\"
```

Caso ocorra alguma exceção do lado do servidor, a informação do erro também será trafegada pela rede. Neste caso, o código e mensagem do erro será encapsulado no padrão JSON-RPC pelo empacotador, conforme apresentado no exemplo do código 3.7.

Código 3.7: Erro no estilo JSON-RPC

```
1 "{\"jsonrpc\":\"2.0\",\"error\":{\"id\":1279562108,\"code\":-32600,\"message\":\"Invalid method name\"}}\"
```

A figura do empacotador é composta pelas seguintes classes:

- *JsonConverter* – Serializa e desserializa objetos java em sua codificação JSON.
- *JsonRpcMethod* – Representa uma mensagem de requisição/resposta codificada no estilo JSON-RPC.

Handler de requisição do cliente

Para se enviar uma solicitação ao servidor de aplicações varias tarefas deverão ser executadas, tais como, estabelecer e configurar conexões, manipulação do tempo de espera, manipulação dos resultados, tratamentos de erros, entre outros. O solicitante tem a função de construir a requisição a ser enviada, no entanto, fica a cargo do *handler* de requisição a gestão de conexões, envio de dados e recuperação dos resultados. Além disso, o *handler* do cliente lida com o período de espera, segmentação e erros de invocação.

O cliente interage com o *handler* de requisição criando uma instância da classe *JsonRpcClientTransport*, passando como parâmetro a URL de conexão com o servidor. Após isto, o mesmo deverá ser passado como parâmetro para o método *getClassFromServer* da classe *JsonRpcClient* (vide código 3.8).

Desta feita, as demais utilizações do *handler* de requisição serão efetuadas internamente pelo *middleware*.

Código 3.8: Transporte do cliente

