Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Informática e Estatística
Bacharelado em Ciências da Computação

Alexandre de Mari

Desenvolvimento de um Servidor HTTP/XML-RPC

Florianópolis, Novembro de 2004

Alexandre de Mari

Desenvolvimento de um Servidor HTTP/XML-RPC

Trabalho de Conclusão de Curso apresentado ao Departamento de Informática e Estatística, da Universidade Federal de Santa Catarina, para a obtenção do Título de Bacharel em Ciências da Computação.

Orientador:
Conrado W. Seibel

Co-orientador:
José Eduardo De Lucca

Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Informática e Estatística

Florianópolis

Novembro de 2004

Trabalho de conclusão de curso sob o título de *"Desenvolvimento de um Servidor HTTP/XML-RPC"*, defendido por Alexandre de Mari e aprovada em 18 de Novembro de 2004, em Florianópolis, Estado de Santa Catarina, pela banca examinadora constituída por:

Conrado W. Seibel
Orientador

Prof. Dr. José Eduardo DeLucca
Co-orientador

Prof. Dr. Mário Dantas
Universidade Federal de Santa Catarina

Agradecimentos

Em especial a Ingo Shmidt, pelo apoio intelectual e de recursos no desenvolvimento deste trabalho.

Aos colegas de trabalho e de curso que auxiliaram de forma direta ou indireta na criação deste trabalho.

Aos meus pais, Nelson e Cecília, aos meus irmãos, Fernando e Marcelo, e meus amigos mais próximos, Arikson, Raphael e Telma, pelas palavras e momentos vividos no decorrer do trabalho.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho, realizado na Reason Tecnologia S.A., empresa que atua no setor elétrico, apresenta o desenvolvimento de uma solução que integra os protocolos HTTP e XML-RPC com disponibilidade de chamadas de procedimento remoto e outros recursos. Alguns dos principais objetivos são a utilização de apenas uma porta de comunicação, funcionamento no sistema operacional GNU/Linux, entre outros. Os conceitos relacionados a estes protocolos, necessários para a compreensão, os detalhes do desenvolvimento da solução, objetivos e motivação, serão apresentados.

Palavas-chave: HTTP, XML-RPC, servidor web.

Abstract

This work, realized at Reason Tecnologia S.A., company that work in the electrical sector, presents the development of a solution that integrate the HTTP and XML-RPC protocols for availability of remote procedure calls and others resources. Some of the main objectives are the use of only one communication port, functioning in the operational system GNU/Linux, among others. The related concepts with this protocols, needed for the understanding, the details of development of solution, objectives and motivation, will be presented.

Keywords: HTTP, XML-RPC, web server.

1 Introdução

Atualmente, grande parte da distribuição de conteúdo na Internet e em redes locais é realizada através do protocolo HTTP (HyperText Transfer Protocol). Este protocolo é utilizado pelos navegadores de Internet, servidores web e outros programas para obterem ou disponibilizarem recursos estáticos ou dinâmicos na Internet. Um desses recursos é a chamada de procedimento remota que utiliza o protocolo XML-RPC (eXtensible Markup Langange - Remote Procedure Call) sobre o protocolo HTTP.

A Reason Tecnologia S.A.[1] passou a desenvolver os programas de interface com o usuário, de uma linha de produtos, baseados nos protocolos HTTP e XML-RPC. Nesses produtos, um servidor HTTP e um servidor XML-RPC são utilizados para fornecer essa interface. Porém, estes programas apresentam problemas, como: grande volume de bibiotecas externas (o servidor XML-RPC utiliza muitas bibliotecas do W3C), a não integração entre o servidor HTTP e o servidor XML-RPC, defeitos no servidor XML-RPC, interface do servidor XML-RPC de difícil utilização, entre outros.

Este trabalho busca propor o desenvolvimento de um programa servidor utilizando os protocolos HTTP e XML-RPC e atenda as principais demandas da organização, que são:

- Utilização de apenas uma porta de comunicação;
- Baixo consumo de memória e processamento;
- API (Application Program Interface) mais fácil e extensível (XML-RPC);
- Desenvolvimento na linguagem ANSI C;

[1] A Reason Tecnologia S.A é uma empresa sediada em Florianópolis, no Estado de Santa Catarina, que atua no setor elétrico nas áreas de oscilografia, qualidade de enegia e sincronismo temporal. Seu portfólio de produtos inclui equipamentos de aquisição de dados com interface de condicionamento e tratamento de sinais, equipamento de sincronismo baseado em GPS, software integrado de comunicação, configuração e análise gráfica e soluções para gerenciamento de comunicação entre equipamentos RDPs (Registradores Digitais de Perturbações) e relés de proteção de diversos fabricantes em uma plataforma de gestão de dados baseado em Intranet.

- Funcionalidades mínimas do protocolo HTTP;
- Reduzir a dependência de bibliotecas externas;
- Funcionamento no sistema operacional GNU/Linux[2];
- Simplicidade, fácil manutenção e;
- Domínio da tecnologia.

[2]O projeto GNU foi lançado em 1984 para desenvolver um sistema operacional completo e livre, similar ao Unix: o sistema GNU (GNU é um acrônimo recursivo que se pronuncia "guh-NEW"ou "guniw"). Variações do sistema GNU, que utilizam o núcleo Linux, são hoje largamente utilizadas; apesar desses sistemas serem normalmente chamados de Linux, eles são mais precisamente chamados Sistemas GNU/Linux.

2 Fundamentação Teórica

2.1 Protocolo HTTP

O HTTP (Hyper Text Transfer Protocol) é o protocolo usado para a comunicação na *World Wide Web*. Há muitas aplicações do HTTP, mas ele é mais famoso por possibilitar a conversação entre os navegadores *web* e servidores *web*. Este protocolo possibilita a visualização de documentos hipermídia, além de outros recursos disponíveis na *web*. Estes recursos são movimentados pelo protocolo HTTP rapidamente, convenientemente, e de maneira confiável de todos os servidores *web* ao redor do mundo para os navegadores *web* nos computadores das pessoas [GOURLEY & TOTTY, 2002].

O conteúdo da *web* vive nos servidores *web*. Servidores *web*, falam o protocolo HTTP, e por isso eles são frequentemente chamados de servidores HTTP. Estes servidores HTTP armazenam as informações da Internet e fornecem essas informações quando são requisitadas pelos clientes HTTP. Os clientes enviam requisições HTTP para os servidores, e os servidores retornam a informação requisitada em respostas HTTP. Juntos, clientes e servidores HTTP formam os componentes básicos da *World Wide Web* [GOURLEY & TOTTY, 2002].

Segundo o RFC[3] 2616 de 1999, o HTTP é um protocolo do nível de aplicação para distribuir informações hipermídia, e que vem sendo usado desde 1990. Sua primeira versão, HTTP/0.9, era um simples protocolo para transferir dados brutos através da Internet. Com o passar do tempo foi ganhando complexidade, os sistemas de informação passaram a requerer mais funcionalidades do que uma recuperação simples de dados.

Na versão atual, HTTP/1.1, é permitido um conjunto sem limites de métodos e cabeçalhos que indicam o propósito da requisição (comandos ou métodos que indicam as ações a serem tomadas pelo servidor ou pelo requisitor), e inclui exigências mais ri-

[3]Nome dado ao resultado e processo de criação de um padrão da Internet. Novos padrões são propostos e publicados *on-line*, como um *Request For Comments* (RFC). O *Internet Engineering Task Force* (IETF) é quem regula a discussão, e eventualmente um novo padrão é estabelecido.

gorosas que as versões anteriores, a fim de assegurar a confiabilidade da implementação de suas características [RFC 2616, 1999].

2.1.1 Definições

Os termos abaixo são utilizados no decorrer deste trabalho e fazem parte da especificação do protocolo HTTP:

Conexão ou Connection

Um circuito virtual da camada de transporte estabelecido entre dois programas a fim de se comunicar.

Mensagem ou Message

A unidade básica de comunicação HTTP, consistindo de uma seqüência estruturada de octetos transmitidos via conexão.

Requisição ou Request

Uma mensagem de requisição HTTP.

Resposta ou Response

Uma mensagem de resposta HTTP.

Recurso ou Resource

Um objeto de dados da rede ou serviço que pode ser identificado por uma URL (Uniform Resource Locator - Localizador de Recurso Uniforme). Recursos podem estar disponíveis em várias representações (por exemplo, múltiplas linguagens, formatos de dados, tamanho, resolução, etc.) ou várias outras formas.

Cliente ou Client

Um programa que estabelece conexões com o propósito de enviar requisições.

Entidade ou Entity

A informação transmitida na carga de uma requisição ou resposta. Uma entidade consiste de meta-informação na forma de campos de cabeçalho-de-entidade e conteúdo na forma de corpo-entidade.

Servidor ou Server

Um programa aplicativo que aceita conexões para receber requisições enviando de volta respostas. Quando se refere a servidor, está se referindo ao papel desempenhado e não a suas capacidades, tendo em vista que um dado programa pode ser capaz de ser ao mesmo tempo cliente e servidor. Além disso, qualquer servidor pode agir como um servidor original, proxy, gateway, ou túnel, tendo o comportamento determinado apenas pela natureza de cada requsição.

2.1.2 Operações

O protocolo HTTP é um protocolo de requisição e resposta. Um cliente envia uma requisição ao servidor na forma de um método de requisição, URL, e versão do protocolo, seguido por uma mensagem MIME-like (Multipurpose Internet Mail Extensions) contendo os modificadores da requisição, informações do cliente e um possível corpo com o conteúdo sobre a conexão com um servidor. O servidor responde com uma linha de status incluindo a mensagem com a versão do protocolo e o código de sucesso ou erro, seguido por uma mensagem MIME-like contendo a informação do servidor, meta-informação da entidade e possivelmente o conteúdo do corpo-entidade [RFC 2616, 1999].

2.1.3 A Mensagem HTTP

As mensagens HTTP são os blocos de dados enviados entre aplicações HTTP. Estes blocos de dados iniciam com alguma meta-informação descrevendo o conteúdo e significado da mensagem, seguido por dado opcional. Cada mensagem contém uma requisição do cliente ou uma resposta do servidor. Elas consistem de três partes: uma linha inicial (start-line) descrevendo a mensagem, um bloco de cabeçalhos (headers) contendo atributos, e um corpo (body) opcional contendo dados [GOURLEY & TOTTY, 2002].

Mensagens de requisição requisitam uma ação de um servidor. Mensagens de resposta carregam resultados de uma requisição de volta ao cliente. Ambas, resposta e requisição, têm a mesma estrutura básica de mensagem [GOURLEY & TOTTY, 2002].

O formato de uma mensagem de requisição é descrito a seguir:

```
<method> <request-URL> <version>
<headers>
```

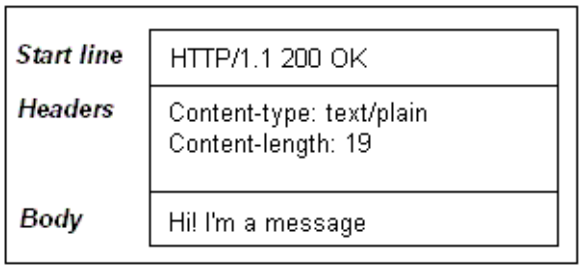

Figura 1: Exemplo de uma mensagem HTTP [GOURLEY & TOTTY, 2002]

```
<entity-body>
```

O formato de uma mensagem de resposta é descrito a seguir:

```
<version> <status> <reason-phrase>
<headers>

<entity-body>
```

Uma breve descrição das várias partes das mensagens de acordo com GOURLEY & TOTTY (2002):

method

A ação que o cliente quer que o servidor execute no recurso.

request-URL

Uma URL completa nomeando o recurso requisitado.

version

A versão do HTTP que a mensagem esta usando.

status-code

Um número de três dígitos descrevendo o que aconteceu durante a requisição.

reason-phrase

Uma versão legível à humanos do código de status numérico.

headers

Zero ou mais cabeçalhos, que é um nome, seguido por dois pontos (:), seguido por espaço em branco opcional, seguido por um valor, seguido por CRLF (Carriage Return Line Feed).

entity-body

O corpo-entidade contém um bloco de dado arbitrário. Nem todas as mensagens contêm corpos de entidade.