```
1 String url = "http://localhost:8080/Calculator/CalcServer";
2
3 JsonRpcClientTransport transport;
4 transport = new JsonRpcClientTransport(new URL(url));
5
6 JsonRpcClient invoker = new JsonRpcClient();
7 Calculator calc = invoker.getClassFromServer(transport, "calc",
    Calculator.class);
```

Para processar uma requisição síncrona, o fluxo de execução segue conforme o diagrama de fluxo da figura 3.6.

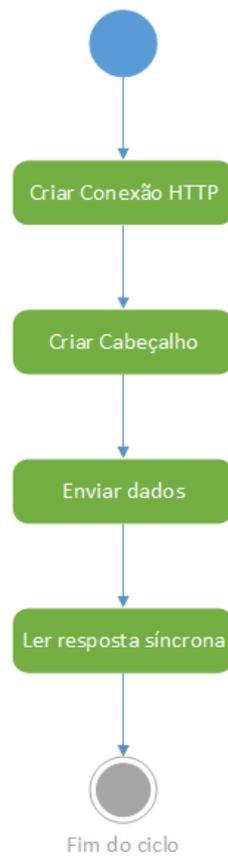


Figura 3.6: Fluxo de requisição síncrona.

Inicialmente, o *handler* abre uma conexão HTTP com a URL passada como parâmetro e, em seguida, prepara o cabeçalho da requisição HTTP de acordo com os parâmetros da tabela 3.1.

URI de Solicitação	/Calculator/CalcServer
Método	POST
Protocolo	HTTP/1.1
Endereço IP do cliente	127.0.0.1
accept-encoding	gzip
user-agent	Java/1.7.0_11
host	localhost:8080
accept	text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
connection	keep-alive
content-type	application/x-www-form-urlencoded
content-length	72

Tabela 3.1: Parâmetros do cabeçalho HTTP.

Feito isto, a conexão HTTP é então estabelecida e iniciado o processo de envio da informação através de um fluxo contínuo de bytes.

Uma vez enviado os dados da solicitação é então iniciado o processo de leitura da resposta. Novamente, os dados são recebidos através de um fluxo de bytes.

Para se realizar uma requisição assíncrona, a instância da classe *JsonRpcClientTransport* é internamente substituída pela classe *AsynchronousTransport*. Além de executar toda a sequência para envio e recebimento de uma resposta síncrona (conforme apresentado anteriormente), ela se encarrega de criar uma *thread* para escuta de novas requisições assíncrona, além de gerenciar o ciclo de vida da mesma caso o cliente ou servidor fechem o canal de comunicação (vice figura 3.7).

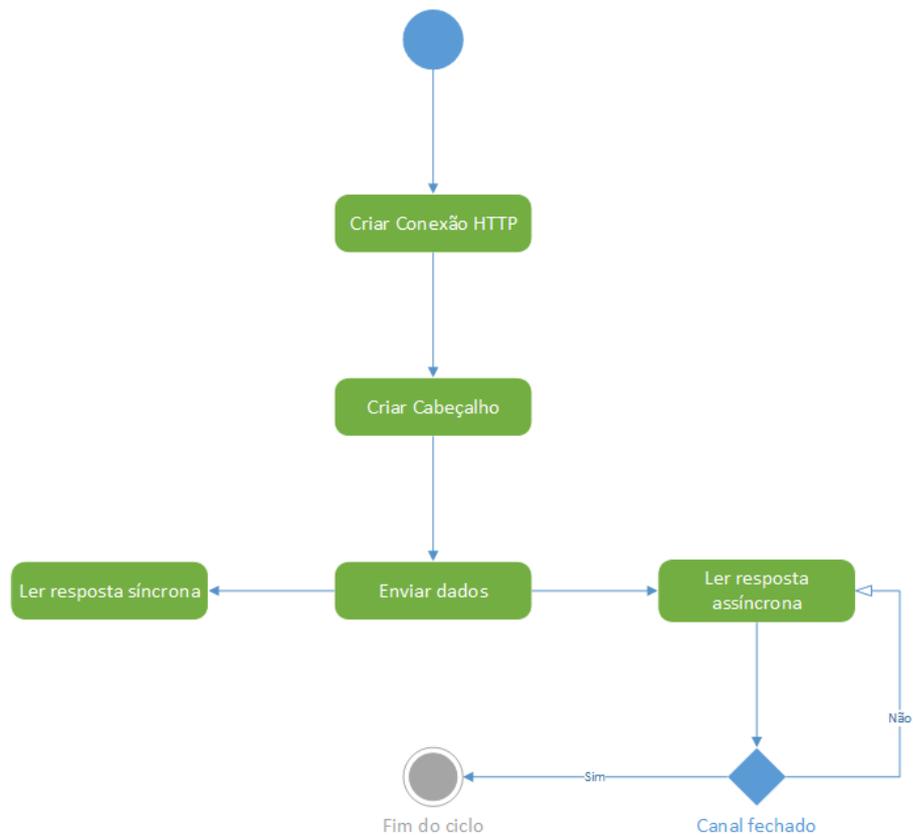


Figura 3.7: Fluxo de requisição síncrona.

Invocador

Quando um cliente envia os dados da requisição pela rede, o objeto remoto alvo tem de ser alcançado de alguma forma. O cliente tem por função apenas fornecer as informações necessárias para selecionar o objeto remoto correspondente. A aplicação servidora deverá lidar com essa chamada e encaminhar a chamada do método ao objeto remoto sem que este tenha que conhecer ou até mesmo implementar qualquer detalhes da comunicação efetuada.

O invocador é o bloco responsável por esta tarefa. Ele recebe uma chamada oriunda do *handler* de requisição e separa devidamente as informações da requisição em identificação do objeto, nome da operação e parâmetros da operação executando o processo de desserialização necessário.

O diagrama de sequência da figura 3.8 apresenta os passos executados para que uma chamada chegue ao objeto remoto de destino. Ao receber uma requisição do *handler* de requisição do cliente, a classe *JsonRpcServer* inicia a invocação solicitando o método a ser executado ao *HandlerManager* passando como parâmetro um *JsonRpcMethod*, o qual

consiste na representação em JSON-RPC da solicitação. O *HandlerManager* localiza o objeto remoto, através do identificador presente na chamada JSON-RPC, e obtém a referência ao objeto remoto (encapsulado em um objeto *HandleEntry*, que representa a associação entre os objetos remotos e suas interfaces).

De posse do *HandleEntry*, o *HandlerManager* requisita o método a ser executado. Isto é feito através da chamada ao método *getExecutableMethod* que se encarrega de varrer a interface do objeto remoto e procurar o método a ser invocado. Neste nível, também é efetuada uma pré-análise dos parâmetros dos métodos, a fim de identificar possíveis métodos sobrescritos, ou seja, métodos que possuem o mesmo nome porém diferentes tipos de parâmetros.

Antes de executar o método recém obtido, é necessário converter os parâmetros recebidos na requisição JSON-RPC para os seus respectivos tipos de dados. O *JsonRpcServer* realiza esta tarefa através do método *getTypedParams* da classe *JsonRpcMethod*.

Por fim, o método desejado pode ser finalmente chamado através do método *invoke*, que recebe como parâmetros o objeto remoto e o array de parâmetros a serem utilizados pelo método solicitado.

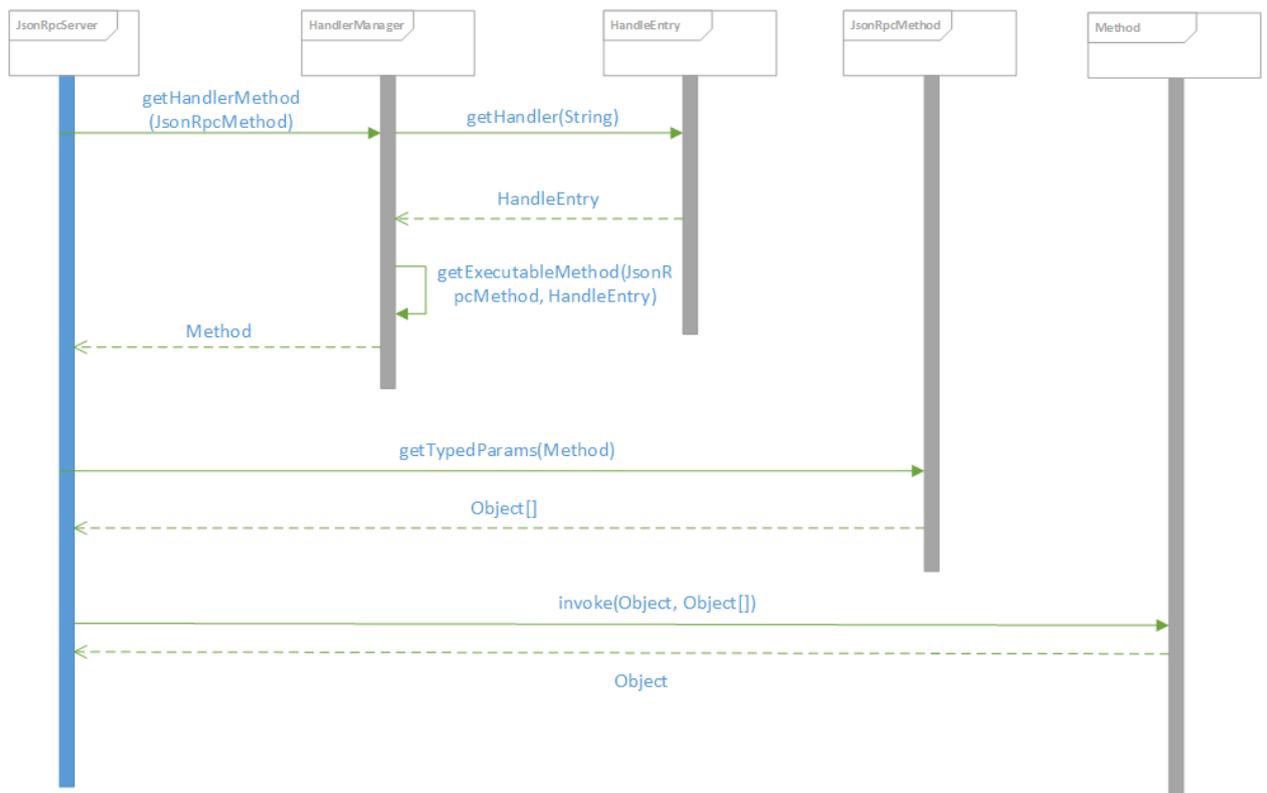


Figura 3.8: Fluxo de invocação.

Após sua execução, o retorno do método é então encapsulado em uma resposta

JSON-RPC e retornado ao cliente.

Erros

Embora a solução de *middleware* torne transparente para o usuário os aspectos de comunicação e acesso remoto, uma chamada de método por parte do cliente pode nunca atingir esse objetivo, seja devido a insegurança inerente a rede de comunicação, falhas de rede, falhas de servidores, ou até mesmo falha ao acessar um objeto remoto. Os clientes precisam ser notificados a fim de que possa lidar com a ocorrência dos mesmo.

No *middleware* desenvolvido, as classes listadas abaixo são utilizadas para representar erros.

- *AsynchronousException* – Representa um erro ocorrido ao tentar enviar uma resposta assincronamente.
- *JsonRpcException* – Erro ao codificar ou interpretar uma mensagem no estilo JSON-RPC.
- *JsonRpcClientException* – Representa um erro ocorrido quando o cliente realiza uma tentativa de enviar um solicitação ao servidor, ou obter a resposta do mesmo.
- *JsonRpcRemoteException* – Representa erros ocorridos por falha de rede.
- *JsonRpcErrorBuilder* – Constrói a representação do erro no padrão JSON-RPC.

Comunicação assíncrona e Proxy do Servidor

Em contraste com as chamadas síncronas, as chamadas assíncronas desassocia o cliente da aplicação servidora, de forma que, o cliente não fica ocioso esperando uma resposta. Quando um resultado torna-se disponível para o solicitante, o cliente deverá ser informado imediatamente, de modo que ele possa trabalhar os resultados obtidos.

Desta forma, é necessário que haja uma forma de notificar o cliente quando uma resposta do servidor esteja disponível. É nesse contexto que surge a figura do *callback*. Um *callback* consiste em uma interface cuja aplicação cliente conheça a implementação. Ela possui métodos que deverão ser chamados do lado do cliente quando uma resposta estiver disponível.

Na chamada do método assíncrono, o cliente passa um objeto *callback* ao servidor. Como a implementação do mesmo é conhecida apenas por parte do cliente, a aplicação servidora utiliza-se da interface *callback* (comum as duas aplicações) para criar um *proxy* (de forma similar ao que ocorre no *proxy* do cliente anteriormente detalhado). O *proxy* do servidor contém uma referência a comunicação com o objeto real do lado do cliente, possibilitando assim, a chamada de métodos do lado do cliente. Quando o resultado

da solicitação se encontra disponível, o middleware de comunicação se encarrega de encaminhar o retorno ao objeto *callback*. Tal cenário é apresentado na figura 3.9

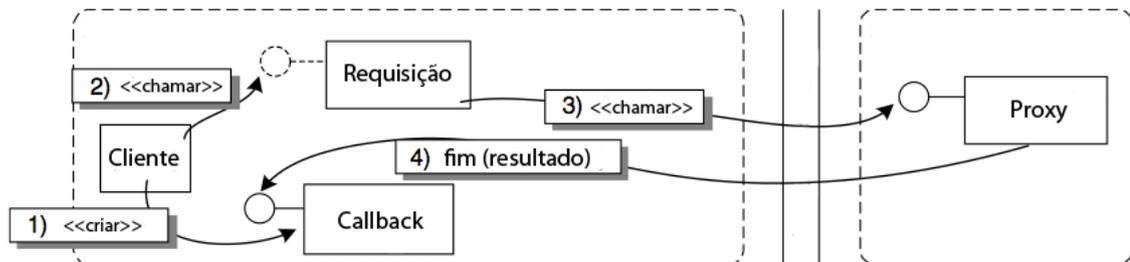


Figura 3.9: Callback para comunicação assíncrona.

O *middleware* JETTY fornece a anotação *AsynchronousMethod* que deverá ser utilizada para informar que um determinado método do objeto remoto do servidor enviará respostas assíncronas. Para isso, é necessário fornecer uma interface (com anotação *Callback*), que será utilizada para chamadas remotas no lado do cliente. O cliente deverá então se encarregar de criar uma classe contendo um ou mais métodos os quais poderão ser chamados pelo servidor. Ao invocar um método assíncrono, o cliente passa a instância do objeto *callback* como parâmetro do método. A requisição é então encaminhada ao servidor que fica responsável por criar o *proxy* que atuará como elo de comunicação com o cliente e será utilizado para enviar eventuais respostas.

O *middleware* torna transparente, tanto ao desenvolvedor do cliente como do servidor, toda a complexa lógica envolvida na transição assíncrona. A figura 3.10 apresenta o fluxo executado na camada cliente.

Quando a aplicação cliente realiza uma requisição assíncrona, o solicitante realiza a verificação da anotação *AsynchronousMethod* e em seguida efetua a troca do *JsonRpcClientTransport* por um *AsynchronousTransport*.

De posse do novo transportador, a solicitação é enviada ao servidor, afim de obter a resposta síncrona correspondente ao retorno do método. Caso o servidor encontre um erro na requisição, o mesmo é retornado e enviado ao cliente.

Uma vez que a requisição retornou com sucesso, é criado um *handler* que armazena o objeto *callback* a ser utilizado para receber as requisições assíncronas. O método *addHandler* efetua esta função checando os parâmetros passados na requisição. Os objetos com anotação *Callback* são então identificados e armazenados para uso posterior em um *HandlerManager*. Caso não haja nenhum objeto *callback* passado como parâmetros, uma exceção do tipo *AsynchronousException* é retornada para o cliente.

Por fim, é criado um *listener* que utiliza-se do *AsynchronousTransport* que contém uma *thread* para leitura periódica de requisições assíncronas. Esta *thread* é inserida em

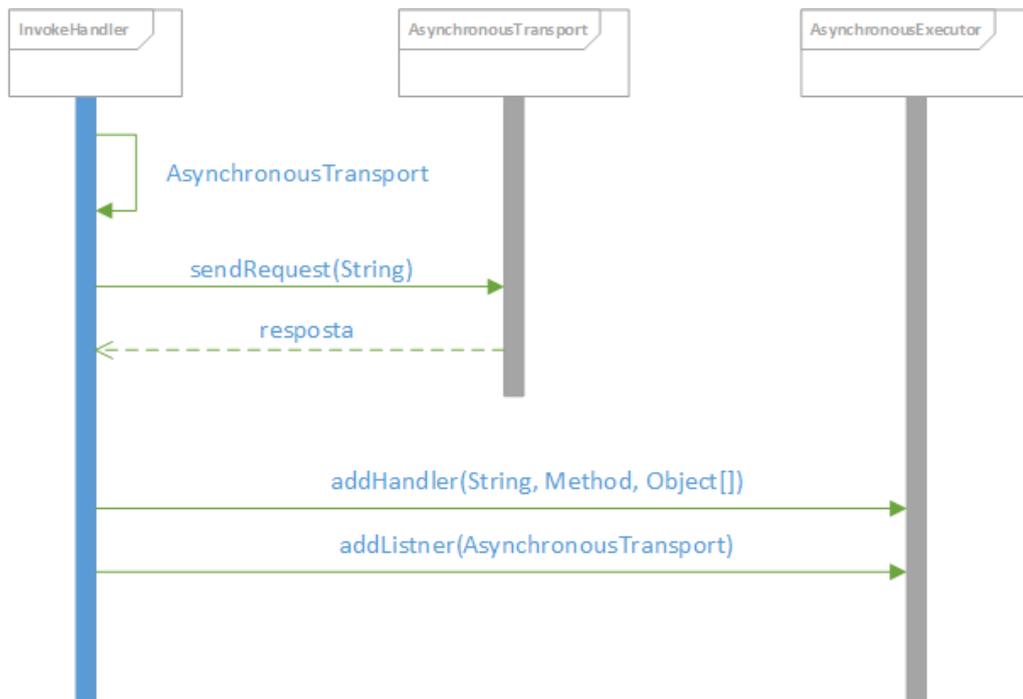


Figura 3.10: Fluxo de execução assíncrona do cliente.

um *pool* de *threads* e associada ao *HandlerManager*, possibilitando assim que, as requisições recebidas sejam encaminhadas aos objetos *callback* correspondentes, de forma semelhante ao que acontece com os objetos remotos do lado do servidor.

Do lado do servidor, a requisição é tratada de forma semelhante a já apresentada no bloco do invocador. A principal diferença consiste na criação do *proxy* do servidor para comunicação com o *callback* do cliente, conforme demonstra do diagrama da figura 3.11.

Ao realizar a conversão dos parâmetros obtidos pela requisição JSON-RPC, a classe *JsonRpcMethod* localiza o argumento com a anotação *Callback*. Neste momento o objeto *AsynchronousTransport*, que possui os dados da requisição do cliente, é utilizado para criar uma instância da classe *AsynchronousHandler*. A classe *AsynchronousHandler*, por sua vez, possui a lógica para tratar uma requisição do *proxy* e encaminha-la através do retorno assíncrono da requisição iniciada pelo cliente.

Por fim, o retorno do método é então encapsulado em uma resposta JSON-RPC e retornado ao cliente, ao passo que, o objeto *proxy* (que representa o *callback*) poderá ser utilizado pelo servidor para a chamada de métodos do lado do cliente, permitindo assim o retorno de respostas assíncronas.

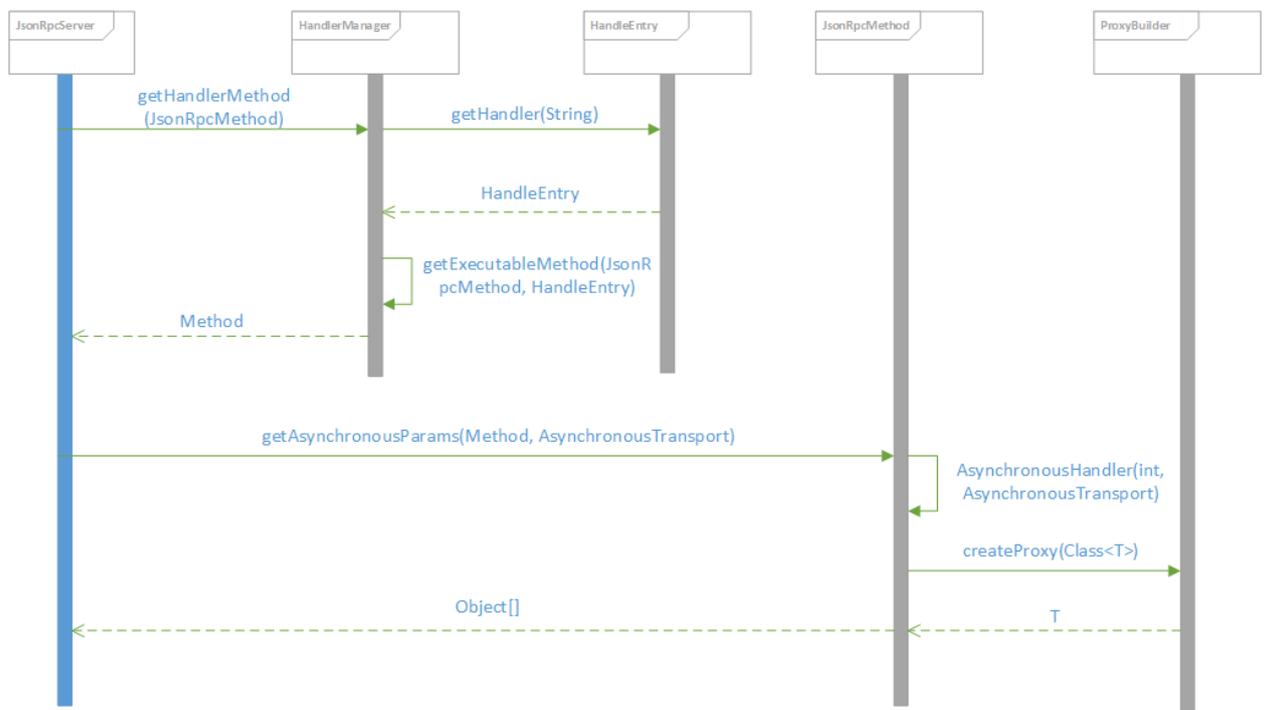


Figura 3.11: Fluxo de execução assíncrona do servidor.

TESTES DE DESEMPENHO

4.1 Testes de Desempenho