As mensagens de requisição pedem ao servidor para fazer alguma coisa com um recurso. A linha inicial da mensagem de requisição, ou linha de requisição, contém um método descrevendo qual operação o servidor deve executar, uma URL descrevendo o recurso no qual executar o método e também a versão HTTP [GOURLEY & TOTTY, 2002].

As mensagens de resposta carregam informação de status e qualquer dado resultante de uma operação de volta para um cliente. A linha inicial de uma resposta, ou linha de resposta (response line), contém a versão HTTP que a mensagem de resposta está usando, um código de status numérico, a uma frase de razão textual descrevendo o status da operação [GOURLEY & TOTTY, 2002].

A especificação do HTTP define um conjunto de métodos de requisição comuns e que são mostrados na Tabela 1.

Método	Descrição	Corpo da mensagem?
GET	Obtém um documento do servidor.	Não
HEAD	Obtém os cabeçalhos de um documento do servidor.	Não
POST	Envia dados para o servidor processar.	Sim
PUT	Armazena o corpo de uma requisição no servidor.	Sim
TRACE	Rastreia a mensagem do servidor Proxy até o servidor.	Não
OPTIONS	Determina quais métodos podem operar no servidor.	Não
DELETE	Remove um documento do servidor.	Não

Tabela 1: Métodos HTTP comuns

Assim como os métodos dizem ao servidor o que fazer, códigos de status dizem ao cliente o que aconteceu. Códigos de status estão presentes nas linhas iniciais das respostas.

Intervalo total	Intervalo definido	Categoria
100-199	100-101	Informações
200-299	200-206	Sucesso
300-399	300-305	Redirecionamento
400-499	400-415	Erro no cliente
500-599	500-505	Erro no servidor

Tabela 2: Classe dos códigos de status

Os diferentes códigos de status são agrupados em classes pelos seus códigos numéricos de três dígitos como descrito na Tabela 2.

Os campos de cabeçalho HTTP fornecem informação adicional para mensagens de requisição e resposta. Eles são basicamente listas de pares nome/valor. Por exemplo, a seguinte linha de cabeçalho atribui o valor 19 para o campo do cabeçalho Content-legth [GOURLEY & TOTTY, 2002]:

Content-length: 19

A especificação HTTP define vários campos de cabeçalhos. As aplicações são livres para inventar seus próprios cabeçalhos. Os cabeçalhos HTTP são classificados em:

General headers

Podem aparecer em ambas mensagens de requisição de resposta.

Request headers

Fornecem mais informações sobre a requisição.

Response headers

Fornecem mais informações sobre a resposta.

Entity headers

Descreve tamanho e conteúdo, ou o próprio recurso.

Extension headers

Novos cabeçalhos que não são definidos na especificação.

Exemplo de cabeçalho	Descrição
Date: Sun, 10 Oct 2004 22:22:22 GMT	A data da resposta gerada pelo servidor
Content-length: 456	O corpo-entidade contém 456 bytes
Content-type: image/jpeg	O corpo-entidade é uma imagem GIF
Accept: image/png, image/jpeg, text/html	O cliente aceita imagens GIF e JPEG e HTML

Tabela 3: Exemplos de cabeçalhos comuns

A terceira parte de uma mensagem HTTP é o opcional corpo-entidade. Os corpos-entidade são as informações que o HTTP foi designado para transportar.

2.1.4 Conexão HTTP

Os dados do protocolo HTTP são transportados sobre o protocolo TCP/IP, que é um dos mais populares protocolos de rede para troca de pacotes utilizado por computadores e dispositivos de redes ao redor do globo. Uma aplicação cliente pode abrir uma conexão TCP/IP para uma aplicação servidor, rodando em qualquer parte do mundo. Uma vez a conexão estabelecida, mensagens trocadas entre o computador cliente e o computador servidor dificilmente são perdidas, danificadas, ou recebidas fora de ordem.

Segundo GOURLEY & TOTTY (2002), as formas mais utilizadas para gerenciar as conexões HTTP são:

- Serial;
- Paralelas;
- Persistentes e;

Em conexões do tipo Serial apenas uma conexão é aceita e o sistema somente aceitará uma nova conexão após o término da conexão corrente.

Nas conexões paralelas é permitido ao cliente abrir múltiplas conexões e realizar múltiplas transações em paralelo, tornando, na maioria das vezes, mais rápidas essas conexões [GOURLEY & TOTTY, 2002].

Conexões paralelas podem aumentar a velocidade de transferência de páginas compostas. Porém, possui algumas desvantagens como [GOURLEY & TOTTY, 2002]:

- Cada transação abre/fecha novas conexões, custando tempo e largura de banda;
- Cada nova conexão tem a performance reduzida por causa do início lento do TCP;
- Existe um limite prático sobre o número de conexões paralelas abertas.

Conexões persistentes reusam a conexão TCP preexistente para futuras requisições HTTP. Esse reuso pode evitar o lento procedimento de conexão. Adicionalmente, a conexão já aberta pode evitar o início lento da fase de adaptação ao congestionamento, permitindo transferência de dados mais rápidas [GOURLEY & TOTTY, 2002].

As conexões persistentes oferecem algumas vantagens sobre as conexões paralelas. Elas reduzem o atraso e a sobrecarga no estabelecimento da conexão, mantendo as conexões

em um estado modulado, e reduz potencialmente a quantidade de conexões abertas. Além disso, as conexões persistentes podem ser mais eficientes quando usadas em conjunto com conexões paralelas [GOURLEY & TOTTY, 2002].

Há dois tipos de conexões persistentes: a antiga conexão HTTP/1.0+ keep-alive e a moderna HTTP/1.1.

Na versão HTTP/1.0+ para os clientes manterem a conexão persistente precisavam utilizar o cabeçalho de requisição Connection: Keep-Alive. Se o servidor concorda em manter a conexão para a próxima requisição, ele deve responder com o mesmo cabeçalho na resposta. Se não há o cabeçalho Connection: keep-alive na resposta, o cliente assume que o servidor não suporta a persistência e que o servidor fechará a conexão quando for enviada a resposta. Clientes e servidores não precisam concordar com a sessão persistente se ela é requisitada [GOURLEY & TOTTY, 2002].

2.2 XML

O desenvolvimento de XML (*Extensible Markup Language*) começou em 1996 e é uma Recomendação W3C (*Word Wide Web Consortium*) desde fevereiro de 1998. Na verdade, esta tecnologia não é muito recente. Antes de XML já existia SGML, desenvolvida no início da década de 1980 e padrão ISO desde 1986, largamente utilizada em grandes projetos de documentação. Os projetistas da XML simplesmente pegaram as melhores partes da SGML, guiados pela experiência acumulada com HTML, e produziram algo que não é em nada menos poderoso que SGML, e amplamente mais regular e simples de usar [W3C Recommmendation, 1999].

XML é um conjunto de regras para projetar formatos de texto que permite estruturar dados. XML não é uma linguagem de programação e não é necessário ser um programador para usá-la ou aprendê-la. XML torna simples para o computador gerar e ler dados, e garantir que sua estrutura não seja ambígua. XML evita os problemas mais comuns em projetos de linguagens: ela é extensível, independente de plataforma e suporta internacionalização e localização [W3C Recommmendation, 1999].

Como HTML, XML usa marcadores (palavras envoltas pelos sinais '<' e '>') e atributos (na forma `nome="valor"`). Mas enquanto HTML especifica o que cada marcador e atributo significa, e às vezes como seu conteúdo aparecerá num navegador, XML usa os marcadores apenas para delimitar os trechos de dados, deixando sua interpretação completamente à cargo da aplicação que os lê. Em outras palavras, ao ver "`<p>`"num

arquivo XML, não assuma que é um parágrafo [W3C Recommmendation, 1999].

XML permite que se defina um novo formato de documento combinando ou reutilizando outros formatos. Como dois formatos desenvolvidos independentemente podem ter elementos ou atributos homônimos, deve-se ter cuidado ao combinar tais formatos [W3C Recommmendation, 1999].

2.3 RPC

RPC (*Remote Procedure Call*) é uma poderosa técnica para construção de aplicações distribuídas baseadas em cliente-servidor. Ela é uma extensão da noção convencional, ou de chamada a procedimento local, de modo que o procedimento chamado não precisa existir no mesmo espaço de endereço que o procedimento chamador. Os dois processos podem estar no mesmo sistema, ou eles podem estar em diferentes sistemas com uma conexão de rede entre eles [MARSHALL, 1999]. Isso aumeta a interoperabilidade, portabilidade e flexibilidade de uma aplicação permitindo que ela seja distribuída sobre múltiplas plataformas heterogêneas [VONDRAK, 1997].

Usando o RPC, os programadores de aplicações distribuídas evitam os detalhes da relação com a rede. A independência do transporte do RPC isola a aplicação dos elementos físicos e lógicos do mecanismo de transmissão de dados e permite que a aplicação use uma variedade de transportes [MARSHALL, 1999].

Uma RPC é analoga a uma chamada a função. Como uma chamada a função, quando uma RPC é realizada os argumentos de chamada são passados para o procedimento remoto e o chamador aguarda a resposta ser retornada do procedimento remoto. A Figura 2 mostra o fluxo da atividade que ocorre durante uma chamada RPC entre dois sistemas conectados. O cliente faz uma chamada a procedimento que envia uma requisição para o servidor e aguarda. A linha de execução é bloqueada até que uma resposta seja recebida, ou o tempo de espera termine. Quando a requisição chega, o servidor chama uma rotina que executa o serviço pedido, e envia a resposta ao cliente. Depois que a chamada é completada, o programa cliente continua [MARSHALL, 1999].

2.4 Protocolo XML-RPC

O XML-RPC (*Extensible Markup Language-Remote Procedure Call*) é um protocolo, desenvolvido por Dave Winer, que permite a um software rodando em um sistema opera-

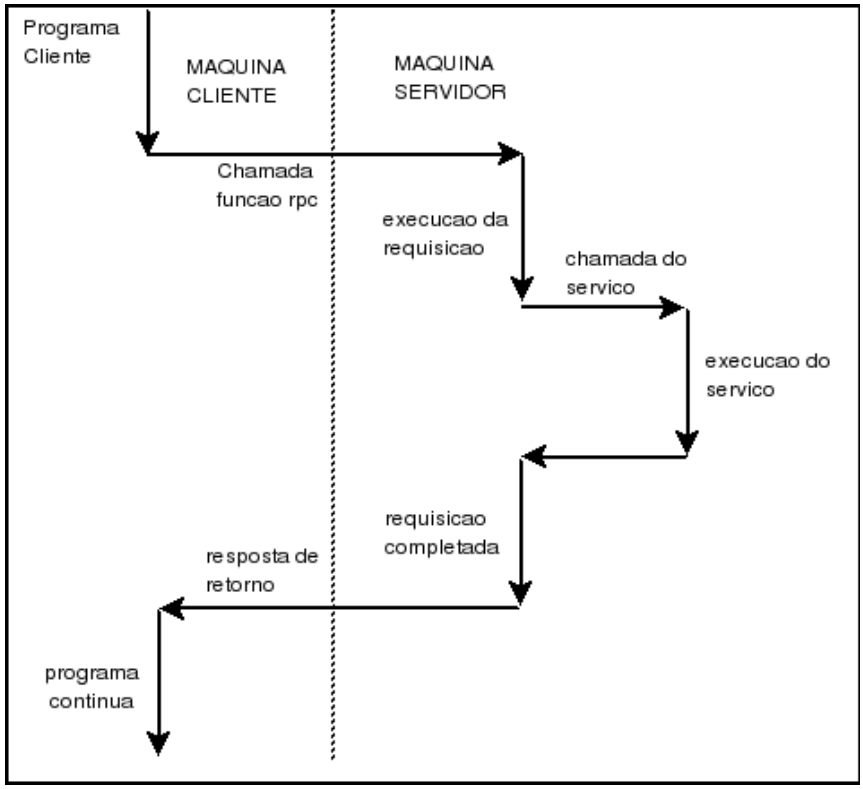

Figura 2: Mecanismo de chamada a procedimento remoto

cional qualquer, em diferentes ambientes fazer chamadas a procedimentos sobre a Internet. Estas chamadas são realizadas usando como transporte o protocolo HTTP e a linguagem de marcação XML para a codificação, como pode-se observar na Figura 3. O protocolo XML-RPC é projetado para ser tão simples quanto possível, enquanto permite a transmissão, processamento e retorno de estruturadas de dados complexas. Ele é de fácil implementação, mas possui algumas limitações.

Uma mensagem XML-RPC é uma requisição HTTP do método POST. O corpo da requisição é em XML. Um procedimento é executado no servidor e o valor retornado também é formatado em XML [WINER, 1999].

Os parâmetros do procedimento podem ser números, strings, datas, etc. e também podem ser registros complexos e listas de estruturas.

2.4.1 Requisição XML-RPC

Um exemplo de requisição XML-RPC é mostrado a seguir.

Exemplo 1 - Mensagem de requisição XML-RPC:

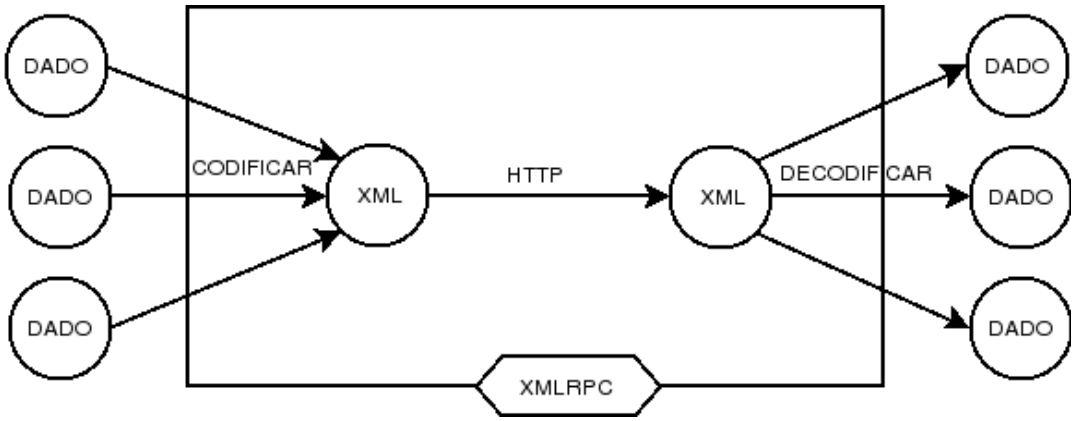

Figura 3: Prococolo XML-RPC [WINER, 1999]

```
POST /RPC2 HTTP/1.0
User-Agent:  Squirrel 1.0
Host:  reason.com.br
Content-Type:  text/xml
Content-length:  174

<?xml version="1.0"?>
<methodCall>
   <methodName>examples.getId</methodName>
      <params>
       <param>
         <value><i4>33</i4></value>
       </param>
      </params>
</methodCall>
```

O formato da URL na primeira linha do cabeçalho não é especificado. Por exemplo, ela pode ser vazia, uma simples barra invertida (slash), se o servidor é somente para chamadas XML-RPC. Entretanto, se o servidor também aceita chamadas HTTP, a URL deve ajudar a direcionar a requisição para o código que aceita XML-RPC [WINER, 1999]. (No exemplo, a URL é /RPC2.)

Um User-Agent e Host devem ser especificados. O cabeçalho Content-Type deve ser

text/xml, e o Content-length deve ser especificado e deve estar correto.

A informação está em XML, em uma única estrutura `<methodCall>`. A `<methodCall>` deve conter um sub-item `<methodName>`, uma string, contendo o nome do método a ser chamado. A string pode ser identificada somente por caracteres, maiúsculos e minúsculos, caracteres numéricos, 0-9, sublinhado, ponto, dois pontos e barra invertida (XML-RPC Specification).

Se o procedimento chamado tem parâmetros, a `<methodCall>` deve conter um sub-item `<params>`. O sub-item `<params>` pode conter qualquer quantidade do sub-item `<param>`, cada um dos quais com um sub-item `<value>` [WINER, 1999].

Os sub-itens `<value>` podem ser escalares, cujo tipo é indicado dentro de `<value>` por um dos sub-itens listados na Tabela 4.

Item	Tipo	Exemplo
`<i4> ou <int>`	Inteiro sinalizado de 4 bytes	-12
`<boolean>`	0 (falso) ou 1 (verdadeiro)	1
`<string>`	string	Teste
`<double>`	Número em ponto flutuante	-12.144
`<dateTime.iso8601>`	date/time	20040610T14:47:55
`<base64>`	Código binário base64	eW61IGNhbid8IHJlYWQgdGhpcyA=

Tabela 4: Tipos permitidos para o sub-item `<value>`

Se nenhum tipo é indicado, o tipo é string.

O `<value>` também pode ter sub-itens do tipo `<struct>`. Um `<struct>` contém sub-itens `<member>` e cada `<member>` contém um sub-item `<name>` e um `<value>` [WINER, 1999].

Exemplo 2 - Formato da informação para elementos `<struct>`:

```
<struct>
  <member>
    <name>nome1</name>
    <value><i4>12345</i4></value>
  </member>
  <member>
    <name>nome2</name>
    <value><i4>98765</i4></value>
  </member>
```

```
</struct>
```

Um <struct> pode ser recursivo, qualquer <value> pode conter um <struct> ou qualquer outro tipo, incluindo um <array>, descrito a seguir.

Outro tipo permitido em <value> é o tipo <array>. Um <array> contém um único elemento <data>, o qual pode conter qualquer quantidade de <value>. Elementos <array> não tem nomes. Pode-se misturar tipos como o exemplo seguinte mostra [WINER, 1999].

Exemplo 3 - Formato da informação para elementos <array> [WINER, 1999]:

```
<array>
  <data>
    <value><i4>-99</i4></value>
    <value><string>Brasil</string></value>
    <value><boolean>1</boolean></value>
    <value><i4>100</i4></value>
  </data>
</array>
```

Um <array> pode ser recursivo, qualquer <value> pode conter um <array> ou qualquer outro tipo, incluindo um <struct> descrito anteriormente.

2.4.2 Resposta XML-RPC

Um exemplo de resposta XML-RPC é mostrado a seguir.

Exemplo 4 - Resposta XML-RPC [WINER, 1999]:

```
HTTP/1.1 200 OK
Connection:  close
Content-Length:  160
Content-Type:  text/xml
Date:  Sun, 10 Oct 2004 22:17:50 GMT
Server:  Squirrel 1.0

<?xml version="1.0"?>
```

```
<methodResponse>
   <params>
      <param>
       <value><string>América do Sul</string></value>
      </param>
   </params>
</methodResponse>
```

Uma resposta ao método POST de uma requisição XML-RPC sempre retorna um código de status HTTP 200 OK, a menos que ocorra algum erro interno no servidor, ou seja, um erro HTTP.

O cabeçalho Content-type é text/xml e o Content-length deve estar presente e correto. O corpo da resposta é em XML e é formado pelo elemento `<methodResponse>`, que pode conter um único `<params>`, que pode conter uma estrutura `<params>` ou uma estrutura `<fault>` [WINER, 1999].

Exemplo 5 - Resposta XML-RPC informando erro:

```
HTTP/1.1 200 OK
Connection:  close
Content-Length:  424
Content-Type:  text/xml
Date:  Sun, 10 Oct 2004 22:17:50 GMT
Server:  Squirrel 1.0

<?xml version="1.0"?>
<methodResponse>
   <fault>
      <value>
       <struct>
         <member>
           <name>faultCode</name>
           <value><int>1</int></value>
         </member>
         <member>
           <name>faultString</name>
           <value><string>Muitos parâmetros.</string></value>
```

```
          </member>
        </struct>
       </value>
    </fault>
</methodResponse>
```

A estrutura `<fault>`, exibida no Exemplo 5, é utilizada para informar erros de processamento na função chamada. Contém um `<value>` que é um `<struct>` contendo dois elementos, um chamado `<faulCode>`, que contém o código do erro e, um chamado de `<faultString>`, com a descrição do código. O elemento `<faultCode>` não possui formalização e, portanto, pode conter qualquer valor, cabendo ao projetista do sistema determinar o que significa este valor [WINER, 1999].

A estrutura `<params>` é utilizada na resposta correta da função chamada.

O elemento `<methodResponse>` não pode conter ambos `<fault>` e `<params>`.

3 Descrição do Projeto

3.1 Requisitos, Restrições e Descrição do Ambiente

O desenvolvimento do servidor HTTP/XML-RPC foi realizado no sistema operacional GNU/Linux From Scratch (LFS - projeto GNU para se gerar distribuições Linux personalizadas) e utilizou-se as seguintes ferramentas, linguagens e scripts:

- Linguagem ANSI C;
- GNU bash, version 2.05a.0(1)-release (i386-pc-linux-gnu)
- GNU Make 3.80;
- GNU gcc (GCC) 3.3.3;
- GNU Emacs;
- txt2tags version 2.0;
- Concurrent Versions System (CVS) 1.11.1p1;
- Analisador XML GNU/Mini-XML versão 2.0;

De acordo com os estudos realizados os seguintes requisitos gerais foram contemplados:

- Utilização do protocolo HTTP/1.1

  É a versão mais utilizada e documentada do protocolo HTTP.

- Utilização do protocolo XML-RPC;

  A utilização do protocolo XML-RPC deveu-se a sua simplicidade, a necessidade de disponibilizar apenas funcionalidades e não objetos, e as restrições de segurança do sistema em questão são menores. Além disso, a facilidade de implementação de clientes XML-RPC também influenciou na escolha.

- Modularidade

  A modularidade é necessária para facilitar o desenvolvimento e manutenção do sistema.

- Funcionamento através de Daemons

  Através de daemons o sistema pode ser executado em segundo plano no sistema operacional.

- Utilização de uma porta de comunicação apenas

  Restrições são criadas pelas organizações quando da necessidade de utilização de mais de uma porta de comunicação e que sejam desconhecidas.

- Conexões limitadas e persistentes

  As conexões são limitadas para evitar a sobrecarga de processamento. A persistência é necessária para evitar a troca de mensagens TCP na criação de conexões.

- Requisições multiprocessadas;

  Para aumentar o desempenho do sistema, a cada nova conexão aceita uma nova linha de execução é criada.

- Biblioteca para composição e análise dos dados XML-RPC

  As funções de análise e composição dos dados XML-RPC são fornecidas através de uma bilioteca estática.

- Interface do servidor XML-RPC de fácil utilização

  Para facilitar a criação das funções que serão disponibilizadas pelo usuário.

- Configuração através de arquivo texto.

  O arquivo texto é simples de editar, analisar e manter.

3.2 Estrutura Modular

O sistema foi desenvolvido de forma modular, tendo como módulos principais o Servidor HTTP e o Servidor XML-RPC exibidos na Figura 4. Além desses módulos principais, outros foram projetados.