Muitos dos sistemas computacionais empresariais de hoje são constituídos por *middlewares* de objetos distribuídos. Tais sistemas são comuns em setores como telecomunicações, automação industrial, finanças, manufatura e governo, os quais, muitas vezes, suportar aplicações que são críticas para determinadas operações de negócio. Devido a isso, o *middleware* de objetos distribuídos é muitas vezes sujeito a condições de extrema exigência em questão de desempenho e confiabilidade.

Esta sessão tem por objetivo descrever os testes de desempenho realizados sobre a ferramenta JETY. Para isto, é feita uma comparação entre a ferramenta aqui apresentada e a API para comunicação remota nativa da linguagem Java, o RMI.

4.1.1 Hardware utilizado

Na realização dos testes, foram utilizadas máquinas distintas para a aplicação cliente e servidora.

Maquina cliente:

- Intel Core i5 – 2.80 GHz
- 6,0 GB – RAM
- Windows 7 – 64 bits

Maquina servidora:

- Intel Core i5 – 2.80 GHz
- 12,0 GB – RAM
- Windows Server 2008 – 64 bits

4.1.2 Cenário de teste

O cenário dos testes efetuados consiste em uma aplicação servidora simples, que possibilita a leitura de variáveis de processo de forma simulada. Tanto a aplicação servidora construída em JETTY como a construída em RMI, funcionam de forma semelhante disponibilizando um método de leitura de variáveis que recebem como parâmetro um objeto que descreve um servidor (conforme apresentado na figura 4.1).

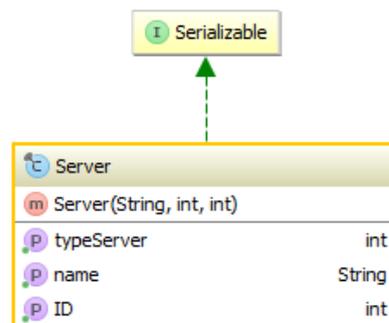


Figura 4.1: Diagrama de classe – Server.

Em resposta a requisição de leitura, é retorna ao cliente uma lista contendo 10 objetos *TagItem*, que simulam o retorno obtido a se ler variáveis de processo de um determinado servidor. O objeto trafegado é representado no diagrama da figura 4.2.

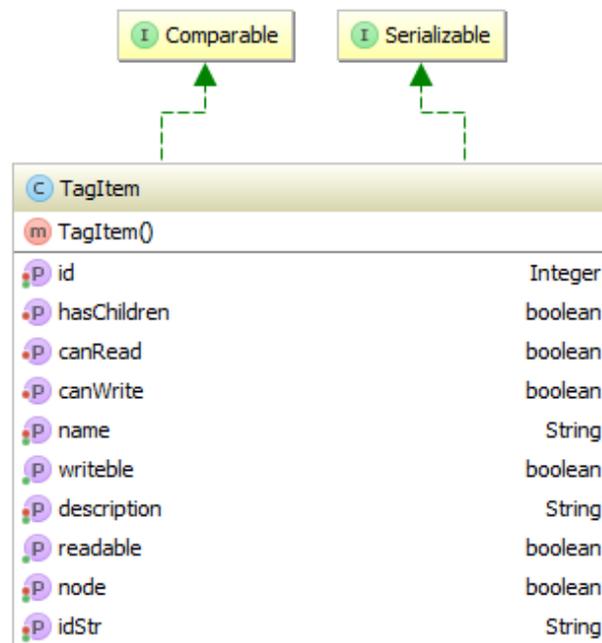


Figura 4.2: Diagrama de classe – TagItem.

Já do lado do cliente, temos aplicações simples que, executam a comunicação remota de acordo com a tecnologia em questão (JETTY e RMI), realizam um determinado número de requisições síncronas em sequência (sem tempo de espera entre as requisições) e por fim, calculam o tempo decorrido no bloco de requisições executado.

4.1.3 Resultados obtidos

O teste realizado consiste em uma série de seis testes onde o cliente realiza um número n de requisições. Cada série foi executada iterativamente 100 vezes, de forma a eliminar ocasionais oscilações na rede e assim obtermos uma estimativa do tempo médio gasto na execução das requisições.

A tabela 4.1 demonstra a execução da aplicação cliente para cada número de requisições, bem como o tempo médio (em milissegundos) gasto durante a execução das mesmas.

A partir do gráfico da figura 4.3, é possível verificar que, o *middleware* aqui apresentado, obteve um desempenho similar a tecnologia nativa da linguagem Java, apresentando um desempenho ligeiramente inferior para as séries com menor número de requisições e chegando a superar a performance da tecnologia RMI para um alto número de requisições.

Requisições	JETY	RMI
100	406 ms	138 ms
500	1100 ms	695 ms
1000	1617 ms	1112 ms
5000	4935 ms	5164 ms
10000	87411 ms	10404 ms
15000	12015 ms	15378 ms

Tabela 4.1: Resultado do tempo médio por requisições.

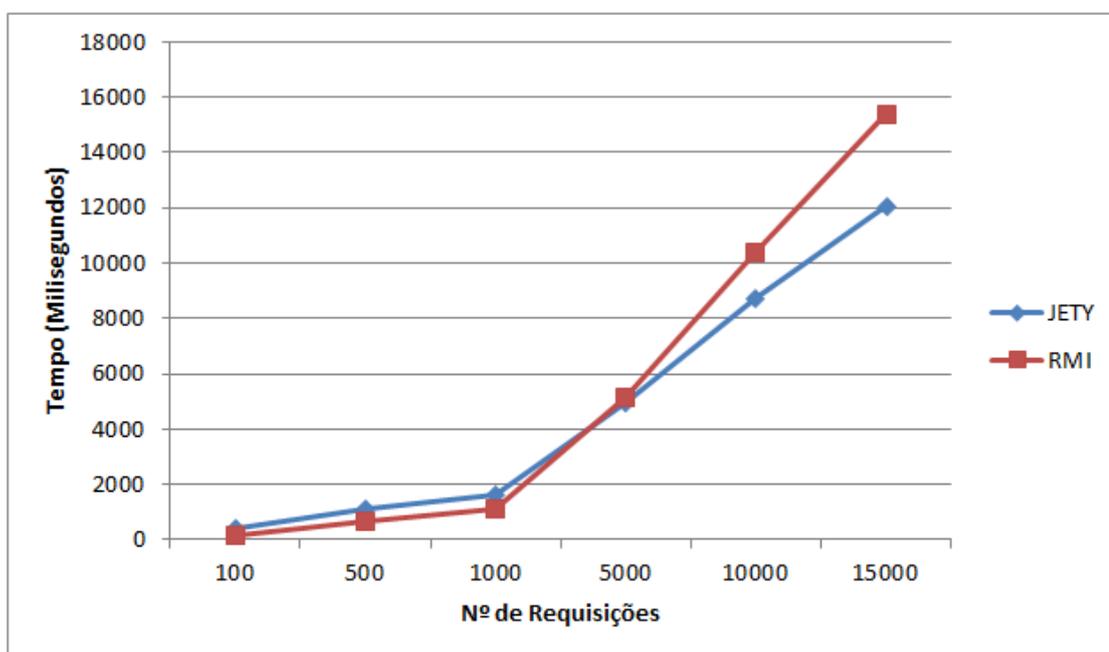


Figura 4.3: Comparação JETY x RMI.

A partir dos resultados obtidos é possível realizar uma média ponderada a fim de obtermos o tempo médio gasto durante toda a bateria de testes. Em média, a aplicação rodando em JETY gastou 9310,68 milissegundos para executar 5300 requisições, ao passo que, a aplicação executando em RMI gastou 11455,38 para executar o mesmo número de requisições.

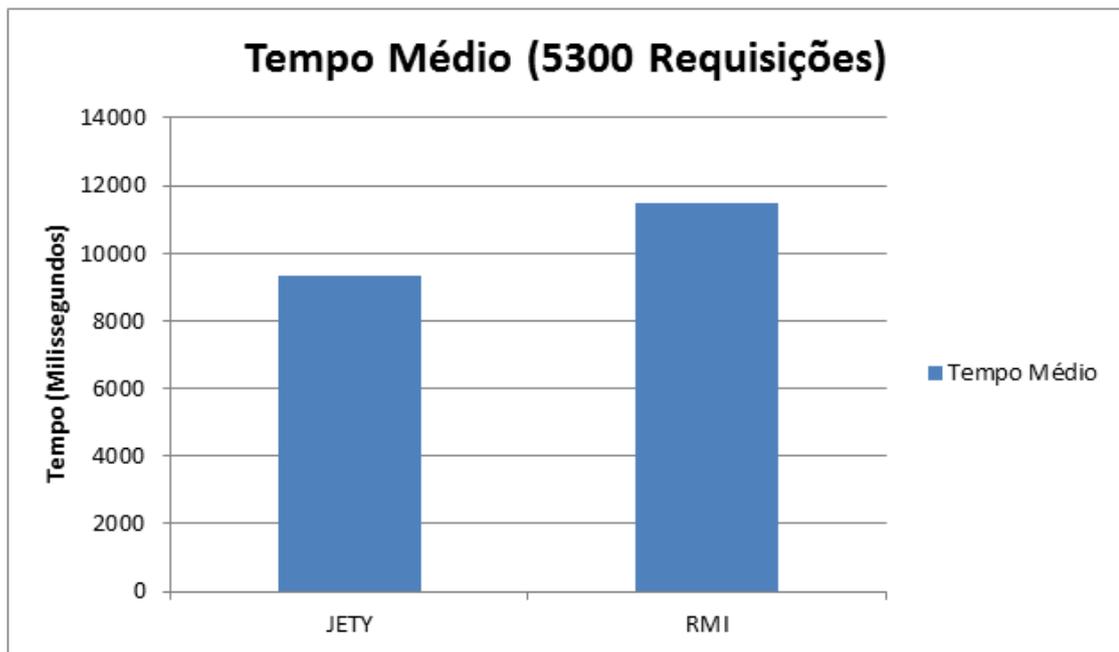


Figura 4.4: Tempo médio durante os testes.

Semelhantemente, os resultados obtidos nos permitem avaliar o tempo médio gasto por cada requisição durante a série de teste. Desta forma, obtemos 1,76 para a aplicação utilizando JETY contra 2.16 da aplicação utilizando RMI.

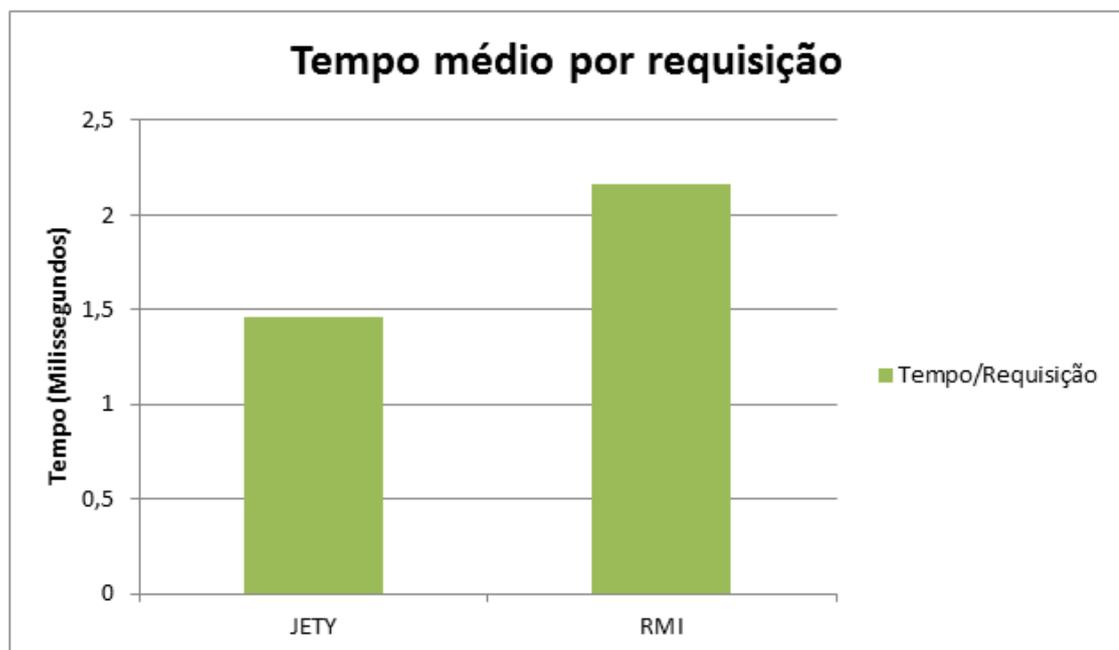


Figura 4.5: Tempo médio por requisição.

CASO DE USO - SISTEMA DE AQUISIÇÃO DE DADOS INDUSTRIAIS

5.1 Estado da arte

Apesar do considerável grau de maturidade e aceitação na indústria, a solução OPC eventualmente apresenta complicações para se adaptar à necessidade industrial devido a dificuldade de configuração do ambiente necessário para utilização da mesma. Outro fator a ser considerado é que, uma vez que o OPC oferece uma integração de diversos níveis da indústria, desde os dispositivos do chão-de-fábrica até aos sistemas corporativos, o mesmo provê uma grande quantidade de métodos e funcionalidades, muitas das quais acabam se mostrando desnecessárias para aplicações de alto nível, podendo até mesmo tornar confusa a sua utilização por partes dos desenvolvedores de aplicações.

Visando facilitar o desenvolvimento, o OPC UA disponibiliza um conjunto de SDKs para programação em diferentes ambientes e linguagens de programação (Ansi C/C++, .Net e Java). Uma das funcionalidades presentes no SDK é o *Server Toolkit* que fornece um conjunto de classes que um servidor UA deve disponibilizar. Apesar desta camada de abstração, ainda se faz necessário o conhecimento e manipulação de uma grande quantidade de métodos e classes para o desenvolvimento de uma aplicação cliente OPC UA, conforme demonstrado na figura 5.1.

Diferente da solução proposta pelo OPC UA, que visa integrar todas as especificações OPC anteriores (provendo acesso a dados atuais e de histórico, bem como alarmes e eventos), a tecnologia CyberOPC é baseada na especificação OPC-XML DA que se limita

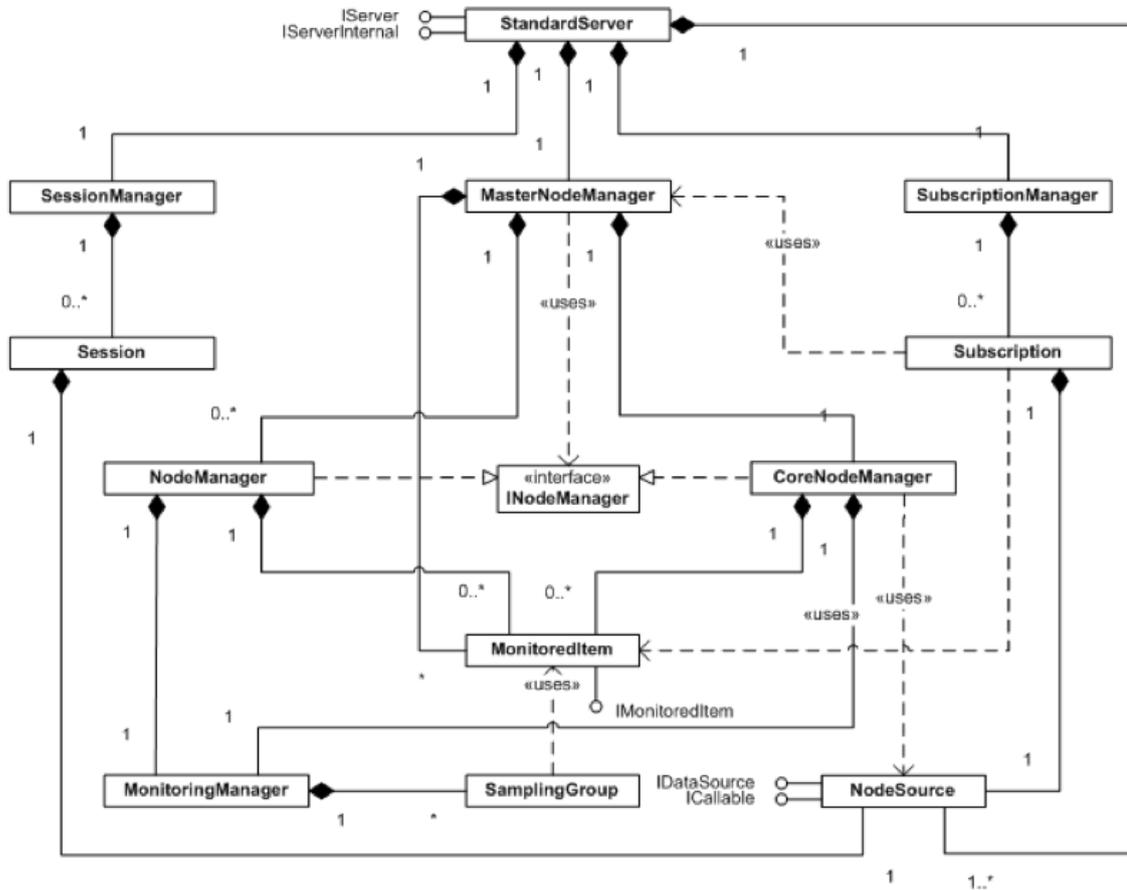


Figura 5.1: Componentes do *Server Toolkit*.

a prover um interface para leitura e escrita de dados em tempo real. Apesar das melhorias de desempenho proposta pelo CyberOPC, o mesmo apresenta um grande conjunto de métodos *opc-like*, ou seja, se faz necessário um conhecimento da tecnologia OPC por parte do desenvolvedor.

A aplicação do *middleware* desenvolvido ao BR-Collector visa combater essas deficiências, propondo uma arquitetura que possibilite a aquisição de dados industriais provenientes de diversas fontes de dados como, por exemplo, OPC DA, OPC A&E e PI, a fim de disponibiliza-los de forma simples e independente de plataforma para camadas superiores. A utilização do *middleware* possibilita a criação de um serviço web simples com suporte a requisições assíncronas, importante passo para a futura criação de assinaturas para receber dados em tempo real.

5.2 Arquitetura web aplicada ao BR-Collector

Como já apresentado, a ferramenta BR-Collector realiza uma abstração de diversas fontes de dados utilizando-se de drivers que fornecem as informações dos servidores de dados de uma forma padronizada e independente do método utilizado para obtê-los. Um driver tem por função transformar a conexão com qualquer servidor de dados em uma conexão homogênea, a fim de que, a mesma possa ser interpretada pelas camadas superiores do BR-Collector. Além disso, os *drivers* OPC presentes no BR-Collector são reimplementação das especificações OPC que eliminam a necessidade de código escrito em linguagem nativa provendo assim um melhor desempenho e uma fácil manutenção do BR-Collector, aliado a capacidade de ser executado em múltiplas plataformas. O BR-Collector fornece aos desenvolvedores uma API simplificada que permite a fácil utilização de dados industriais. A figura 5.2, por exemplo, ilustra a simplicidade do fluxo esperado para a captura de dados de variáveis de processo.

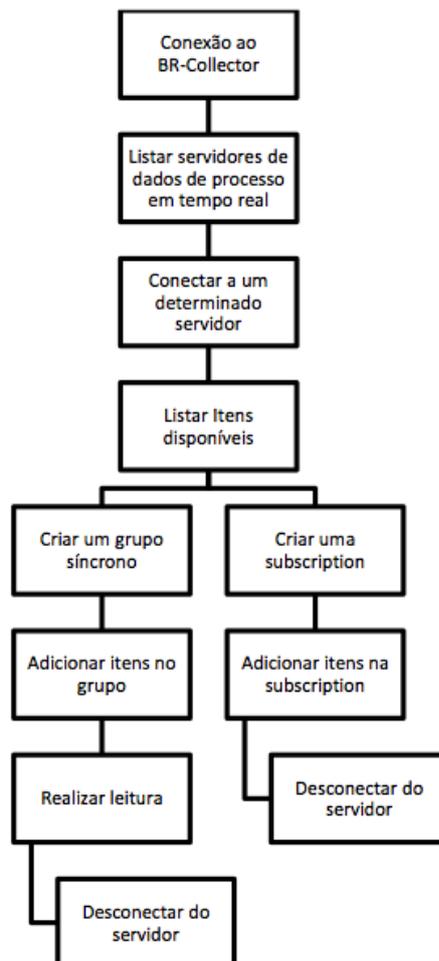


Figura 5.2: Fluxo para coleta de dados de variáveis de processo no BR-Collector.

Atualmente, o transporte dos dados do BR-Collector é realizado através da tecnologia Java RMI (*Remote Method Invocation*) que permite que outros aplicativos feitos em Java possam chamar métodos para a captura de dados no BR-Collector de forma remota. A arquitetura aplicada, visa substituir a camada de comunicação RMI e adicionar uma comunicação remota via serviço web, permitindo assim a utilização do BR-Collector por aplicações construídas em diferentes tecnologias e não apenas em Java.

Além da abstração da comunicação via protocolo HTTP provida pelo *middleware* desenvolvido, a sua utilização possibilita que as mudanças aplicadas ao BR-Collector, tanto do lado do servidor como do lado do cliente, sejam mínimas, uma vez que os conceitos de definição de interfaces e acesso a objetos remotos através de proxy se dá de maneira similar ao implementado pelo RMI.

Tal arquitetura consiste em um modelo cliente/servidor HTTP (vide figura 5.3) onde as funcionalidades providas pela API do BR-Collector são encapsuladas em uma interface, inseridas no *middleware* e, por fim, disponibilizadas como serviço através de uma servlet. Na aplicação cliente, o *middleware* efetua o papel do cliente HTTP que recupera os dados codificados em Json para que os mesmos sejam utilizados na aplicação cliente.

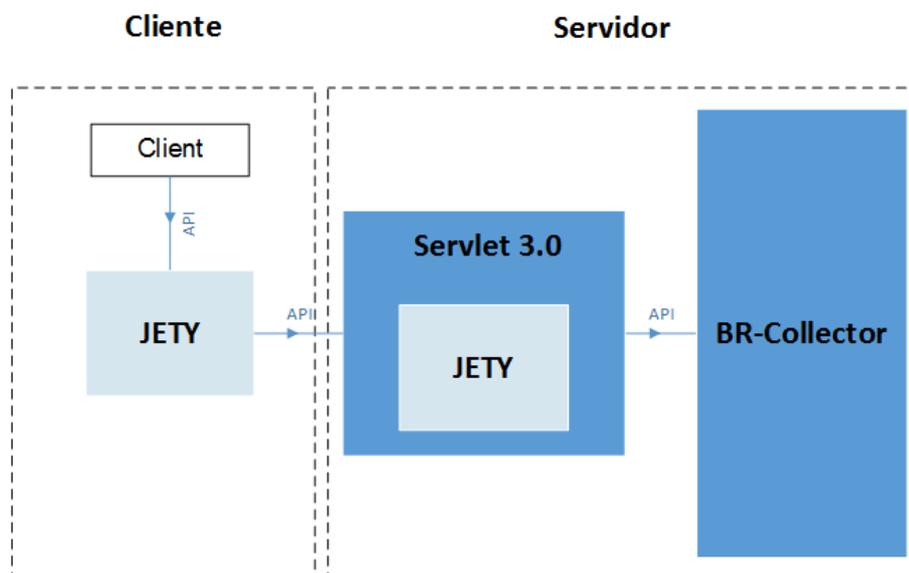


Figura 5.3: Modelo da arquitetura proposta.

5.2.1 Servidor web do BR-Collector

A implementação da arquitetura web do BR-Collector consiste em codificar a definição dos serviços remotos em interfaces Java provendo assim a API que será vista pelo cliente para que o mesmo possa solicitar os serviços. A implementação do serviço remoto é

codificada em uma classe. Logo, as interfaces definem o comportamento e as classes definem a implementação. A classe que implementa o comportamento roda do lado do servidor, ao passo que, a classe que roda no cliente atua como um *proxy* para o serviço remoto.

Dentre as diversas documentações providas pelo BR-Collector, a fim de validar a arquitetura proposta, será detalhada neste documento a implementação para a especificação para captura de variáveis de processo em tempo real. No entanto, a implementação das demais funcionalidades ocorre de maneira similar e igualmente fácil.

Uma vez que toda a lógica de aquisição e escrita de variáveis de processo já se encontra implementada no BR-Collector, a criação de um servidor web pode ser efetuada de forma bastante simples utilizando-se de uma servlet associada ao *middleware* JETTY. A criação da servlet consiste na criação de uma classe que estende a classe *HttpServlet*. A partir da especificação servlet 3.0, é possível ainda informar os parâmetros funcionais da servlet via *annotation*, como, por exemplo, o nome da servlet, o valor da URL e se esta terá ou não suporte assíncrono, conforme demonstra o código 5.1.

Código 5.1: Servlet Real Time DA