Os módulos que compõem o sistema e que têm funções específicas à desempenhar são os seguintes:

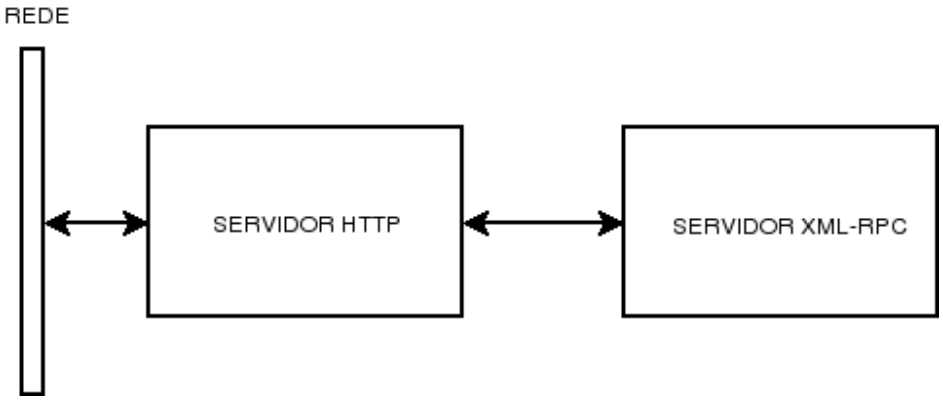

Figura 4: Módulos principais do sistema.

- Módulo Common: funções comuns do sistema.
- Módulo Config: leitura e análise do arquivo de configuração do sistema.
- Módulo Connection: construção e gerenciamente de conexões.
- Módulo HTTP: processamento de requisições e construção de respostas do protocolo HTTP.
- Módulo Log: cria e escreve as mensagens de log em um arquivo
- Módulo XMLRPC: processamento de requisições e construção de respostas do protocolo XMLRPC.

Na Figura 5 pode-se verificar as depêndencias entre os módulos.

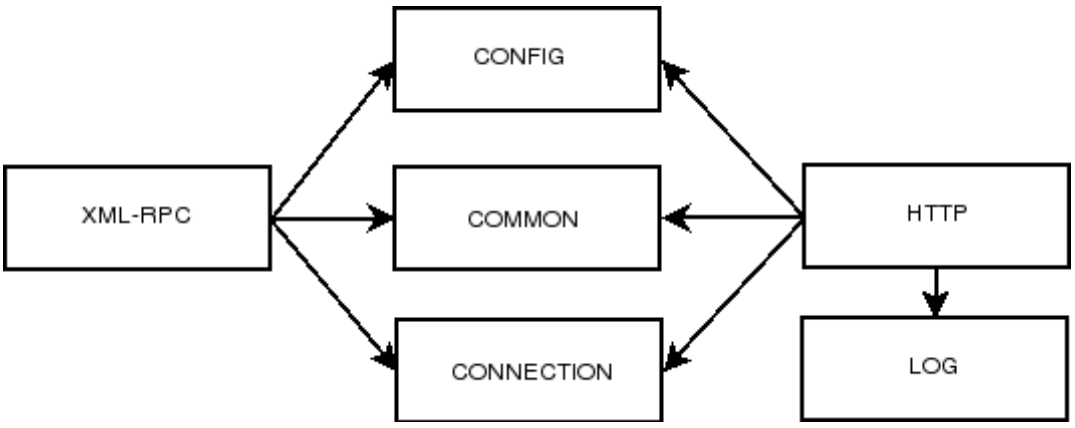

Figura 5: Dependências entre os módulos do sistema.

3.3 Servidor HTTP

Nesta seção são descritas as etapas de projeto do servidor HTTP. Essas etapas se basearam nas características básicas necessárias para que um servidor HTTP tenha um mínimo de funcionalidades de forma a operar adequadamente.

De acordo com GOURLEY & TOTTY (2002), um servidor web (web server) HTTP processa requisições e gera respostas. Servidores web implementam o protocolo HTTP, gerenciamento de recursos web e fornecem características administrativas para configurar e controlá-los. O servidor web compartilha a responsabilidade de gerenciar as conexões TCP com o sistema operacional.

3.3.1 Descrição Funcional

O servidor deve estabelecer uma conexão (TCP/IP) para cada cliente de forma simples e segura. A partir do estabelecimento da conexão o servidor deve estar apto a receber mensagens de requisição do protocolo HTTP. Essas mensagens devem ser processadas de modo a identificar qual ação deve ser executada. Após esta identificação deve-se processar a ação e construir a resposta, de acordo com a especificação do protocolo HTTP, que será enviada ao cliente que fez a requisição. Se a requisição é referente à chamada de método remota o servidor deve, de algum modo, invocar o método, obter o resultado, e construir a resposta de acordo com os protocolos HTTP e XML-RPC.

Segundo GOURLEY & TOTTY (2002), a seqüência de funcionamento de um servidor web é a seguinte:

1. Aceitar conexões de clientes.
2. Receber mensagens de requisição.
3. Processar requisições.
4. Mapear e acessar os recursos.
5. Construir as respostas.
6. Enviar as respostas.
7. Fazer o log.

1. Aceitar conexões de clientes.

   Verificar se o cliente já possui uma conexão aberta no servidor que poderá ser usada para enviar sua requisição. Caso contrário, deve-se abrir uma nova conexão para o servidor.

2. Receber mensagens de requisição.

   Como os dados chegam através das conexões, o servidor web lê os dados da conexão de rede e faz a análise das partes da requisição. Quando verifica a mensagem de requisição, o servidor web deve:

   - Analisa a primeira linha da mensagem, verificando o método requerido, o recurso especificado, e o número da versão, cada um separado por espaço em branco, e terminado por CRLF.
   - Ler os cabeçalhos da mensagem, cada um terminado com CRLF.
   - Detectar a linha em branco que finaliza os cabeçalhos, terminada com CRLF.
   - Ler o corpo da requisição.
   - Armazenar os dados lidos em uma estrutura.

   Apenas os métodos GET (utilizado para fazer requisições de recursos), HEAD (retorna o cabeçalho do documento) e POST (utilizado para enviar dados aos servidor - XML-RPC faz uso deste método) foram implementados.

3. Processar requisições.

   Após recebida e armazenada na estrutura deve-se processar a requisição de acordo com o método, recurso, cabeçalhos, e corpo opcional.

4. Mapear e acessar os recursos.

   Identificar o caminho do recurso solicitado, mapeando a URL da mensagem de requisição. O servidor deve ter uma raiz de documentos (docroot) configurada (no arquivo de configuração) para indicar o diretório base do mapeamento.

   De acordo com GOURLEY & TOTTY (2002), se a mensagem não especifica o arquivo o servidor pode:

   - Retornar um erro;
   - Retornar uma lista de arquivos do diretório base;

- Procurar o diretório, e retornar uma página HTML com o conteúdo (index.html, etc.);

No arquivo de configuração deve existir uma diretiva que configura o conjunto de nomes de arquivos que serão interpretados como arquivos padrão do diretório.

O recurso solicitado pode ser uma invocação remota de método que deve ser identificada e processada pelo sistema.

5. Construir as respostas.

   Depois de identificado o recurso é necessário construir a resposta.

   As mensagens de respostas usualmente contêm:

   - Um cabeçalho Content-Type, descrevendo o tipo MIME do corpo da resposta;
   - Um cabeçalho Content-Length, descrevendo o tamanho do corpo da resposta;
   - O conteúdo atual do corpo da mensagem.

   O servidor deve determinar o tipo MIME do corpo da resposta.

   O servidor pode retornar uma resposta de redirecionamento para um determinado recurso que foi movido de lugar (permanente ou temporariamente).

   O corpo da resposta para invocação remota de métodos é um texto XML, de acordo com a especificação do protocolo XML-RPC, que representa o resultado da invocação do método.

6. Enviar as respostas.

   Após a construção da resposta é necessário enviá-la ao cliente. O servidor deve manter o caminho do estado da conexão e ter especial atenção sobre a conexão persistente. Em conexões persistentes a conexão deve ficar aberta, e o servidor deve ter cuidado extra na computação correta do cabeçalho Content-length, ou o cliente não terá meios de saber quando a resposta termina.

7. Fazer o log.

   Quando a transação está completa, o servidor deve escrever no arquivo de log a descrição da transação.

Os seguintes cenários foram contemplados nesta primeira versão do servidor HTTP:

- Requisições do método GET
- Requisições do método POST
- Requisições do método HEAD
- Requisições do método OPTIONS

3.3.2 Descrição Modular

As funcionalidades necessárias ao servidor HTTP foram modeladas na forma de módulos. Estes módulos realizam tarefas específicas. Cada módulo recebe dados na entrada, processa estes dados e gera uma saída.

O servidor HTTP foi dividido nos seguintes módulos:

- Módulo Daemon: integra os módulos, inicializa e executa o servidor HTTP.
- Módulo Request: centraliza as tarefas relacionadas ao tratamento das requisições HTTP.
- Módulo Response: gerencia a construção e envio de respostas.
- Módulo Resource: mapeia e acessa os recursos.

A Figura 6 exibe os módulos do servidor HTTP.

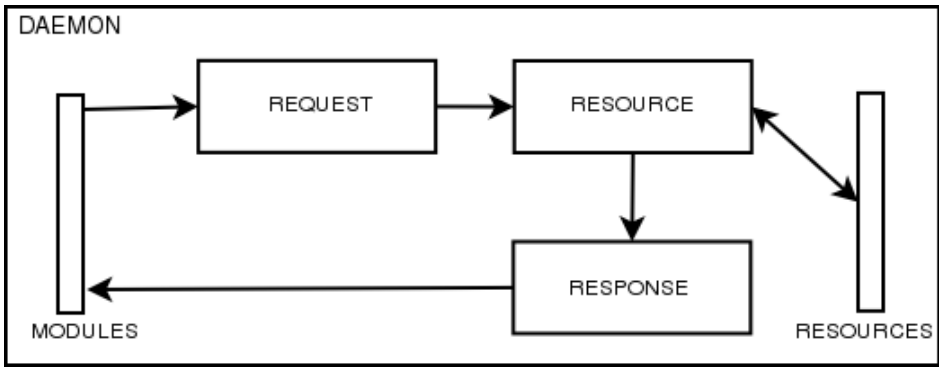

Figura 6: Módulos internos que integram o servidor HTTP

3.4 Servidor XML-RPC

O módulo Servidor XML-RPC foi projetado para realizar a análise dos dados do protocolo XML-RPC e gerar uma resposta por meio de funções disponibilizadas através de uma biblioteca (libxmlrpc) estática. Este projeto é descrito nesta seção.

3.4.1 Descrição Funcional

O funcionamento do servidor XML-RPC segue a mesma seqüência do servidor HTTP descrito anteriormente, exceto que não escreve mensagens de log e os recursos mapeados são funções.

As conexões aceitas pelo servidor XML-RPC ocorrem via socket UNIX, tendo como ponto de ligação um arquivo no sistema operacional.

As mensagens trocadas entre o servidor XML-RPC e o cliente devem ser de acordo com a seguinte definição:

```
Endpoint:  endpoint CRLF
Content-length:  length CRLF
CRLF
body
```

Onde `endpoint` é a referência do método a ser acessado, `length` o tamanho do corpo (`body`) da mensagem e `CRLF` é a representação para os caracteres de controle (carriage return e line feed). `body` deve estar no formato especificado pelo protocolo XML-RPC.

Exemplo 6 - Mensagem aceita pelo servidor XML-RPC:

```
Endpoint:  /web/RPC2
Content-Length:  99

<?xml version="1.0"?>
<methodCall>
   <methodName>getId</methodName>
   <params>
   </params>
</methodCall>
```

O Exemplo 6 mostra uma mensagem enviada para o servidor XML-RPC. Esta mensagem informa que o método a ser executado pelo servidor é `getId` e tem como referência `/web/RPC2`.

Para realizar a análise dos dados XML utilizou-se a bilbioteca do analisador XML GNU/Mini-XML versão 2.0, acessado em 07/06/2004 no seguinte endereço: http://www.easysw.com/~mike/mxml/

3.4.2 Descrição Modular

Os módulos que integram o servidor XML-RPC são os seguintes:

- Módulo Composer: realiza a composição da resposta XML-RPC.
- Módulo Daemon: integra os módulos, inicializa e executa o Servidor XML-RPC.
- Módulo Functions: funções disponibilizadas pelo servidor.
- Módulo Parser: faz a análise dos dados XML-RPC.
- Módulo Register: gerencia o registro das funções.
- Módulo Request: recebe e analisa a requisição XML-RPC.
- Módulo Response: gerencia a construção e envio da resposta XML-RPC.