```
1 @WebServlet(name = "brcollector", value = {"/rtda"}, asyncSupported
    = true)
2 public class CollectorServlet extends HttpServlet
```

Ao iniciar a servlet (no construtor da classe) o *middleware* é então utilizado para associar os serviços que serão providos pela servlet. Para isso, basta criar uma instancia de classe *JsonRpcServer* e adicionar um ou mais serviços através do método *addHandler* (descrito no código 3.1).

O código 5.2 demonstra o procedimento efetuado para adicionar os serviços de variáveis de processo em tempo real do BR-Collector. O método *addHandler* recebe então como parâmetro um identificador (que deverá ser conhecido pelo cliente, a fim de que, ele invoque o objeto remoto correto), um objeto da classe que implementa a lógica do serviço (no caso, uma instancia da classe *ServiceRDAManager*) e por fim, uma ou mais interface que descrevem os serviços. Tal interface compõe a API do BR-Collector e é descrita no apêndice A.

Código 5.2: Handler para tratar variáveis de processo

```
1 private JsonRpcServer bind() {
2     JsonRpcServer executor = new JsonRpcServer();
3
4     ServiceRDA daImp = new ServiceRDAManager();
5     executor.addHandler("RDA", daImp, ServiceRDA.class);
```

```
6 // add more services here
7
8 return executor;
9 }
```

Uma vez adicionado os serviços, basta indicar que as requisições realizadas a servlet serão tratadas pelo *middleware*. Isto é feito de forma simples sobrescrevendo o método *doPost* da servlet e encapsulando seus parâmetros em um objeto *JsonRpcServerTransport* conforme demonstra o código 5.3

Código 5.3: Ação do método POST

```
1 @Override
2 protected void doPost(HttpServletRequest req, HttpServletResponse
    resp) throws ServletException, IOException {
3     executor.execute(new JsonRpcServerTransport(req, resp));
4 }
```

Conforme visto na figura 5.2, o BR-Collector trabalha com duas formas de obtenção de dados de variáveis de processo: o método síncrono e o método assíncrono. Pelo método síncrono o cliente tem a responsabilidade de checar periodicamente os valores das variáveis de processo. Já no método assíncrono, o cliente realiza uma requisição ao BR-Collector informando que deseja realizar uma assinatura (*subscription*) para monitorar um grupo de variáveis de processo de um determinado servidor. A partir desse instante, o BR-Collector é quem tem a função de notificar o cliente após uma alteração significativa do valor dessas variáveis.

Para efetuar a comunicação assíncrona, o servidor precisa indicar quais métodos deverão ser executados assincronamente. Para isso, o *middleware* disponibiliza a anotação *AsynchronousMethod* que deverá ser utilizada na interface que define o mesmo.

Uma vez que o cliente efetua uma requisição assíncrona, o servidor deverá informa-lo a medida que uma resposta estiver disponível. Para tal, é necessário a criação de uma classe *callback*. A interface *callback* precisa ser comum tanto a aplicação servidora como ao cliente e deverá ser anotada com o instrução *Callback*, como demonstra o exemplo do código 5.4.

Código 5.4: Interface callback

```
1 @Callback
2 public interface CallbackRDA {
3     void onDataChange(List<DataRDA> hm);
4 }
```

A interface define então os métodos disponíveis para que o servidor possa notificar o cliente quando novos dados estiverem disponíveis.

5.2.2 Cliente web do BR-Collector

Uma vez disponibilizada a API contendo as interfaces de cada serviço, a construção de uma aplicação cliente utilizando o *middleware* se torna trivial.

A partir da URL de acesso ao servidor do BR-Collector, é possível solicitar a comunicação remota para acesso a variáveis de processo em tempo real utilizando-se da interface da API.

O código 5.5 demonstra o procedimento para acesso a servlet do BR-Collector. Primeiramente é inicializado uma instancia da classe *JsonRpcClientTransport*, a qual recebe como parâmetro a URL de conexão. Feito isto, um objeto da classe *JsonRpcClient* é instanciado a fim de que seja possível obter uma referência ao objeto remoto *ServiceRDAManager* previamente adicionado no servidor. A partir do método *getClassFromServer*, obtemos essa referência passando como parâmetro o transporte recém criado, o identificador do objeto cadastrado pelo servidor (RDA) bem como a interface *ServiceRDA*.

Código 5.5: Conexão do cliente

```
1 String url = "http://localhost:22873/BRCollector-webservice/rtda";
2
3 JsonRpcClientTransport transport;
4 transport = new JsonRpcClientTransport(new URL(url));
5
6 JsonRpcClient invoker = new JsonRpcClient();
7 ServiceRDA rda = invoker.getClassFromServer(transport, "RDA",
    ServiceRDA.class);
```

5.2.3 Cliente web C#

Para demonstrar a flexibilidade provida pela camada do *middleware*, como continuidade deste trabalho, foi desenvolvida uma API cliente para acesso ao BR-Collector na linguagem de programa C#. Tal API implementa alguns dos blocos básicos do lado do cliente, apresentados no capítulo de desenvolvimento. Trata-se basicamente das funcionalidades de empacotamento no estilo JSON-RPC e transporte de informação via HTTP.

Mapeamento da API

O passo inicial consiste em criar uma API do BR-Collector em C#, semelhantemente a existente em Java. Em outras palavras, as interfaces que provem os serviços assim como as classes de domínio que serão trafegadas pela rede, deverão ter seus equivalentes em C# uma vez que é necessário manter a integridade entre os dados transmitidos.

Por exemplo, ao solicitar a listagem de tags a partir do método *searchTagItems* da API do BR-Collector, o cliente espera receber uma lista de objetos *TagItem*. Desta forma, é necessário que o cliente C# interprete corretamente a representação deste objeto em JSON e crie um objeto correspondente em C#.

Devido a sua popularidade em aplicações web, existem diversas bibliotecas que possibilitam a serialização de objetos de uma determinada classe em uma *string* JSON. No desenvolvimento do cliente C#, foi utilizada a biblioteca JSON.NET, desenvolvida pelo grupo *Newtonsoft* [Newtonsoft 2013].

Desta forma, a utilização de tal biblioteca permite a fácil associação das classes de domínio em sua representação JSON.

Tomando como exemplo a classe *TagItem*, temos a seguinte classe de domínio (código 5.6 associado a representação JSON apresentada no código 5.7).

Código 5.6: Classe TagItem

```

1 public class TagItem : IComparable<TagItem>
2 {
3     [JsonProperty("id")]
4     public int Id { get; set; }
5
6     [JsonProperty("idStr")]
7     public string IdStr { get; set; }
8
9     [JsonProperty("name")]
10    public string Name { get; set; }
11
12    [JsonProperty("description")]
13    public string Description { get; set; }
14
15    [JsonProperty("children")]
16    public bool Children { get; set; }
17
18    [JsonProperty("node")]
19    public bool Node { get; set; }
20
21    [JsonProperty("canRead")]

```

```

22 public bool CanRead { get; set; }
23
24 [JsonProperty("canWrite")]
25 public bool CanWrite { get; set; }
26
27 public TagItem()
28 {
29     this.CanRead = true;
30     this.CanWrite = false;
31 }
32 }

```

Código 5.7: Representação JSON de um objeto TagItem