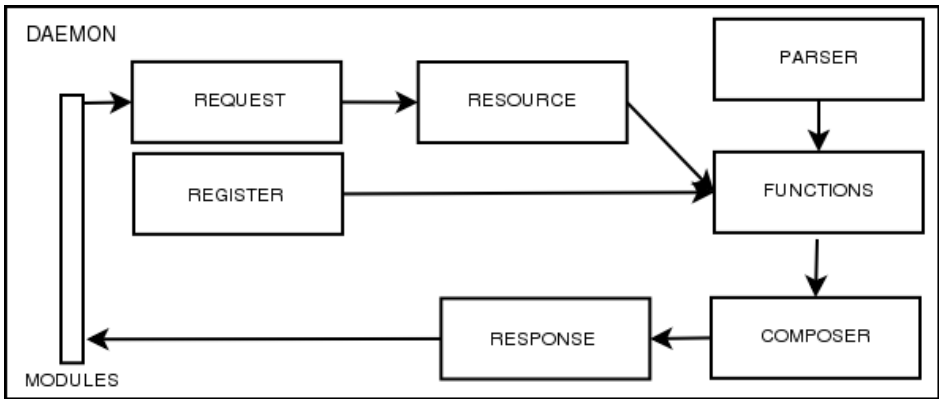

Figura 7: Módulos internos que integram o servidor XML-RPC

A integração destes módulos é realizada pelo módulo Daemon, como pode-se observar na Figura 7.

Os módulos Parser e Composer são utilizados pelas funções que são registradas pelo módulo Register para serem disponibilizadas pelo servidor XML-RPC.

3.5 Servidor HTTP/XML-RPC

Com os módulos descritos e definidos eles passam a ser integrados, como pode-se observar na Figura 8, para formarem o sistema, ou seja, o Servidor HTTP/XML-RPC.

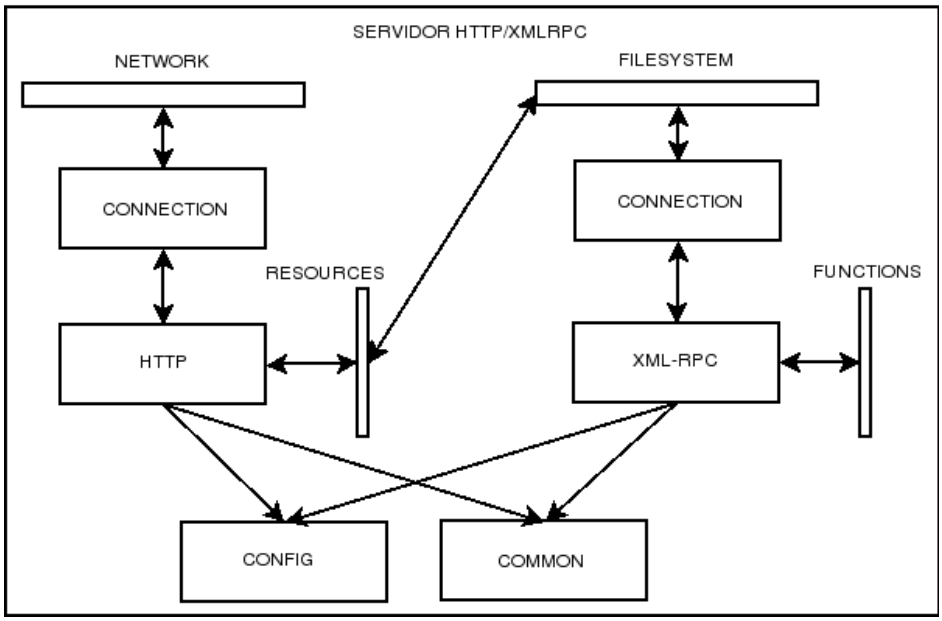

Figura 8: Módulos do Servidor HTTP/XML-RPC

4 Implementacão

A implementação do sistema procurou refletir a estrutura modular definida no projeto e integrar os módulos gerados.

4.1 Estruturas de Dados

As informações a serem manipuladas pelo sistema são armazenadas em estruturas que são utilizadas pelos módulos do mesmo.

Estas estruturas implementadas encapsulam as informações dinâmicas (definidas em tempo de execução) e estáticas (definidas em tempo de compilação). As infomações dinâmicas representam a requisição HTTP, os cabeçalhos da requisição, o recurso solicitado na requisição, e a resposta construída para esta requisição. Já as insformações estáticas representam os códigos de status da resposta, e as funções disponibilizadas pelo servidor XML-RPC.

O fragmento de código a seguir se refere a estrutura utilizada para armazenar as informações relativas a requisição do protocolo HTTP.

```
typedef struct {
  /* Cabecalhos da requisicao. */
  KEY_VALUE *header;
  /* Armazena o nome do metodo. */
  unsigned char method[REQUEST_METHOD_SIZE];
  /* Armazena a versao do protocolo HTTP. */
  unsigned char version[REQUEST_VERSION_SIZE];
  /* Armazena a caminho do recurso. */
  unsigned char path[REQUEST_PATH_SIZE];
  /* O corpo da requisicao. */
```

```
  unsigned char *body;
  /* Decritor do socket utilizado. */
  int descriptor;
  /* Numero total de caracteres lidos na requisicao. */
  unsigned long int length;
  /* Indica se a estrutura esta sendo utilizada. */
  unsigned used : 1;
  /* Indica se a linha de requisicao esta correta. */
  unsigned rl_status : 1;
  /* Indica se a requisicao esta completa. */
  unsigned status : 1;
  /* Identificador da estrutura. */
  int id;
} request_t;
```

As estruturas das informações dinâmicas são alocadas em tempo de compilação, na forma de vetores de tamanho fixo, de acordo com a quantidade máxima de conexões simultâneas definida pelo módulo Config no arquivo ConfigDefine.h listado a seguir. Este arquivo também define as configurações padrão do sistema.

```
#ifndef _CONFIG_DEFINE_
#define _CONFIG_DEFINE_

#define CONFIG_MAX_CONNECTIONS 50

#define CONFIG_USER_SIZE 64
#define CONFIG_GROUP_SIZE 64
#define CONFIG_DIR_INDEX_SIZE 64
#define CONFIG_DEFAULT_TYPE_SIZE 64
#define CONFIG_FILE_NAME_SIZE 128
#define CONFIG_DOCROOT_SIZE 128
#define CONFIG_LOG_FILE_SIZE 128
#define CONFIG_ADDRESS_SIZE 128
#define CONFIG_MIME_TYPES_SIZE 128
#define CONFIG_FILE_NAME "/etc/httpd.conf"
```

```
#define CONFIG_HTTP_PORT 80
#define CONFIG_LOG_FILE "/var/log/httpd.log"
#define CONFIG_KEEP_ALIVE 5
#define CONFIG_XMLRPC_ADDRESS "/etc/un_socket"
#define CONFIG_SERVER_NAME "Squirrel Server 1.0"

#endif
```

4.2 Mútiplas Conexões

O sistema foi implementado para possibilitar múltiplas conexões simultâneas. Esta capacidade depende da bilbioteca pthread. Ambos, o servidor HTTP e o servidor XML-RPC, utilizam esta biblioteca.

O sistema aceita uma conexão e cria uma nova linha de execução para trata-lá. Após criar a nova linha de execução, o sistema está apto a aceitar novas conexões enquanto a linha de execução criada trata a conexão anterior.

4.3 Biblioteca libxmlrpc

Esta biblioteca integra as funções necessárias para efetuar a análie e composição de dados XML de acordo com o protocolo XML-RPC. Os módulos Composer, Parser e Register, compõem esta biblioteca e são descritos nesta seção.

4.3.1 Módulo Composer

Este módulo é utilizado para realizar a composição de documentos XML-RPC e possui as seguintes funções implementadas e disponíveis:

- `XML_NODE *XmlRpcBuildComposer (XML_NODE *tree, int isFault);`
- `XML_NODE *XmlRpcAddInteger (XML_NODE *parent, int value, int isParam);`
- `XML_NODE *XmlRpcAddBoolean (XML_NODE *parent, int value, int isParam);`

- `XML_NODE *XmlRpcAddBase64 (XML_NODE *parent, char *value, int isParam);`
- `XML_NODE *XmlRpcAddDouble (XML_NODE *parent, double value, int isParam);`
- `XML_NODE *XmlRpcAddDate (XML_NODE *parent, char *value, int isParam);`
- `XML_NODE *XmlRpcAddString (XML_NODE *parent, char *value, int isParam);`
- `XML_NODE *XmlRpcAddArray (XML_NODE *parent, int isParam);`
- `XML_NODE *XmlRpcAddStruct (XML_NODE *parent, int isParam);`
- `XML_NODE *XmlRpcAddStructMember (XML_NODE *struct_node, char *name, void *value, int type_value);`

A função `XmlRpcBuildComposer` cria uma nova árvore XML, tendo como nodo raiz tree, e retorna um ponteiro para o elemento `<params>` se `isFault` é `XMLRPC_NO_FAULT`, o elemento `<fault>` se `isFault` é `XMLRPC_FAULT` ou `NULL`. Adiciona os seguintes elementos na árvore:

```
<?xml version="1.0"?>
<methodResponse>
<params> ou <fault>
```

As funções `XmlRpcAddInteger`, `XmlRpcAddBoolean`, `XmlRpcAddBase64`, `XmlRpcAddDouble`, `XmlRpcAddDate` e `XmlRpcAddString`, criam um elemento para adicionar `value` com os seguintes elementos a partir do elemento `parent`:

```
<param><value><type>value</type></value></param>
```

Onde `type` é um dos tipos básicos de dados do protocolo XML-RPC (i4 ou int, boolean, string, base64, double, dateTime.iso8601) de acordo com cada função. Os elementos criados serão filhos de `<param>`. Os valores possíveis são: `XMLRPC_PARAM_ELEMENT` ou `XMLRPC_NO_PARAM_ELEMENT`.

O elemento `<param>` é criado se `isParam` for `XMLRPC_PARAM_ELEMENT`. O retorno dessas funções é o elemento `parent` ou `NULL`.

A função `XmlRpcAddArray` gera o elemento `<array>` a partir do elemento `parent`: `<param><value><array><data></data></array></value></param>`. O retorno desta função é o elemento `<data>` ou NULL.

A função `XmlRpcAddStruct` adiciona os seguintes elementos no nodo `parent`: `<param><value><struct></struct></value></param>`. A função retorna o elemento `<struct>` ou `NULL`.

A função `XmlRpcAddStructMember` adiciona os seguintes elementos no elemento `struct_node`:
`<member><name>name</name><value><type>value</type></value></member>` . Onde `name` é o nome do membro da estrutura, `type` é o tipo do valor do elemento definido por `type_value`, e `value` é o valor do elemento. Esta função retorna o elemento `<struct>`, exceto para os tipos `XMLRPC_ARRAY_TYPE` que retorna o elemento `<data>` e `XMLRPC_STRUCT_TYPE` que retorna o elemento `<struct>` criado, ou `NULL`.

Os valores possíveis para `type_value` são:
`XMLRPC_INT_TYPE, XMLRPC_STRING_TYPE, XMLRPC_BOOLEAN_TYPE, XMLRPC_DATE_TYPE, XMLRPC_DOUBLE_TYPE, XMLRPC_BASE64_TYPE, XMLRC_ARRAY_TYPE, XMLRPC_STRUCT_TYPE.`
Para os tipos `XMLRPC_ARRAY_TYPE` e `XMLRPC_STRUCT_TYPE` `value` não é considerado.

4.3.1.1 Exemplos

Exemplo de composição do seguinte documento XML-RPC:

```
<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
       <value><string>alexandre</string></value>
      </param>
      <param>
       <value><i4>24</i4></value>
      </param>
   </params>
</methodResponse>
```

A implementação do método que compõem o documento pode ser observada a seguir:

```
XML_NODE *simpleComposer(XML_NODE *response_tree) {
    XML_NODE *params_node;
    char *user = "alexandre";
    int age = 24;

    params_node = XmlRpcBuildComposer(response_tree, XMLRPC_NO_FAULT_ELEMENT);

    params_node = XmlRpcAddString(params_node, user, XMLRPC_PARAM_ELEMENT);
    params_node = XmlRpcAddInteger(params_node, age, XMLRPC_PARAM_ELEMENT);

    return params_node;
}
```

4.3.2 Módulo Parser

A análise de documentos XML-RPC é realizada por este módulo que possui as seguintes funções implementadas e disponíveis:

- `int XmlRpcFunctionName (XML_NODE *tree, char *name);`
- `int XmlRpcIntegerValue (XML_NODE *node, int *value);`
- `int XmlRpcStringValue (XML_NODE *node, char *value);`
- `int XmlRpcBooleanValue (XML_NODE *node, int *value);`
- `int XmlRpcDoubleValue (XML_NODE *node, double *value);`
- `int XmlRpcDateValue (XML_NODE *node, char *value);`
- `int XmlRpcBase64Value (XML_NODE *node, char *value);`
- `int XmlRpcArraySize (XML_NODE *array_node, int *size);`
- `XML_NODE *XmlRpcArrayElement (XML_NODE *array_node, int idx, int array_size);`
- `XML_NODE *XmlRpcStructMember (XML_NODE *struct_node, char *member_name);`

- `XML_NODE *XmlRpcNode(XML_NODE *node, XML_NODE *top, char *name);`

A função `XmlRpcFunctionName` obtém o nome da função XML-RPC a partir de `tree` que é a àrvore do documento XML e armazena este nome em `name`. Retorna 0 se Ok, -1 se ocorrer erro.

As funções `XmlRpcIntegerValue`, `XmlRpcStringValue`, `XmlRpcBooleanValue`, `XmlRpcDoubleValue`, `XmlRpcDateValue` e `XmlRpcBase64Value` obtêm o valor contido no elemento `node` que deve ser um dos tipos de dados XML-RPC de acordo com cada função. Retornam 0 se Ok, ou -1 se ocorrer erro.

A função `XmlRpcArraySize` obtém o tamanho do array a partir do elemento `array_node` que deve ser o elemento `<array>` e armazena este valor em `size`. Retorna 0 se Ok, e -1 se ocorrer erro.

A função `XmlRpcArrayElement` obtém o elemento do array `array_node`, de tamanho `array_size`, indicado pelo índice `idx`. Retorna um elemento que representa um dos tipos de dados XML-RPC ou `NULL`.

A função `XmlRpcStructMember` obtém o membro `member_name` da estrutura indicada pelo elemento `struct_node`. Retorna o elemento `<member>` da estrutura ou `NULL`.

A função `XmlRpcNode` obtém o elemento de nome name a partir do elemento `node` que tem como elemento pai `top`. Retorna o elemento desejado ou `NULL`.

4.3.2.1 Exemplos

Exemplo de análise do seguinte documento XML-RPC:

```
<?xml version="1.0"?>
<methodCall>
   <methodName>printValues</methodName>
   <params>
      <param>
       <value><i4>41</i4></value>
      </param>
      <param>
       <value><string>test</string></value>
      </param>
```

```
      <param>
       <value><double>45.0</double></value>
      </param>
   </params>
</methodCall>
```

O método que faz a análise do documento pode ser observado a seguir:

```
int printValues (XML_NODE *request_tree)
{
    XML_NODE *param_node, *value_node;
    int i;
    char c[32];
    double d;

    if ((param_node = XmlRpcNode(request_tree,
                            request_tree,
                            "param")) == NULL) {
      printf("xml parser error.");

      return -1;
    }

    if ((value_node = XmlRpcNode(param_node,
                            param_node->parent,
                            "value")) == NULL) {
      printf("xml parser error: value element not found.");

      return -1;
    }

    XmlRpcIntegerValue(value_node, &i);

    if ((param_node = XmlRpcNode(request_tree,
```

```
                        request_tree,
                        "param")) == NULL) {
  printf("xml parser error.");

  return -1;
}

if ((value_node = XmlRpcNode(param_node,
                        param_node->parent,
                        "value")) == NULL) {
  printf("xml parser error: value element not found.");

  return -1;
}

strcpy(c, XmlRpcStringValue(value_node));


if ((param_node = XmlRpcNode(request_tree,
                        request_tree,
                        "param")) == NULL) {
  printf("xml parser error.");

  return -1;
}

if ((value_node = XmlRpcNode(param_node,
                        param_node->parent,
                        "value")) == NULL) {
  printf("xml parser error: value element not found.");

  return -1;
}

XmlRpcDoubleValue(value_node, &d);
```

```
    printf("integer value: %d\n", i);
    printf("string value: %s\n", c);
    printf("double value: %f\n", d);

    return 0;
}
```

4.3.3 Módulo Register

O registro das funções é realizado por este módulo que possui uma única função que implementada e disponível:

- `int XmlRpcRegisterFunction (char *name, char *endpoint, PFUNCTION pf);`

A função `XmlRpcRegisterFunction` realiza o registro da função `pf` de nome `name` e endereço `endpoint`. Retorna 0 se OK, ou -1 se ocorrer erro.

5 Configuração e Produção

A configuração do sistema é feita através de um arquivo texto que define as informações através de pares chave/valor.

5.1 Arquivo de Configuração

O objetivo deste arquivo é efetuar a configuração do servidor HTTP/XML-RPC. A forma como as informações devem ser fornecidas segue a seguinte regra:

`<key>: *SP <value> CR LF`

`<key>` é o nome da informação, seguido por dois pontos (:), 0 (zero) ou mais espaços em branco (SP), onde `<value>` é o valor da informação, e `CR LF` os caracteres de nova linha. `<value>` pode ter 0 (zero) ou mais strings separadas por vírgula (,).

Exemplo de um arquivo de configuração:

```
Port: 80
User: demari
Group: pqfw
AccessLog: /var/log/httpd.log
DocumentRoot: /eqpt/www
DirectoryIndex: index.htm,index.html
KeepAliveTimeout: 30
MimeTypes: /etc/mime.types
DefaultType: text/txt
HTTPMaxConnections: 50
XMLRPMaxConnections: 50
XMLRPCAddress: /etc/un_socket
```

- Port - A porta de comunicação.
- User - O usuário do processo.
- Group - O grupo do processo.
- AccessLog - O arquivo de log de acesso.
- DocumentRoot - O diretório base dos recursos que serão disponibilizados.
- DirectoryIndex - O nome do arquivo que será o indexador do diretório.
- KeepAliveTimeout - Tempo de keep-alive (em segundos).
- MimeTypes - Arquivo para gerar os tipos MIME.
- DefaultType - Tipo MIME padrão.
- HTTPMaxConnections - Quantidade máxima de conexões simultâneas no servidor HTTP.
- XMLRPMaxConnections - Quantidade máxima de conexões simultâneas no servidor XML-RPC.
- XMLRPCAddress - O arquivo de ligação para a conexão do servidor XMLRPC.

Comentários podem ser adicionados com o caracter #.

5.2 Disponibilização de Funções XML-RPC

As funções a serem disponibilizadas pelo servidor XML-RPC devem ter o seguinte protótipo:

```
XML_NODE *NomeDaFuncao (XML_NODE *request_tree, XML_NODE *response_tree)
```

As etapas para disponibilizar uma função no servidor XML-RPC são as seguintes:

1. Editar o arquivo XmlRpcFunctions.h e definir o protótipo da função.
2. Editar o arquivo XmlRpcRegisterManager.c e através da função `XmlRpcRegisterFunction` registrar a função com o nome e a referência da mesma.

3. Editar o arquivo xmlrpc/daemon/Makefile e adicionar o arquivo objeto da função na compilação do servidor.

Para exemplificar, seja a função de nome `getName`, listada a seguir, e com a referência `/exemplo/RPC2`.

```
XML_NODE *getName (XML_NODE *request_tree, XML_NODE *response_tree)
{
        XML_NODE *params;
        char *name = "Alexandre de Mari";

        params = XmlRpcBuildComposer(response_tree,
                                     XMLRPC_NO_FAULT_ELEMENT);

        XmlRpcAddString(params, name, XMLRPC_PARAM_ELEMENT);

        return params;
}
```

A definição do protótipo da função no arquivo XmlRpcFunctions.h ficaria da seguinte forma:

```
extern XML_NODE *getName (XML_NODE *request_tree, XML_NODE *response_tree);
```

Enquanto isso, no arquivo XmlRpcRegisterManager.c na função de mesmo mome:

```
XmlRpcRegisterFunction("getName", "/exemplo/RPC2", getId);
```

Para finalizar, basta adicionar a dependência da função no arquivo Makefile utilizado para a compilaçao.

5.3 Compilação e Execução

A compilação do sistem é realizada através do compilador GNU gcc utilizando o programa GNU Make.

Os módulos Servidor HTTP, Servidor XML-RPC, e a biblioteca (libxmlrpc) podem ser compilados separadamente ou juntos.

- Compilação do Servidor HTTP

Executar o Makefile no diretório ../http/daemon/.

- Compilação do Servidor XML-RPC

  Executar o Makefile no diretório ../xmlrpc/daemon/.

- Compilação da biblioteca libxmlrpc

  Executar o Makefile no diretório ../xmlrpc/libxmlrpc/.

- Compilação do sistema

  Executar o Makefile no diretório principal.

Para executar o Servidor HTTP deve-se digitar na linha de comando no diretório principal do sistema `http/daemon/httpd` "arquivo de configuração"e o Servidor XML-RPC `xmlrpc/daemon/xmlrpcd` "arquivo de configuração".

6 Conclusão

A oportunidade de desenvolver o trabalho de conlusão de curso em uma empresa de tecnologia e aplicar os conhecimentos obtidos no curso de Ciências da Computação foi muito importante. O ótimo ambiente de trabalho colaborou para o êxito na realização das atividades envolvidadas no desenvolvimento do servidor HTTP/XML-RPC.

As dificuldades encontradas foram principalmente em relação a linguagem de programação ANSI C, devido a falta de conhecimento do autor sobre a mesma, pois é pouco utilizada durante o curso de Ciências da Computação. Entretanto, é uma linguagem muito utilizada para o desenvolvimento de sistemas embutidos, sistemas operacionais, e é a principal linguagem utilizada pela Reason Tecnologia S.A.

Os assuntos tratados neste trabalho têm boa quantidade de documentação, tanto em livros como na Web, o que propiciou uma facilidade na obtenção das informações necessárias para a elaboração do mesmo.

Os objetivos descritos na introdução foram em sua maioria alcançados como, por exemplo, a utilização de apenas uma porta de comunicação e uma única biblioteca externa. No decorrer do trabalho, optou-se conscientemente pela utilização de conexões paralelas e persistentes a despeito do aumento no consumo de memória e processamento. Nesta abordagem, o servidor HTTP acabou tendo um consumo de memória maior devido aos benefícios adicionais que as conexões paralelas e persistentes proporcionam que são, a maior responsividade e menor carga de processamento (obtida pela utilização de múltiplas linhas de execução). No servidor XML-RPC o consumo e processamento foram equivalentes ao servidor originalmente utilizado, porém, com a expansão do sistema eles tornam-se menores. Além disso, a API proposta mostrou-se simples e eficiente.

Como trabalhos futuros pode-se citar a alteração na gerência interna de conexões para um gerência dinâmica que é solucionada através de linhas de processamento e alocação dinâmica de memória, a implementação do sistema de autenticação, a adição de novas funcionalidades do protocolo HTTP, e melhorias estruturais do sistema.