```

1 "idStr":"textual.color","name":"textual.color","children":false,"
  node":false,"canRead":true,"canWrite":false

```

Empacotamento

Na API C#, a função de empacotamento da requisição e resposta é efetuada de forma bastante simples, utilizando-se das classes *JsonRpcRequest* e *JsonRpcResponse*. Tais classes funcionam como classes de domínio que armazenam parâmetros necessários para uma requisição e resposta no estilo JSON-RPC. Tais classes são apresentadas no diagrama da figura 5.4.

Novamente, é utilizada a biblioteca JSON.NET para conversão em uma *string* JSON.



Figura 5.4: Diagrama das classes *JsonRpcRequest* e *JsonRpcResponse*.

Handler de requisição do cliente

Conforme anteriormente especificado, esta camada tem por função gerenciar as conexões entre o cliente e o servidor, enviar e receber dados bem como o tratamento de eventuais erros de conexão.

A classe *ClientTransport* agrega esta função, executando o fluxo das figuras 3.6 e 3.7, para execução síncrona e assíncrona respectivamente.

De forma semelhante ao que ocorre na implementação do *middleware*, o *handler* abre uma conexão HTTP com a URL passada como parâmetro e prepara o cabeçalho HTTP a ser utilizado na conexão.

O *handler* também se encarrega de estabelecer e gerenciar o processo de conexão e enviar e receber os dados trafegados através de um fluxo de bytes.

Solicitante

Conforme já explicitado, o bloco solicitante se encarrega de receber os parâmetros passados pelo cliente e preencher os objetos das classes de empacotado a fim de que os mesmos sejam serializados e enviados ao *handler* de requisição para transporte. Ao receber uma resposta, a mesma é encaminhada ao cliente como retorno do método.

O bloco solicitante é composto pela classe *InvokeHandler* e seu fluxo básico é semelhante ao apresentado na figura 3.4.

5.2.4 Aplicação cliente em C#

A API C# desenvolvida é apresentada no apêndice E. De posse desta API, a criação de uma aplicação cliente é efetuada de forma semelhantemente fácil a já apresentada no cliente Java.

A conexão é efetuada criando-se uma instância da classe *ClientTransport* passando como parâmetro a URL de acesso ao servidor do BR-Collector. Desta feita, um objeto da classe *InvokeHandler* é então instanciado e o mesmo é associado ao objeto remoto "RDA" cadastrado no servidor.

Por se tratar de um cliente mais simples e específico para aplicação do BR-Collector, diferente do que acontece no lado do cliente utilizando JETTY, a referência ao objeto remoto do lado do servidor não acontece via *proxy*. Desta forma, a interface *ServiceRDA*, que representa a API para comunicação com o BR-Collector em C#, necessita interagir diretamente com o *InvokeHandler*, ou seja, fica a cargo do programador da API conhecer e encaminhar a chamada do método ao solicitante.

Uma vez obtida a referencia a classe *ServiceRDA*, se torna possível a chamada de métodos da mesma, conforme apresenta o exemplo do código 5.8.

Código 5.8: Conexão do cliente

```
1 static void Main(string[] args)
2 {
3     string url = "http://localhost:22873/BRCollector-webservice/
         jsonrpc";
4
5     ClientTransport transport = new ClientTransport(url);
6     InvokeHandler handler = new InvokeHandler("RDA", transport);
7
8     ServiceRDA service = new ServiceRDA(handler);
9
10    List<string> result = service.ListServers();
11
12    string serverStr = result[0];
13
14    Debug.WriteLine("Vai conectar a: " + serverStr);
15    Server server = service.ConnectServer(serverStr);
16
17    bool support = service.SupportsHierarchialListing(server);
18    Debug.WriteLine("Suporta listagem hier rquica?: " + support);
19
20    if (support) {
21        List<TagItem> listItems = service.BrowseNode(server, null);
22
23        foreach (TagItem tagItem in listItems) {
24            Debug.WriteLine(tagItem);
25            List<TagItem> listItems2 = service.BrowseNode(server, tagItem)
                ;
26
27            foreach (TagItem tagItem1 in listItems2) {
28                Debug.WriteLine(" ->> " + tagItem1);
29            }
30        }
31    }
32 }
```

5.3 Supervisório para dispositivos com sistema operacional Android

Através da tecnologia web service, associada ao *middleware* desenvolvido por este trabalho, é possível que uma gama de novas aplicações possam obter os dados fornecidos pelo BR-Collector permitindo assim que o desenvolvedor tenha liberdade na escolha da linguagem de programação e plataforma a ser utilizada.

Neste contexto, foi desenvolvido o sistema *Industrial Supervisor* que consiste em um sistema supervisório de processos industriais para dispositivos móveis com sistema operacional Android.

O aplicativo visa fornecer uma série de ferramentas que auxiliam no monitoramento de processos indústrias possibilitando a fácil visualização dos estados das variáveis do processo e a realização de análises estatísticas. Essas ferramentas incluem: indicador de estado de variáveis discretas, indicador de valores instantâneos de variáveis contínuas, gráfico de barras, gráfico de séries temporais, gráfico de pizza, velocímetro e termômetro [Silva 2013]. O sistema também possibilita atuar em certos tipos de variáveis que permitam escrita de valores, isto é, é possível enviar um sinal para o processo indicando qual valor o usuário gostaria de estabelecer para uma determinada variável discreta ou contínua.

A figura 5.5 apresenta a tela do sistema no modo de monitoramento que possibilita ao operador supervisionar os processos no Modo de Execução.



Figura 5.5: Aplicativo Industrial Supervisor no Modo de Execução.

A utilização do *middleware* no lado do cliente Android possibilitou a criação de um

CAPÍTULO 5. CASO DE USO - SISTEMA DE AQUISIÇÃO DE DADOS INDUSTRIAIS⁵⁷

sistema leve e ao mesmo tempo poderoso, uma vez que toda a lógica de captura e escrita de variáveis de processos, necessárias para um sistema supervisor encontra-se do lado do servidor e é apenas disponibilizada para o cliente em forma de serviço web. Além desta vantagem, o desenvolvedor se encontra totalmente livre da lógica de comunicação web abstraída pelo *middleware* sendo necessário apenas o conhecimento da API do BR-Collector.

O sistema *Industrial Supervisor* demonstra a inerente versatilidade do trabalho aqui desenvolvido permitindo ampliar as fronteiras da área industrial, trazendo para ela novas tecnologias que permitam auxiliar no monitoramento de processos.

CONCLUSÃO

O principal objetivo da biblioteca desenvolvida é possibilitar aos desenvolvedores uma forma fácil e intuitiva de criar sistemas distribuídos através de requisições JETY. Para isso, o programador não deverá encontrar dificuldades em adaptar a sua forma de desenvolvimento atual para um arquitetura de comunicação distribuída. O *middleware* JETY cumpre essas expectativas, provendo uma API de fácil assimilação que encapsula toda a comunicação HTTP além de permitir efetuar chamadas remotas de maneira similar a chamadas locais.

A biblioteca JETY possibilita a construção de aplicações servidoras que provêem serviços a partir de objetos remotos. Utilizando-se das facilidades fornecidas pela tecnologia servlet 3.0, é possível criar aplicações robustas com suporte a comunicação síncrona e assíncrona. Para aplicações cliente, a API JETY comporta um cliente Java, o qual possibilita a fácil comunicação com a aplicação servidora. Desta forma, além da abstração da comunicação web, toda a comunicação com o protocolo de troca de dados utilizado pelo JETY é utilizada de forma transparente pelo desenvolvedor.

Além destas funcionalidades, o *middleware* foi desenvolvido utilizando-se de padrões largamente utilizados em sistemas distribuídos, o que possibilita a sua fácil manutenção bem como a adição de novas funcionalidades.

No quesito performance, os testes realizados demonstraram que, a solução apresentada por este trabalho obteve um desempenho satisfatório, permitindo a troca de mensagens com baixa latência, podendo assim suprir as necessidades dos mais diversos tipos de aplicações, mesmo aquelas em que o tempo de resposta é um fator crítico.

A biblioteca desenvolvida apresenta ainda a vantagem de trabalhar com padrões bem

difundidos, como é o caso do protocolo HTTP para troca de mensagem é da serialização via JSON que é largamente utilizada na web. Diferente da tecnologia RMI, que trabalha com padrões de serialização e transmissão de chamadas próprias da linguagem de programação Java, o *middleware* desenvolvido realiza requisições HTTP do tipo POST, onde os dados trafegados são encapsulados via JSON. Esta característica torna a arquitetura desenvolvida bastante flexível, possibilitando a criação de diversas APIs para comunicação com aplicação clientes/servidoras em diferentes linguagens de programação. Tal característica foi demonstrada, por exemplo, no desenvolvimento do cliente C#.

Além das diversas vantagens já citadas, o trabalho desenvolvido apresenta um grande avanço no que diz respeito a comunicação assíncrona, se mostrando uma das poucas alternativas para desenvolvimento via serviços web presentes no mercado com suporte a tal funcionalidade.

O suporte a comunicação assíncrona possibilita a notificação ao cliente por parte do servidor, característica essa que se mostra de extrema importância para aplicações industriais executando em tempo real. A associação do *middleware* desenvolvido por este trabalho ao BR-Collector resulta em um ambiente computacional modular para desenvolvimento de aplicações de forma rápida e robusta na área de suporte à operação de processos na área de petróleo e gás. Tal união visa suprir uma demanda presente na indústria no que diz respeito ao desenvolvimento de ferramentas de suporte a operação, fornecendo aos desenvolvedores uma API simplificada que permite a fácil utilização de dados industriais provenientes de diversas fontes de dados via web, possibilitando assim que os clientes possam receber dados, estejam eles nos mais diversos tipos de aparelhos, com suporte a qualquer linguagem de programação que possa realizar uma conexão HTTP.

A arquitetura apresentada neste trabalho elimina a restrição de dependência a uma única linguagem de programação, permitindo que os desenvolvedores tenham uma maior liberdade, não só na escolha da linguagem utilizada para implementar suas aplicações, mas também permitindo que o foco do desenvolvimento se resuma a aplicação em si, uma vez que, os detalhes da implementação estão encapsulados em forma de biblioteca, permitindo a utilização dos mesmos de uma forma fácil e eficiente.

A biblioteca desenvolvida apresenta um grande potencial a ser explorado, constituindo assim a base para uma gama de aplicações clientes. Busca-se ainda estender os resultados desse trabalho aplicando a estrutura desenvolvida a outras especificações do BR-Collector conforme forem se mostrando necessárias.

Apêndice A - BR-Collector-webAPI: ServiceRDA

Código 1: Métodos da API Service RDA

```
1 // Interface comum entre o servidor web e o cliente RDA.
2 public interface ServiceRDA {
3
4     // Lista os servidores disponiveis no BR-Collector para se obter
5     // dados de processo em tempo real.
6     List<String> listServers();
7
8     // Realiza a conexao com um determinado servidor de tempo real de
9     // variaveis de processo.
10    Server connectServer(String serverName);
11
12    // Desconecta de um servidor de dados em tempo real de variaveis
13    // de processo.
14    boolean disconnectServer(Server server);
15
16    // Verifica se um determinado servidor suporta listagem de itens
17    // hierarquica.
18    boolean supportsHierarchialListing(Server server);
19
20    // Buca por tags que satisfa am a mascara (express o regular).
21    List<TagItem> searchTagItems(Server server, String mask);
22
23    // Realiza a listagem dos itens filhos de um no especificado pelo
```

```
    parametro node.  
20 List<TagItem> browseNode(Server server, TagItem node);  
21  
22 // Metodo auxiliar que retorna informa es adicionais sobre um  
    determinado item.  
23 TagInfo getTagInfos(Server server, TagItem tagItem);  
24  
25 // Cria uma assinatura com o BR-Collector para receber dados de  
    variaveis de processo.  
26 @AsynchronousMethod  
27 ProcessVariableSubscription createSubscription(Server server,  
    String name, boolean active, int updateRate, float deadBand,  
    CallbackRDA callback);  
28  
29 // Remove uma determinada subscription.  
30 boolean removeSubscription(Server server,  
    ProcessVariableSubscription subscription);  
31  
32 // Adiciona itens a serem monitorados a uma determinada  
    subscription.  
33 List<Boolean> addItemToSubscription(Server server,  
    ProcessVariableSubscription subscription, TagItem... tagItem);  
34  
35 // Remove um ou mais itens de uma determinada subscription.  
36 List<Boolean> removeItemsFromSubscription(Server server,  
    ProcessVariableSubscription subscription, TagItem... tagItem);  
37  
38 // Cria um grupo para leitura sincrona.  
39 TagItemGroup createGroup(Server server, String groupName);  
40  
41 // Remove um determinado grupo.  
42 boolean removeGroup(Server server, TagItemGroup group);  
43  
44 // Adiciona um ou mais itens a um grupo.  
45 List<Boolean> addItemToGroup(Server server, TagItemGroup group,  
    TagItem... tagItem);  
46  
47 // Remove um ou mais itens de um grupo.  
48 List<Boolean> removeItemsFromGroup(Server server, TagItemGroup  
    group, TagItem... tagItem);  
49  
50 // Realiza a leitura dos valores atuais dos itens pertencente ao  
    grupo especificado.  
51 List<DataRDA> getCurrentValues(Server server, TagItemGroup  
    tagGroup);
```