Considero que o desenvolvimento do Servidor HTTP/XML-RPC atendeu as expectativas. O domínio da tecnologia e o conhecimento e maturidade obtidos durante o desenvolvimento foram muito importantes e certamente serão úteis em trabalhos futuros e novos projetos.

Referências

GOURLEY, David; TOTTY, Brian. *HTTP: The Definitive Guide.* Estados Unidos: O'Relly & Associates, Inc. 2002.

NICHOLS, B; BUTTLAR, D; FARRELL, J. P. *Pthreads Programming.* Estados Unidos: O'Relly & Associates, Inc. 1996.

STEVENS, W. Richard. *Advanced Programming in the Unix Environment.* Estados Unidos: Addison-Wesley, 1993.

KERNIGHAN, Brian W.; RITCHIE, Dennis M. *The C Programming Language.* Estados Unidos: Prentice Hall, 1988.

KERNIGHAN, Brian W.; PIKE, Rob. *The Unix Programming Environment.* Estados Unidos: Prentice Hall, 1984.

FIELDING, R.; GETTYS, J.; MOGUL, J; FRYSTYK, H.; MASINTER, L.; LEACH, P.; BERNERS-LEE, T. *Request For Comments 2616 (RFC 2616): Hypertext Transefer Protocol - HTTP/1.1.* 1999. Disponível em http://www.w3.org/Protocols/rfc2616/rfc2616.txt, acesso em 15 de janeiro de 2004.

WINER, Dave. *XML-RPC Specification.* 1999. Disponível em http://www.xmlrpc.com/spec, acesso em 15 de janeiro de 2004.

W3C. *XML in 10 Points.* 1999. Disponível em http://www.w3.org/XML/1999/XML-in-10-points.html, acesso em 16 de janeiro 2004.

MARSHALL, Dave. *Programming in C - UNIX System Calls and Subroutines using C.* 1999 Disponível em http://www.cs.cf.ac.uk/Dave/C/node33.html, acesso em 16 de

janeiro de 2004.

VONDRAK, Cory. *Remote Procedure Call*. 1997. Disponível em http://www.sei.cmu.edu/str/descriptions/rpc.html, acesso em 16 de janeiro de 2004.