```
52
53 // Escreve um determinado valor em um item.
54 List<Boolean> writeValues(Server serverGroup, List<GenericDataRDA>
    tagItemsToWrite);
55 }
```

Apêndice B - Diagrama de classes do Middleware

.1 Diagrama de classes por pacote

.1.1 Client

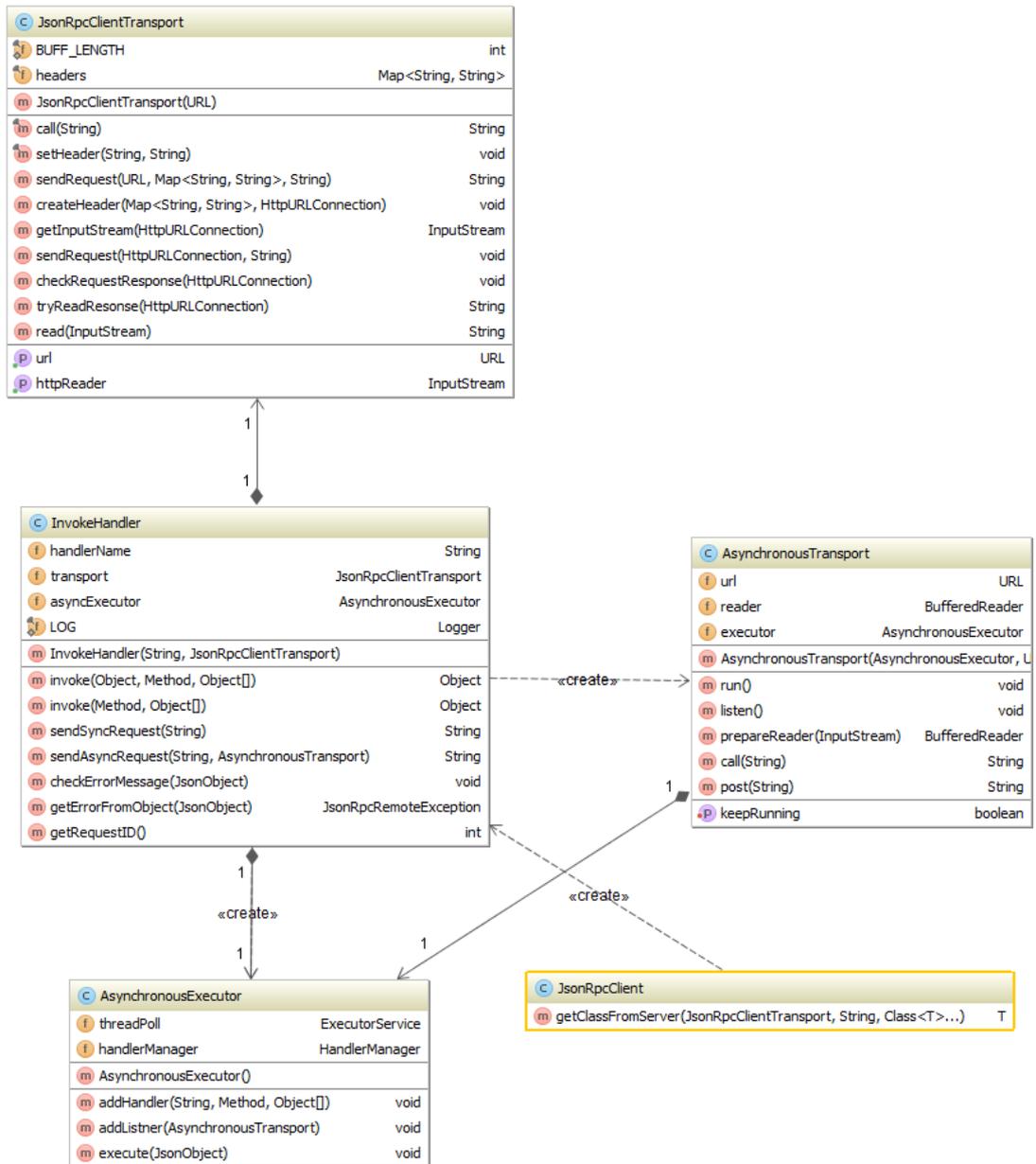


Figura 1: Classes do pacote Client.

.1.2 Domain

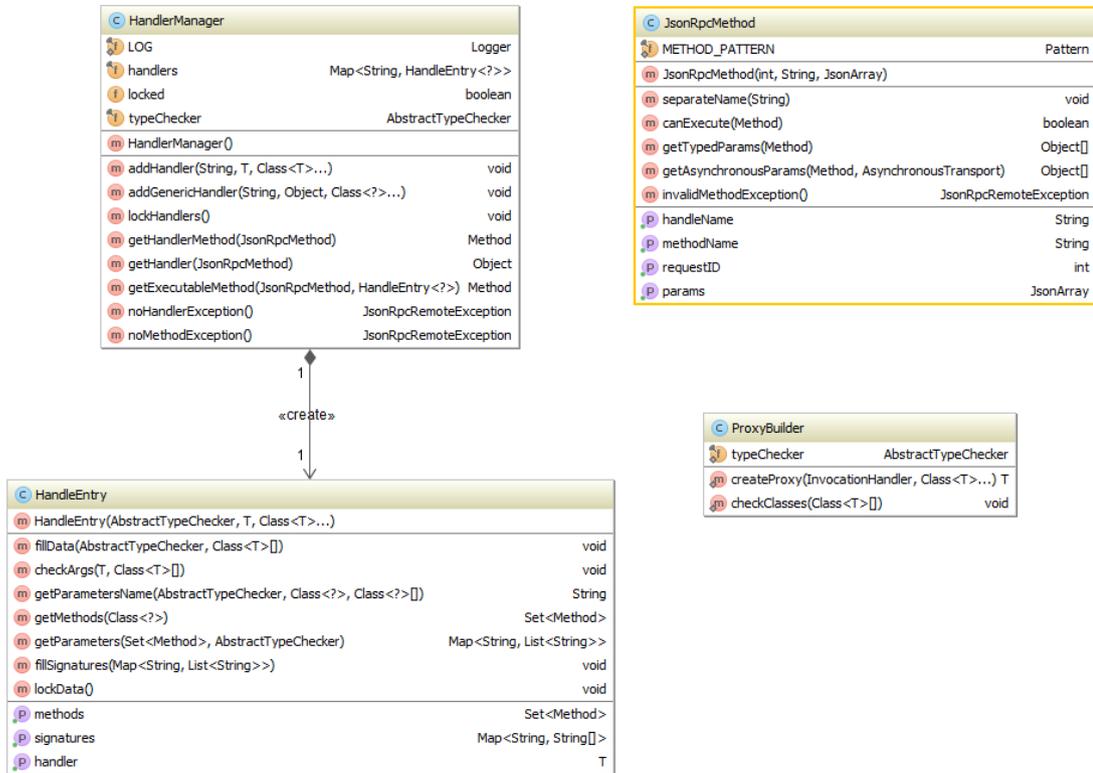


Figura 2: Classes do pacote Domain.

.1.3 Exception

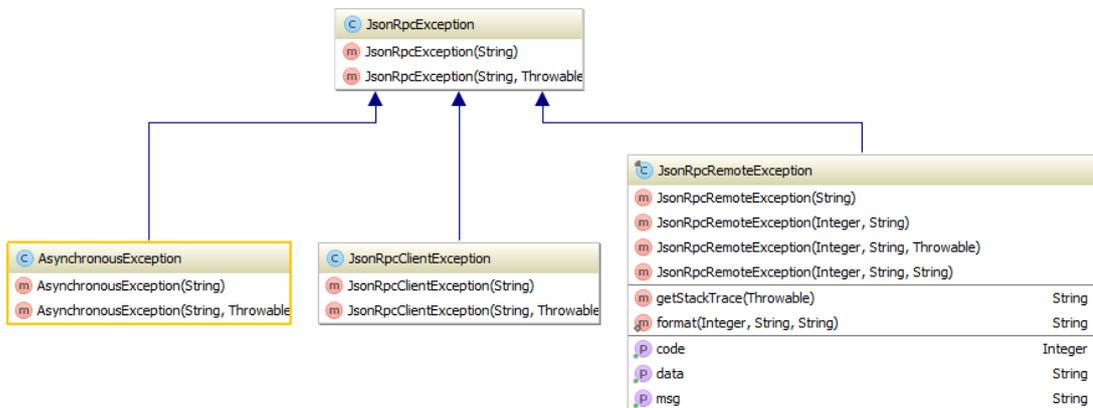


Figura 3: Classes do pacote Exception.

.1.4 Server

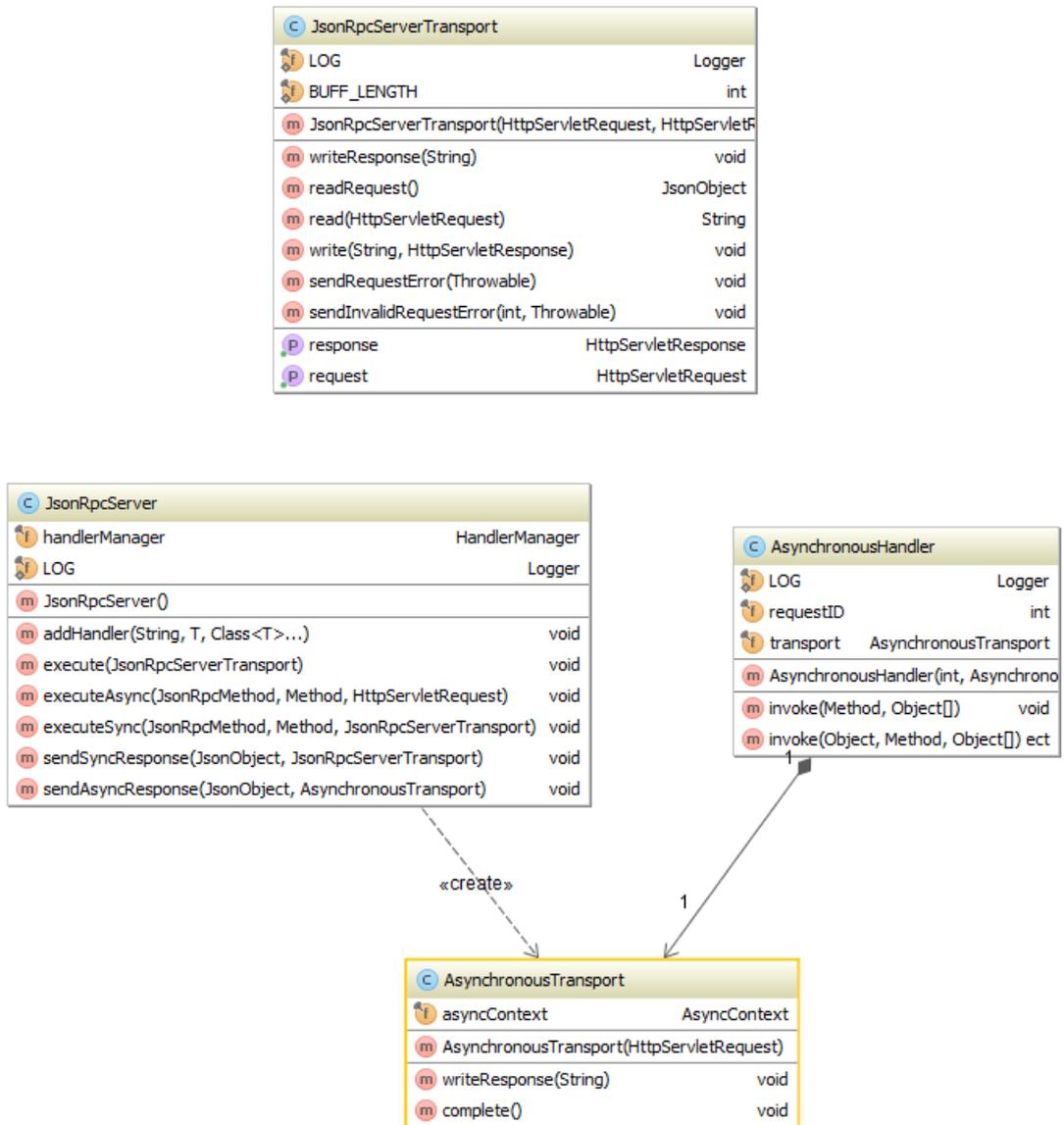


Figura 4: Classes do pacote Server.

.1.5 Typechecker

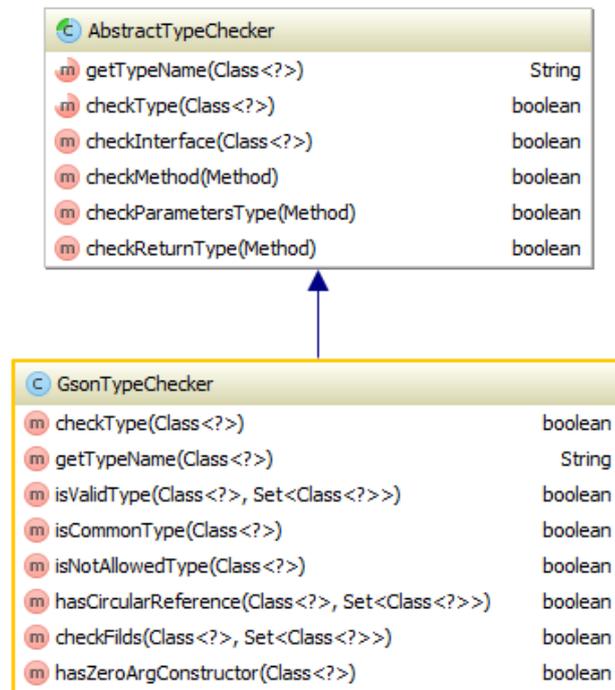


Figura 5: Classes do pacote Typechecker.

.1.6 Util

C JsonRpcConstants	
f	PROPERTY_VERSION String
f	PROPERTY_ID String
f	PROPERTY_CODE String
f	PROPERTY_MESSAGE String
f	PROPERTY_ERROR String
f	PROPERTY_RESULT String
f	PROPERTY_DATA String
f	PROPERTY_METHOD String
f	PROPERTY_PARAMS String
f	JSON_VERSION String
f	PARSE_ERROR_CODE int
f	INVALID_REQUEST_ERROR_CODE int
f	METHOD_NOT_FOUND_ERROR_CODE int
f	INVALID_PARAMS_ERROR_CODE int
f	INTERNAL_ERROR_CODE int
f	SERVER_ERROR_START int
m	JsonRpcConstants()
m	getServerError(int) int

C CommunicationUtil	
m	creatMethodRequest(int, String, Method, Object[]) JsonObject
m	createResponse(int, Object) JsonObject
m	getRequestMethod(JsonObject) JsonRpcMethod
m	getMetohd(JsonObject) JsonRpcMethod
m	createJsonMethodRequest(int, String, Object[]) JsonObject

C JsonConverter	
m	toJsonObject(String) JsonObject
m	fromJsonObject(JsonElement, Type) T
m	convertParameters(Method, JSONArray) Object[]
m	convertAsynchronousParameters(int, Method, JSONArray)
m	createJsonBuilder() Gson

C AsynchronousUtil	
m	isAsynchronous(Method) boolean
m	isCallback(Class<?>) boolean

C JsonRpcErrorBuilder	
m	getRequestParserError(Throwable) String
m	getInvalidRequestError(Integer, Throwable) String
m	getInvalidMethodError(Integer) String
m	buildError(Integer, JsonRpcRemoteException) String
m	buildError(Integer, Integer, String, String) String

Figura 6: Classes do pacote Util.

Apêndice C - Diagrama de classes de domínio

.2 Diagrama de classes trafegadas e suas representações em JSON

.2.1 Server

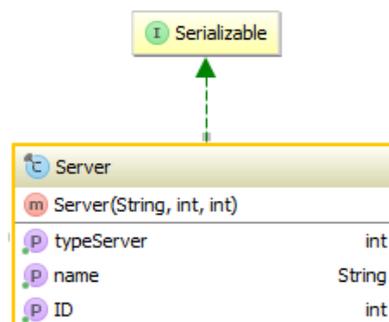


Figura 7: Classe Server

```
{ "name": "DA", "typeServer": 1, "serverGroupID": 1 }
```

.2.2 TagItem

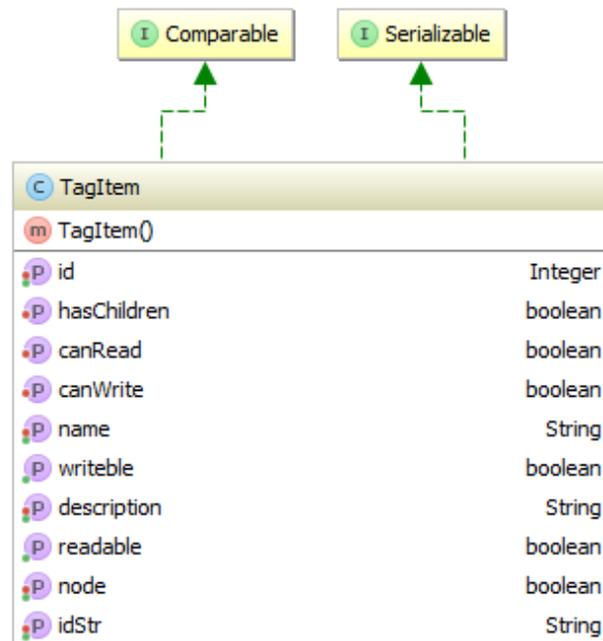


Figura 8: Classe TagItem

```

i {"idStr":"time","name":"time","children":true,"node":true,"canRead":
  true,"canWrite":false}

```

.2.3 TagInfo

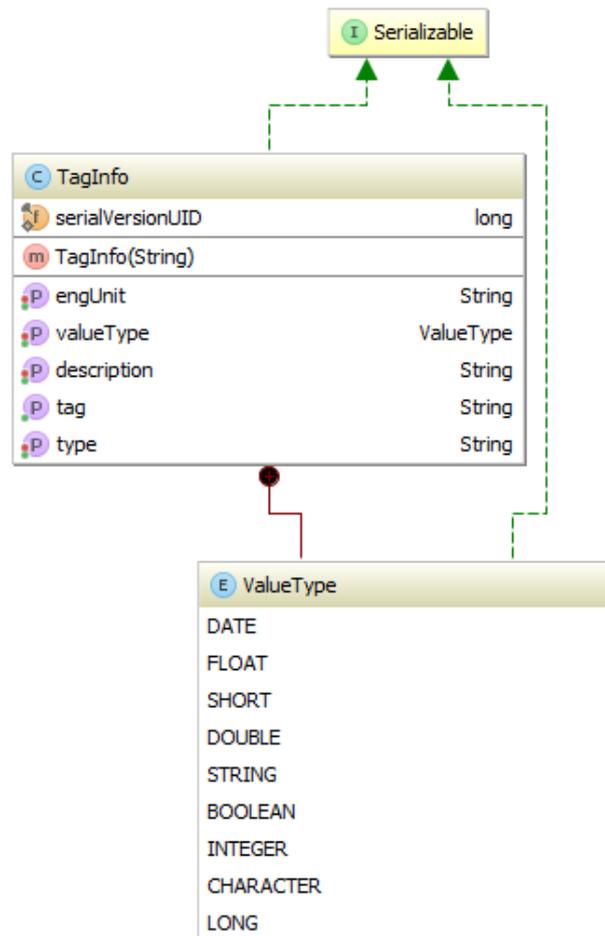


Figura 9: Classe TagInfo

```
1 {"valueType": "DOUBLE", "type": "DOUBLE", "tag": "options.sawfreq"}
```

.2.4 Callback RDA

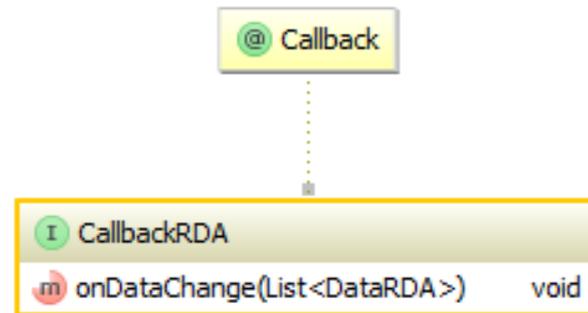


Figura 10: Classe CallbackRDA

```
1 {}
```

.2.5 ProcessVariableSubscription

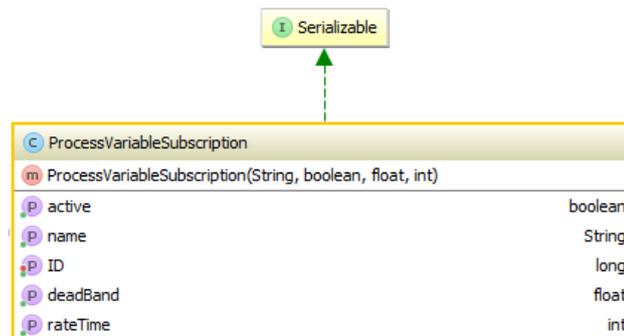


Figura 11: Classe ProcessVariableSubscription

```
1 {"id":1381412065780,"name":"SubTest","active":true,"deadBand":0.0,"
   rateTime":0}
```

.2.6 TagItemGroup

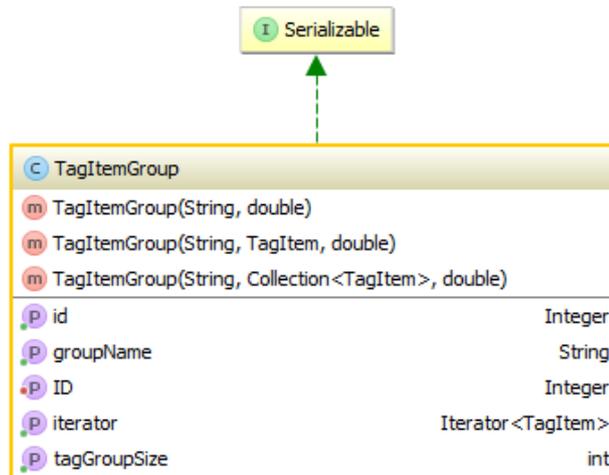


Figura 12: Classe TagItemGroup

```

{ "id":10, "passkey":0.9415610225576103, "groupName":"SyncGROUP", "
  tagList":[] }

```

.2.7 DataRDA

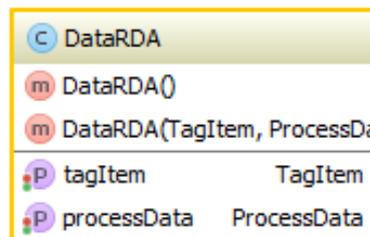


Figura 13: Classe DataRDA

```

{ "tagItem":{"idStr":"options.sawfreq", "name":"options.sawfreq", "
  children":false, "node":false, "canRead":true, "canWrite":true}, "
  processData":{"valueType":"BOOLEAN", "Value":true}}

```

.2.8 GenericDataRDA

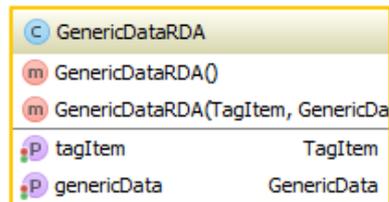


Figura 14: Classe GenericDataRDA

```
1 {"tagItem":{"idStr":"options.sawfreq","name":"options.sawfreq","
  children":false,"node":false,"canRead":true,"canWrite":true},"
  genericData":{"valueType":"BOOLEAN","floatValue":1.0,"shortValue
  ":1,"doubleValue":1.0,"stringValue":"true","booleanValue":true,"
  integerValue":1,"characterValue":84,"longValue":1}}
```

Apêndice D - Exemplo de requisições HTTP do BR-Collector

.3 Requisições e respostas via JSON-RPC

.3.1 Método listServer

```
1 {"jsonrpc":"2.0","id":1450961000,"method":"RDA.listServers","params":[]}  
2  
3 {"jsonrpc":"2.0","id":1450961000,"result":["DA"]}
```

.3.2 Método connectServer

```
1 {"jsonrpc":"2.0","id":850869730,"method":"RDA.connectServer","params":["DA"]}  
2  
3 {"jsonrpc":"2.0","id":850869730,"result":{"name":"DA","typeServer":1,"serverGroupID":1}}
```

.3.3 Método supportsHierarchialListing

```
1 {"jsonrpc":"2.0","id":131794052,"method":"RDA.supportsHierarchialListing","params":[{"name":"DA","typeServer":1,"serverGroupID":1}]}  
2  
3 {"jsonrpc":"2.0","id":131794052,"result":true}
```

.3.4 Método browseNode

```
1 {"jsonrpc":"2.0","id":69864174,"method":"RDA.browseNode","params
 ":[{"name":"DA","typeServer":1,"serverGroupID":2},{ "idStr":"time
  ","name":"time","children":true,"node":true,"canRead":true,"
  canWrite":false}]}
2
3 {"jsonrpc":"2.0","id":69864174,"result":[{"idStr":"time.current",
  "name":"current","children":false,"node":false,"canRead":true,"
  canWrite":false},{ "idStr":"time.random","name":"random","
  children":false,"node":false,"canRead":true,"canWrite":false}]}
```

.3.5 Método searchTagItems

```
1 {"jsonrpc":"2.0","id":969073877,"method":"RDA.searchTagItems",
  "params":[{"name":"DA","typeServer":1,"serverGroupID":5},
  {"textual.color"}]}
2
3 {"jsonrpc":"2.0","id":969073877,"result":[{"idStr":"textual.color",
  "name":"textual.color","children":false,"node":false,"canRead":
  true,"canWrite":false}]}
```

.3.6 Método getTagInfos

```
1 {"jsonrpc":"2.0","id":2131046541,"method":"RDA.getTagInfos",
  "params":[{"name":"DA","typeServer":1,"serverGroupID":1},
  {"idStr":"options.sawfreq","name":"options.sawfreq","children":false,"node
  ":false,"canRead":true,"canWrite":true}]}
2
3 {"jsonrpc":"2.0","id":2131046541,"result":{"valueType":"DOUBLE",
  "type":"DOUBLE","tag":"options.sawfreq"}}
```

.3.7 Método createSubscription

```
1 {"jsonrpc":"2.0","id":1831217366,"method":"RDA.createSubscription",
  "params":[{"name":"DA","typeServer":1,"serverGroupID":8},
  {"SubTest",true,100,0.1,{}]}
2
3 {"jsonrpc":"2.0","id":1831217366,"result":{"id":1381412065780,"name
  ":"SubTest","active":true,"deadBand":0.0,"rateTime":0}}
```

.3.8 Método addItemsToSubscription

```
1 {"jsonrpc":"2.0","id":766665344,"method":"RDA.addItemsToSubscription",
  "params":[{"name":"DA","typeServer":1,"serverGroupID":4},{
  "id":1381413570111,"name":"SubTest","active":true,"deadBand":0.0,"
  rateTime":0},{
  [{"idStr":"textual.color","name":"textual.color","
  children":false,"node":false,"canRead":true,"canWrite":false}]]}
2
3 {"jsonrpc":"2.0","id":766665344,"result":[true]}
```

.3.9 Método createGroup

```
1 {"jsonrpc":"2.0","id":676567405,"method":"RDA.createGroup","params":
  [{"name":"DA","typeServer":1,"serverGroupID":2},"SyncGROUP"]}
2
3 {"jsonrpc":"2.0","id":676567405,"result":{"id":10,"passkey":
  "0.9415610225576103","groupName":"SyncGROUP","tagList":[]}}
```

.3.10 Método addItemsToGroup

```
1 {"jsonrpc":"2.0","id":793649411,"method":"RDA.addItemsToGroup","
  params":[{"name":"DA","typeServer":1,"serverGroupID":2},{
  "id":10,"passkey":0.6669624703715018,"groupName":"SyncGROUP","
  tagList":[]},{
  [{"idStr":"options.sawfreq","name":"options.sawfreq",
  "children":false,"node":false,"canRead":true,"canWrite":true
  }]]}
2
3 {"jsonrpc":"2.0","id":793649411,"result":[true]}
```

.3.11 Método writeValues

```
1 {"jsonrpc":"2.0","id":627857607,"method":"RDA.writeValues","params":
  [{"name":"DA","typeServer":1,"serverGroupID":2},{
  "tagItem":{"idStr":"options.sawfreq","name":"options.sawfreq","
  children":false,"node":false,"canRead":true,"canWrite":true},
  "genericData":{"valueType":"BOOLEAN","floatValue":1.0,"shortValue":1,
  "doubleValue":1.0,"stringValue":"true","booleanValue":true,"
  integerValue":1,"characterValue":84,"longValue":1}}]}
2
3 {"jsonrpc":"2.0","id":627857607,"result":[true]}
```

.3.12 Método getCurrentValues

```
1 {"jsonrpc":"2.0","id":197672139,"method":"RDA.getCurrentValues","
   params":[{"name":"DA","typeServer":1,"serverGroupID":2},{"id
   ":10,"passkey":0.6669624703715018,"groupName":"SyncGROUP","
   tagList":[]}]}
```

2

```
3 {"jsonrpc":"2.0","id":197672139,"result":null}
```

Apêndice E - Diagrama de classes do cliente C#

.4 Diagrama de classes por pacote

.4.1 JSONRPC

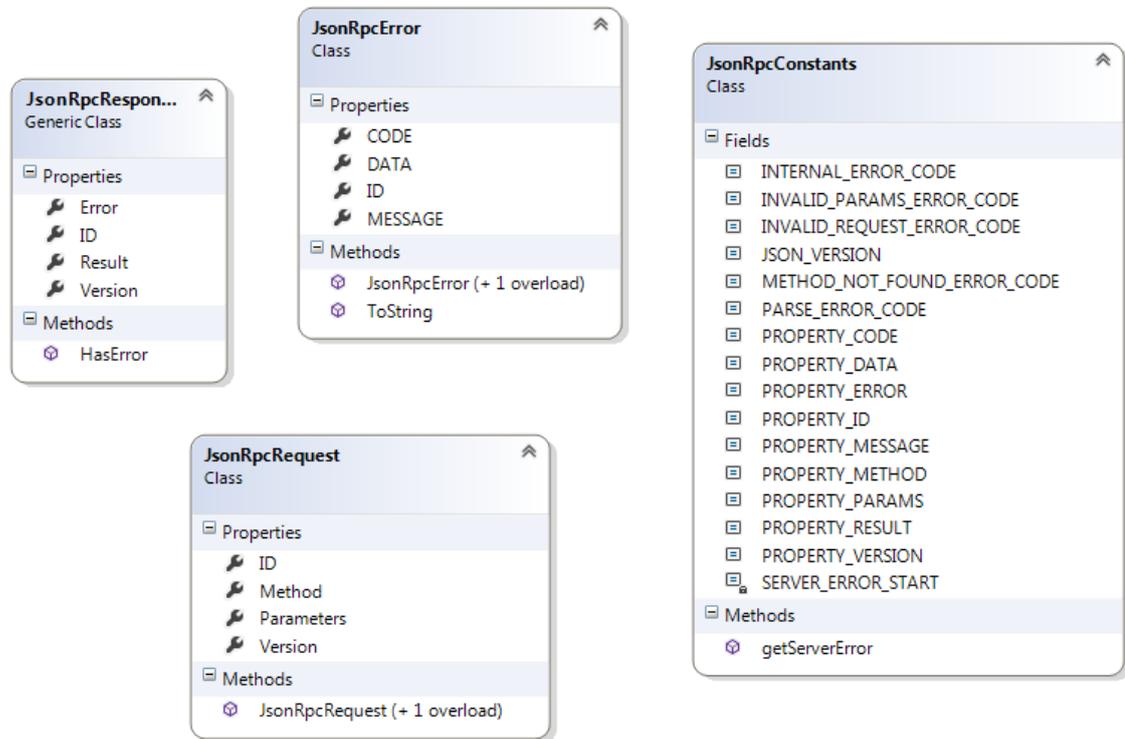


Figura 15: Classes do pacote JSONRPC.

.4.2 Util

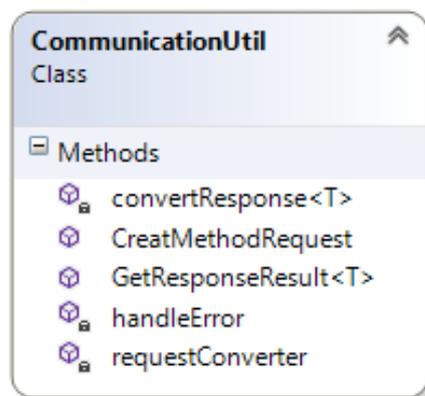


Figura 16: Classes do pacote Util.

.4.3 WebClient

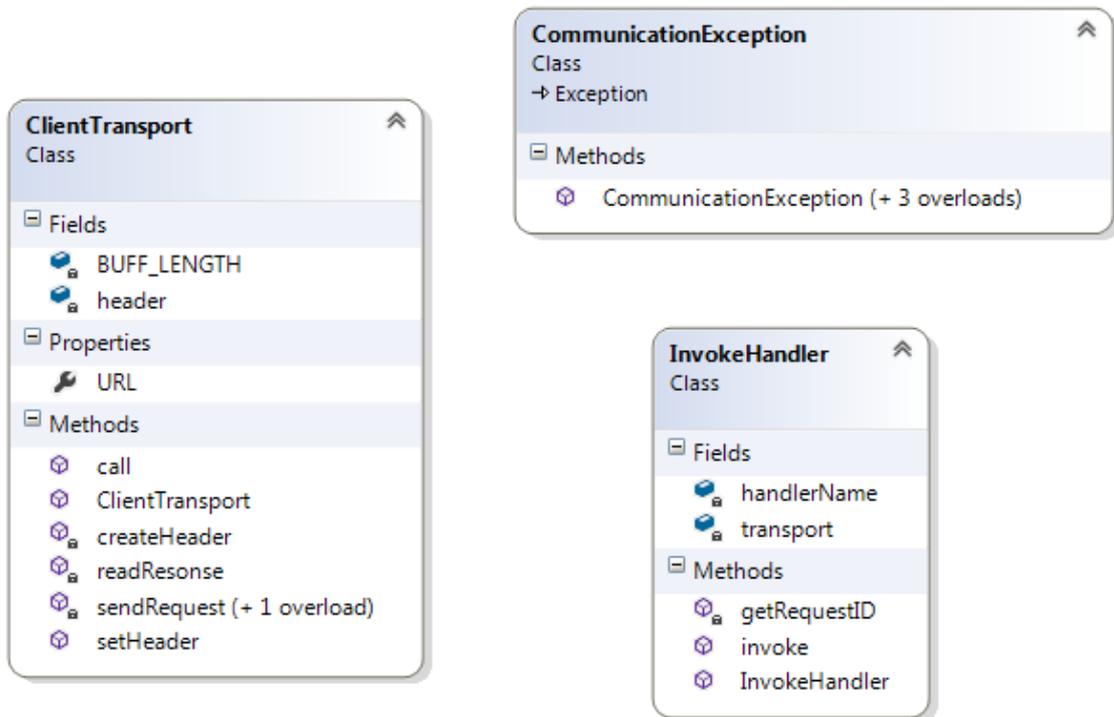


Figura 17: Classes do pacote WebClient.

Apêndice F - BR-Collector-webAPI: Cliente C#

Código 2: Métodos da API Service RDA

```
1 using BRCollector_client.Web;
2 using Newtonsoft.Json;
3 using System;
4 using BRCollector_client.JsonRpc;
5 using System.Collections.Generic;
6 using System.Linq;
7 using System.Text;
8 using BRCollector_client.Util;
9 using BRCollector_domain.domain;
10 using BRCollector_domain.Process;
11
12 namespace BRCollector_client.API
13 {
14     public class ServiceRDA
15     {
16         private InvokeHandler handler;
17
18         public ServiceRDA(InvokeHandler handler) {
19             this.handler = handler;
20         }
21
22         public List<string> ListServers()
23         {
```

```
24         string methodName = "listServers";
25
26         string result = handler.invoke(methodName, null);
27         return CommunicationUtil.GetResponseResult<List<string
28             >>(result);
29     }
30     public Server ConnectServer(string serverName)
31     {
32         string methodName = "connectServer";
33         Object[] args = { serverName };
34
35         string result = handler.invoke(methodName, args);
36         return CommunicationUtil.GetResponseResult<Server>(
37             result);
38     }
39     public bool DisconnectServer(Server server)
40     {
41         string methodName = "disconnectServer";
42         Object[] args = { server };
43
44         string result = handler.invoke(methodName, args);
45         return CommunicationUtil.GetResponseResult<bool>(result)
46             ;
47     }
48     public bool SupportsHierarchialListing(Server server)
49     {
50         string methodName = "supportsHierarchialListing";
51         Object[] args = { server };
52
53         string result = handler.invoke(methodName, args);
54         return CommunicationUtil.GetResponseResult<bool>(result)
55             ;
56     }
57     public List<TagItem> BrowseNode(Server server, TagItem node)
58     {
59         string methodName = "browseNode";
60         Object[] args = { server, node };
61
62         string result = handler.invoke(methodName, args);
63         return CommunicationUtil.GetResponseResult<List<TagItem
64             >>(result);
```

```
64     }
65
66     public List<TagItem> SearchTagItems(Server server, string
        mask)
67     {
68         string methodName = "searchTagItems";
69         Object[] args = { server, mask };
70
71         string result = handler.invoke(methodName, args);
72         return CommunicationUtil.GetResponseResult<List<TagItem
        >>(result);
73     }
74
75     public TagInfo GetTagInfos(Server server, TagItem tagItem)
76     {
77         string methodName = "getTagInfos";
78         Object[] args = { server, tagItem };
79
80         string result = handler.invoke(methodName, args);
81         return CommunicationUtil.GetResponseResult<TagInfo>(
        result);
82     }
83
84     public TagItemGroup CreateGroup(Server server, string
        groupName)
85     {
86         string methodName = "createGroup";
87         Object[] args = { server, groupName };
88
89         string result = handler.invoke(methodName, args);
90         return CommunicationUtil.GetResponseResult<TagItemGroup
        >(result);
91     }
92
93     public bool RemoveGroup(Server server, TagItemGroup group)
94     {
95         string methodName = "removeGroup";
96         Object[] args = { server, group };
97
98         string result = handler.invoke(methodName, args);
99         return CommunicationUtil.GetResponseResult<bool>(result)
        ;
100    }
101
102    public List<Boolean> AddItemsToGroup(Server server,
```

```
        TagItemGroup group, params TagItem[] tagItem)
103     {
104         string methodName = "addItemToGroup";
105         Object[] args = { server, group, tagItem };
106
107         string result = handler.invoke(methodName, args);
108         return CommunicationUtil.GetResponseResult<List<Boolean
            >>(result);
109     }
110
111     public List<Boolean> RemoveItemsFromGroup(Server server,
        TagItemGroup group, params TagItem[] tagItem)
112     {
113         string methodName = "removeItemsFromGroup";
114         Object[] args = { server, group, tagItem };
115
116         string result = handler.invoke(methodName, args);
117         return CommunicationUtil.GetResponseResult<List<Boolean
            >>(result);
118     }
119
120     public List<DataRDA> GetCurrentValues(Server server,
        TagItemGroup tagGroup)
121     {
122         string methodName = "getCurrentValues";
123         Object[] args = { server, tagGroup };
124
125         string result = handler.invoke(methodName, args);
126         return CommunicationUtil.GetResponseResult <List<DataRDA
            >>(result);
127     }
128
129     public List<Boolean> WriteValues(Server server, List<
        GenericDataRDA> tagItemsToWrite)
130     {
131         string methodName = "writeValues";
132         Object[] args = { server, tagItemsToWrite };
133
134         string result = handler.invoke(methodName, args);
135         return CommunicationUtil.GetResponseResult<List<Boolean
            >>(result);
136     }
137 }
138 }
```

REFERÊNCIAS BIBLIOGRÁFICAS

- Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad & M. Stal (1996), *Pattern-Oriented Software Architecture – A System of Patterns*, John Wiley Sons Ltd.
- Carvalho, A. S., D. B. Nascimento & R. C. Silva (2008), 'Sistema de comunicação opc para uma coluna de destilação piloto', *SEGeT – Simpósio de Excelência em Gestão e Tecnologia*.
- Carvalho, Paulo Cesar (2004), 'Sistemas de execução da manufatura - "manufacturing execution system"', *Revista Mecatrônica Atual*.
- Chetty, Damodar (2010), 'Asynchronous processing support in servlet 3.0'. (Acessado em 03 de dezembro de 2013).
URL: <http://www.softwareengineeringsolutions.com/blogs/2010/08/13/asynchronous-servlets-in-servlet-spec-3-0/>
- Chung, P. Emerald, Yennun Huang & Shalini Yajnik (1998), 'Dcom and corba side by side, step by step, and layer by layer', *C++ Report, Vol. 10, No. 1*.
- Cândido, Ronei Vilas Bôas (2004), 'Padrão opc: Uma alternativa de substituição dos drivers proprietários para acessar dados de plcs'.
- Fielding, Roy Thomas (2000), Architectural styles and the design of network-based software architectures, Dissertação de mestrado, Universidade Federal do Rio Grande do Norte.
- Fonseca, Marcos Oliveira (2002), 'Comunicação opc – uma abordagem prática', *Seminário de automação de Processos da ABM*.

Google (2013), 'Gson'. (Acessado em 25 de agosto de 2013).

URL: <http://code.google.com/p/google-gson/>

Gutierrez, Regina Maria Vinhais & Simon Shi Koo Pan (2008), 'Complexo eletrônico: Automação do controle industrial'. (Acessado em 02 de outubro de 2012).

URL: <http://www.bndes.gov.br/>

Hadlich, Thomas (2006), 'Providing device integration with opc ua', *Industrial Informatics* .

Hochgurtel, Brian (2003), *Cross-Platform Web Services Using C and Java*, Charles River Media.

JavaWorld (2008), 'Asynchronous http comet architectures'. (Acessado em 03 de dezembro de 2013).

URL: <http://www.javaworld.com/javaworld/jw-03-2008/jw-03-asynchhttp-test.html>

JavaWorld (2009), 'Asynchronous servlets in servlet spec 3.0'. (Acessado em 03 de dezembro de 2013).

URL: <http://www.javaworld.com/javaworld/jw-02-2009/jw-02-servlet3.html?page=2>

JSON-RPC (2013), 'Json-rpc specification'. (Acessado em 16 de outubro de 2013).

URL: <http://json-rpc.org/wiki/specification>

Kalin, Martin (2009), *Java Web Services: Up and Running*, O'Reilly Media, Inc.

Leitão, Gustavo Bezerra Paz (2006), 'Arquitetura e implementação de um cliente opc para aquisição de dados na indústria do petróleo'.

Leitão, Gustavo Bezerra Paz (2008), Algoritmos para análise de alarmes em processos petroquímicos, Dissertação de mestrado, Universidade Federal do Rio Grande do Norte.

Microsoft (2013), 'What is com'. (Acessado em 02 de julho de 2013).

URL: <http://www.microsoft.com/com/default.mspx>

Newtonsoft (2013), 'Codeplex - json.net'. (Acessado em 02 de novembro de 2013).

URL: <http://json.codeplex.com/>

Nogueira, Thiago Augusto (2009), Redes de comunicação para sistemas de automação industrial, Dissertação de mestrado, Universidade Federal do Rio Grande do Norte.

OMG (2013), 'Corba'. (Acessado em 15 de novembro de 2013).

URL: <http://www.corba.org>

OPC Foundation (2013). (Acessado em 02 de julho de 2013).

URL: <http://www.opcfoundation.org>

OPC UA Specification: Part 6 – Mapping. Release Candidate 0.93 (2006), Relatório técnico, OPC Foundation.

Oracle (2013a), 'Java ee - overview'. (Acessado em 20 de agosto de 2013).

URL: <http://www.oracle.com/technetwork/java/javaee/overview/index.html>

Oracle (2013b), 'Java remote method invocation'. (Acessado em 03 de dezembro de 2013).

URL: <http://docs.oracle.com/javase/6/docs/technotes/guides/rmi/index.html>

Oracle (2013c), 'Java servlet technology'. (Acessado em 03 de dezembro de 2013).

URL: <http://www.oracle.com/technetwork/java/index-jsp-135475.html>

Pudá, Adriano. P. (2008), Padronização da comunicação através da tecnologia opc, Relatório técnico, SoftBrasil Automação Ltda.

Schleipen, Miriam (2008), 'Opc ua supporting the automated engineering of production monitoring and control systems', *ETFA - Emerging Technologies and Factory Automation*.

Silva, Renan Oliveira (2013), 'Supervisório industrial para dispositivos com sistema operacional android'.

Torrise, Nunzio Marco (2010), 'Cyberopc json-rpc-da 1.0 specification'. (Acessado em 16 de outubro de 2013).

URL: <http://www.cyberopc.org>

Torrise, Nunzio Marco (2011), 'Monitoring services for industrial applications based on reverse ajax technologies', *IEEE Industrial Electronics Magazine*.

Vinoski, Steve (1997), 'Corba: Integrating diverse applications within distributed heterogeneous environments', *IONA Technologies*.

Völter, Markus, Michael Kircher & Uwe Zdun (2005), *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*, John Wiley Sons Ltd.

W3C (2012), 'Soap specifications'. (Acessado em 14 de agosto de 2013).

URL: <http://www.w3.org/TR/soap/>