Anexos

```
/*****************************************************************************
 * $Source:
 * $Name: $
 * $Revision: 1.4 $
 * $Author: demari $
 * $Date: 2004/09/30 15:31:21 $
 *****************************************************************************/

#include <string.h>
#include <syslog.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <time.h>
#include "../../common/util.h"
#include "../../common/defs.h"
#include "../../connection/Connection.h"
#include "Request.h"

/*****************************************************************************
   Funcoes externas
*****************************************************************************/

int RequestManager (int sock,
                    unsigned char line_sep,
                    unsigned char kv_sep,
                    unsigned char value_sep,
                    REQUEST *request)
```

```
{
  unsigned char request_message[CONNECTION_MESSAGE_SIZE];
  struct timeval timeout;
  fd_set read_set;
  int ready;
  int str_index;
  int kv_index;
  int body_len;
  int body_index;
  int next_step;
  int prev_step;
  int request_size;
  int end_crlf;
  int total;

  end_crlf = 0;
  str_index = 0;
  kv_index = 0;
  body_index = 0;
  body_len = 0;
  next_step = RECEIVE;
  prev_step = -1;
  request_size = 0;
  total = 0;
  ready = 0;

  timeout.tv_sec = CONNECTION_TIMEOUT_SEC;
  timeout.tv_usec = 0;

  FD_ZERO(&read_set);
  FD_SET(sock, &read_set);

  memset(request_message, '\0', CONNECTION_MESSAGE_SIZE);

  while (!ready) {
```

```
switch (next_step) {
case RECEIVE:
  if ((total = select(FD_SETSIZE,
                      &read_set,
                      (fd_set *)NULL,
                      (fd_set *)NULL,
                      &timeout)) < 0) {
    return -1;
  }

  if (!FD_ISSET(sock, &read_set)) {
    return -1;
  }

  /* Recebe a mensagem de requisicao. */
  if (ReceivingMessage(sock,
                       CONNECTION_MESSAGE_SIZE,
                       (void *)request_message + request_size) != 0) {
    syslog(LOG_ERR, "receiving request error.");

    return -1;
  } else if ((request_size = (int)strlen(request_message)) <= 0) {
    return -1;
  }

  if (prev_step == -1) {
    next_step = REQUEST_LINE;
  } else {
    next_step = prev_step;
  }

  prev_step = RECEIVE;
  break;

case REQUEST_LINE:
```

```
/* Se a linha de requisicao ainda nao foi analisada corretamente. */
if (!request->rl_status) {
  /* Faz a analise da linha de requisicao. */
  if (ParseRequestLine(request_message,
                       value_sep,
                       line_sep,
                       &str_index,
                       request->method,
                       request->path,
                       request->version) != 0) {
    syslog(LOG_ERR, "parse request line error");

    return -1;
  } else {
    request->rl_status = 1;
  }
}

/* Se os proximos caracteres sao de nova linha. */
if (ContainsCRLF(request_message, &str_index)) {
  /* Se os proximos caracteres sao de nova linha,
     entao indicam o final da requisicao.
  */
  if (ContainsCRLF(request_message, &str_index)) {
    /* Conclui a analise da requisicao. */
    ready = 1;
  } else {
    /*
      Se os dois caracteres anteriores aos dois
      ultimos sao de nova linha.
    */
    if (end_crlf == 1) {
      ready = 1;
    } else if (str_index < request_size) {
      next_step = HEADER;
```

```
      } else {
        next_step = UPDATE_REQUEST;
        /* Indica que os dois ultimos caracteres sao de nova linha. */
        end_crlf = 1;
      }
    }
    /* Senao, se os dois ultimos caracteres nao sao de nova linha. */
  } else if (end_crlf == 0) {
    /* Se o indexador estah no final da mensagem. */
    if (str_index == request_size) {
      next_step = UPDATE_REQUEST;
      /* Senao, a linha de requisicao estah incorreta.*/
    } else {
      ready = 1;
    }
  } else {
    next_step = HEADER;
    end_crlf = 0;
  }

  prev_step = REQUEST_LINE;

  break;

case HEADER:
  /* Se o indexador estiver antes do final da string. */
  if (str_index < request_size) {
    /* Se os proximos caracteres sao de nova linha. */
    if (ContainsCRLF(request_message, &str_index)) {
      if (end_crlf || ContainsCRLF(request_message, &str_index)) {
        if (VerifyContentLength(request->header, &body_len) == 0) {
          next_step = BODY;
          prev_step = HEADER;

          break;
```

```
    } else {
      /* Conclui a analise da requisicao. */
      ready = 1;

      break;
    }
  } else {
    end_crlf = 1;
  }
} else if (!end_crlf && prev_step == RECEIVE) {
  ready = 1;

  break;
}

if (str_index < request_size) {
  if (ParseKeyValue(request_message,
                    kv_sep,
                    line_sep,
                    &str_index,
                    &kv_index,
                    request->header) != 0) {
    syslog(LOG_ERR, "key value parse error");

    return -1;
  } else {
    /* Se os proximos caracteres sao de nova linha. */
    if (ContainsCRLF(request_message, &str_index)) {
      if (ContainsCRLF(request_message, &str_index)) {
        if (VerifyContentLength(request->header, &body_len) == 0) {
          next_step = BODY;
          prev_step = HEADER;

          break;
        } else {
```

```
            /* Conclui a analise da requisicao. */
            ready = 1;
          }
        } else {
          next_step = UPDATE_REQUEST;
          end_crlf = 1;
        }
      } else if (prev_step == RECEIVE || end_crlf == 0) {
        next_step = UPDATE_REQUEST;
        end_crlf = 0;
      }
    }
  } else {
    next_step = UPDATE_REQUEST;
  }
}
prev_step = HEADER;

break;

case BODY:
if (request_size == 0) {
  ready = 1;

  break;
}

/* Se o passo anterior foi o HEADER. */
if (prev_step == HEADER) {
  if (VerifyContentLength(request->header, &body_len) == 0) {
    /* Se o tamanho do corpo for maior que o maximo permitido. */
    if (body_len > REQUEST_BODY_SIZE)
      body_len = REQUEST_BODY_SIZE;

    request->body = (char *)malloc(body_len + 1);
```

```
    memset(request->body, '\0', body_len + 1);
  } else {
    ready = 1;

    break;
  }
}
/* Realiza a analise do corpo da requisicao HTTP. */
if (ParseBody(request_message,
              request->body,
              str_index,
              &body_index,
              body_len) != 0) {

  free(request->body);

  request->body = (char *)NULL;

  /*
     Se o tamanho do corpo analisado eh igual ao tamanho indicado no
     cabecalho.
  */
} else if (body_index == body_len) {
  ready = 1;
} else {
  memset(request_message, '\0', CONNECTION_MESSAGE_SIZE);

  next_step = UPDATE_REQUEST;
  prev_step = BODY;

  str_index = 0;
}

break;
```

```
    case UPDATE_REQUEST:
      if (prev_step != BODY) {
        if (request_size == 0) {
          return -1;
        }
      } else {
        request_size = 0;
      }

      next_step = RECEIVE;

      break;

    }
  }

  return 0;
}


/*****************************************************************************
 * $Source:
 * $Name: $
 * $Revision: 1.4 $
 * $Author: demari $
 * $Date: 2004/09/30 15:31:21 $
 *****************************************************************************/

#include <string.h>
#include <syslog.h>
#include "../../connection/Connection.h"
#include "Response.h"

/*****************************************************************************
   Constantes
```

```
*****************************************************************************/
static const method_t _http_method[MAX_HTTP_METHODS] = {{"GET", GET, 1},
                                                        {"HEAD", HEAD, 1},
                                                        {"OPTIONS", OPTIONS, 1},
                                                        {"POST", POST, 1},
                                                        {"PUT", PUT, 0},
                                                        {"DELETE", DELETE, 0},
                                                        {"TRACE", TRACE, 0}};

/*****************************************************************************
   Funcoes externas
*****************************************************************************/

int ResponseManager (char *docroot,
                     REQUEST *request,
                     RESPONSE *response,
                     RESOURCE *resource)
{
  unsigned char response_message[CONNECTION_MESSAGE_SIZE];

  //syslog(LOG_DEBUG, "Generating response.");

  memset(response_message, '\0', CONNECTION_MESSAGE_SIZE);

  /* Se a linha de requisicao estiver correta. */
  if (request->rl_status) {
    /* Verifica o identificador do metodo. */
    VerifyMethodID(request->method,
                   &response->method_id,
                   _http_method);

    /* Verifica se a conexao sera mantida. */
    VerifyKeepAlive(request->header,
                    request->version,
                    &response->keep_alive,
```

```
                &response->conn_status);

VerifyHost(request->header, response->server_address);

if (response->method_id != OPTIONS) {
  //syslog(LOG_DEBUG, "Mapping resource: %s", request->path);

  /* Realiza o mapeamento do recurso. */
  if (MappingResource(docroot,
                      request->path,
                      resource) != 0) {
    syslog(LOG_ERR, "mapping error");

    return -1;
  }

  /* Se o recurso existe. */
  if (resource->exist) {
    switch (response->method_id) {
    case HEAD:
      break;
    case GET:
      if (OpenFile(resource->absolute_path, &resource->fd)) {
        syslog(LOG_ERR, "error openning file.");
      }

      break;
    case POST:
      if (resource->type == RESOURCE_XMLRPC) {
        if (request->body != NULL && request->body != 0) {
          if (OpenRPC2(request->body,
                       docroot,
                       resource->absolute_path,
                       &resource->fd) != 0) {
            syslog(LOG_ERR, "error openning RPC2.");
```

```
          }
        }
      }

      break;
    }
  }
 } else {

 }
}

if (StatusCode(response, resource) != 0) {
  syslog(LOG_ERR, "status code error");

  return -1;
}
//syslog(LOG_DEBUG, "status code OK.");

StatusLineBuilder(response->status_line,
                  response->reason_phrase,
                  response->status_code);
//syslog(LOG_DEBUG, "status line builder OK.");

HeaderBuilder(response, resource, _http_method);
//syslog(LOG_DEBUG, "header builder OK: %s", response->header);

strcat(response_message, response->status_line);
strcat(response_message, response->header);

/* Envia a mensagem de resposta. */
if (SendingMessage(response->socket_des,
                   resource->fd,
                   response_message) != 0) {
  syslog(LOG_ERR, "sendind response error.");
```

```
    return -1;
  }

  return 0;
}


/*****************************************************************************
 * $Source:
 * $Name: $
 * $Revision: 1.2 $
 * $Author: demari $
 * $Date: 2004/09/30 15:31:21 $
 *****************************************************************************/
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>
#include <syslog.h>
#include "Resource.h"

/*****************************************************************************
   Variaveis internas
*****************************************************************************/

static char _directory_index[RESOURCE_MAX_DIR_INDEX][RESOURCE_DIR_INDEX_SIZE];
static int _dir_index_counter = 0;

/*****************************************************************************
   Funcoes internas
*****************************************************************************/

static int VerifyDirectoryIndex (unsigned char *path, struct stat *buf)
{
```

```
  unsigned char aux[RESOURCE_ABSOLUTE_PATH_SIZE];
  int i;

  for (i = 0; i < _dir_index_counter; i++) {
    memset(aux, '\0', RESOURCE_ABSOLUTE_PATH_SIZE);

    strncpy(aux, path, RESOURCE_ABSOLUTE_PATH_SIZE);
    strncat(aux, _directory_index[i], RESOURCE_ABSOLUTE_PATH_SIZE);

    if (stat(aux, buf) == 0) {
      strncat(path, _directory_index[i], RESOURCE_ABSOLUTE_PATH_SIZE);

      return 0;
    }
  }

  return -1;
}

/*****************************************************************************
   Funcoes externas
*****************************************************************************/

int AddDirectoryIndex (char *dir_index)
{
  if (_dir_index_counter >= RESOURCE_MAX_DIR_INDEX) {
    syslog(LOG_ERR, "directory index files out of bound.");

    return -1;
  }

  strncpy(_directory_index[_dir_index_counter],
          dir_index,
          RESOURCE_DIR_INDEX_SIZE);
```

```
  _dir_index_counter++;

  return 0;
}

int MappingResource (unsigned char *docroot,
                     unsigned char *uri,
                     RESOURCE *resource)
{
  struct stat buf;
  int len;

  if (strlen(docroot) == 0) {
    syslog(LOG_ERR, "mapping resource -> bad document root.");

    return -1;
  }

  if (resource == (void *)NULL) {
    syslog(LOG_ERR, "mapping resource -> bad resource struct.");

    return -1;
  }

  /* Verifica se a uri tenta acessar arquivos abaixo do docroot. */
  if (strlen(uri) != 0) {
    if (strstr(uri, "../") != (char *)NULL) {
      resource->exist = 0;

      return 0;
    }
  }

  strncpy(resource->absolute_path, docroot, RESOURCE_ABSOLUTE_PATH_SIZE);
```

```
len = strlen(resource->absolute_path);

/* Concatena o caractere '/' com o docroot se for necessario. */
if (resource->absolute_path[len - 1] != '/' && uri[0] != '/') {
  strcat(resource->absolute_path, "/");
} else {
  strncpy(resource->relative_path, uri, RESOURCE_RELATIVE_PATH_SIZE);
}

strncat(resource->absolute_path,
       resource->relative_path,
       RESOURCE_ABSOLUTE_PATH_SIZE);

len = strlen(resource->absolute_path);

/* Obtem os atributos do recurso. */
if (stat(resource->absolute_path, &buf) < 0) {
  resource->exist = 0;

  return 0;
}

/* O recurso existe. */
resource->exist = 1;

/* Verifica as permissoes do recurso. */
if (VerifyPermissions(buf.st_mode, resource) != 0) {
  syslog(LOG_ERR, "mapping resource -> verify permissions error");

  return -1;
}

/* Verifica se o recurso e um diretorio. */
if (S_ISDIR(buf.st_mode)) {
  /* Verifica se o recurso e uma cahamda XML-RPC. */
```

```
  if (strstr(resource->absolute_path, RESOURCE_RPC2) != (char *)NULL &&
      resource->absolute_path[len - 1] == '2') {
    resource->type = RESOURCE_XMLRPC;
  } else {
    if (resource->absolute_path[len - 1] == '/') {
      /* Verifica se o diretorio possui um arquivo indexador. */
      if (VerifyDirectoryIndex(resource->absolute_path, &buf) != 0) {
        resource->exist = 0;
      } else {
        resource->type = RESOURCE_FILE;
        resource->size = buf.st_size;

        strftime(resource->date,
                 (size_t)RESOURCE_DATE_SIZE,
                 "%c",
                 gmtime(&buf.st_mtime));
      }
    } else {
      strcat(resource->relative_path, "/");
      resource->type = RESOURCE_DIR;
    }
  }
} else if (S_ISSOCK(buf.st_mode)) {
  resource->type = RESOURCE_DYN;
} else if (S_ISREG(buf.st_mode)) {
  resource->type = RESOURCE_FILE;
  resource->size = buf.st_size;

  strftime(resource->date,
           (size_t)RESOURCE_DATE_SIZE,
           "%c",
           gmtime(&buf.st_mtime));
} else {
  resource->type = RESOURCE_OTHER;
}
```

```
  return 0;
}


/*****************************************************************************
 * $Source:
 * $Name: $
 * $Revision: 1.2 $
 * $Author: demari $
 * $Date: 2004/09/30 15:31:21 $
 *****************************************************************************/
#include <string.h>
#include <syslog.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <time.h>
#include "../../common/util.h"
#include "../../common/defs.h"
#include "../../common/KeyValue.h"
#include "../../connection/Connection.h"
#include "XmlRpcRequest.h"

/*****************************************************************************
   Funcoes externas
*****************************************************************************/

char *XmlRpcRequestManager (char *request_body, char *endpoint, int sock)
{
  unsigned char request_message[CONNECTION_MESSAGE_SIZE];
  unsigned char header[VALUE_SIZE];
  unsigned char key[KEY_SIZE];
  unsigned char value[KEY_SIZE];
  struct timeval timeout;
```

```
fd_set read_set;
int ready;
int total;
int idx;
int body_len;
int body_idx;
int request_size;
int init_body;
int next_step;
int prev_step;
int end_crlf;

body_len = -1;
body_idx = 0;
init_body = 1;
next_step = RECEIVE;
prev_step = -1;
request_size = 0;
end_crlf = 0;
total = 0;
ready = 0;

timeout.tv_sec = CONNECTION_TIMEOUT_SEC;
timeout.tv_usec = 0;

FD_ZERO(&read_set);
FD_SET(sock, &read_set);

memset(header, '\0', VALUE_SIZE);
memset(key, '\0', KEY_SIZE);
memset(request_message, '\0', CONNECTION_MESSAGE_SIZE);

while (!ready) {

  switch (next_step) {
```

```
  /* Recebe a mensagem. */
case RECEIVE:
  /* Obtem o status do descritor do socket. */
  if ((total = select(FD_SETSIZE,
                      &read_set,
                      (fd_set *)NULL,
                      (fd_set *)NULL,
                      &timeout)) < 0) {
    return(NULL);
  }

  /* Verifica se o descritor do socket esta pronto para leitura. */
  if (!FD_ISSET(sock, &read_set)) {
    return(NULL);
  }

  /* Recebe a mensagem de requisicao. */
  if (ReceivingMessage(sock,
                       CONNECTION_MESSAGE_SIZE,
                       (void *)request_message + request_size) != 0) {
    syslog(LOG_ERR, "receiving request error.");

    return(NULL);
  }

  if (prev_step == -1) {
    next_step = HEADER;
  } else {
    next_step = prev_step;
  }
  //printf("\nXMLRPC, request: \n %s\n", request_message);
  break;

case HEADER:
  if (strlen(endpoint) == 0) {
```

```
  /* Obtem o cabecalho 'Endpoint' da requisicao. */
  if (ParseValue(request_message, key, ':', &idx, KEY_SIZE) != 0) {
    syslog(LOG_ERR, "parse value error");

    return(NULL);
  }

  idx++;
  if (ParseValue(request_message, endpoint, '\n', &idx, VALUE_SIZE) != 0) {
    syslog(LOG_ERR, "parse value error");

    return(NULL);
  }
}
/*
   Verifica se os proximos caracteres sao de nova linha. Se nao forem a
   requisicao esta incompleta.
*/
if (ContainsCRLF(request_message, &idx)) {
  if (end_crlf || ContainsCRLF(request_message, &idx)) {
    next_step = BODY;

    break;
  }
  end_crlf = 1;
} else if (body_len >= 0) {
  next_step = UPDATE_REQUEST;
  prev_step = HEADER;

  break;
}

if (body_len == -1) {
  if (idx < (int)strlen(request_message)) {
    memset(key, '\0', KEY_SIZE);
```

```
      memset(value, '\0', KEY_SIZE);
      /* Obtem o cabecalho 'Content-Length' da requisicao. */
      if (ParseValue(request_message, key, ':', &idx, KEY_SIZE) != 0) {
        syslog(LOG_ERR, "parse value error");

        return(NULL);
      }

      idx++;
      if (ParseValue(request_message, value, '\n', &idx, KEY_SIZE) != 0) {
        syslog(LOG_ERR, "parse value error");

        return(NULL);
      }

      body_len = atol(value);
      end_crlf = 0;
    } else {
      next_step = UPDATE_REQUEST;
      prev_step = HEADER;

      break;
    }
  }

  break;
  /* Faz a analise dos cabecalhos e do corpo da mesagem. */
case BODY:
  if ((request_size = strlen(request_message)) == 0) {
    ready = 1;

    break;
  }

  if (init_body) {
```

```
  if (body_len == 0) {
    ready = 1;

    break;
  }
  /* Se o tamanho do corpo for maior que o maximo permitido. */
  if (body_len > BODY_SIZE)
    body_len = BODY_SIZE;

  /* Aloca memoria para o corpo da requisicao. */
  request_body = (char *)malloc(body_len);
  memset(request_body, '\0', body_len);

  init_body = 0;
}

if (idx < request_size) {
  /* Realiza a leitura do corpo da mensagem. */
  while (body_idx < body_len) {
    request_body[body_idx] = request_message[idx];

    idx++;
    if (idx < request_size) {
      body_idx++;
    } else {
      break;
    }
  }

  /* Verifica se todo o corpo da mensagem foi obtido. */
  if (body_idx != body_len - 1) {
    next_step = UPDATE_REQUEST;
    prev_step = BODY;
  } else {
    /* A analise da requisicao esta completa. */
```

```
          ready = 1;
        }
      } else {
        next_step = UPDATE_REQUEST;
        prev_step = BODY;
      }

      break;

    case UPDATE_REQUEST:
      if (prev_step != BODY) {
        if ((request_size = strlen(request_message)) == 0) {
          return(NULL);
        }
      } else {
        request_size = 0;
      }
      next_step = RECEIVE;

      break;
    }
  }

  return request_body;
}


/*****************************************************************************
 * $Source:
 * $Name: $
 * $Revision: 1.3 $
 * $Author: demari $
 * $Date: 2004/09/30 15:31:21 $
 *****************************************************************************/
#include <string.h>
#include <syslog.h>
```

```
#include <pthread.h>
#include "../parser/XmlRpcParser.h"
#include "../composer/XmlRpcComposer.h"
#include "../register/XmlRpcRegister.h"
#include "../../common/KeyValueDefine.h"
#include "../../connection/Connection.h"
#include "XmlRpcResponse.h"

/*****************************************************************************
   Variaveis internas
*****************************************************************************/

static pthread_mutex_t _load_string = PTHREAD_MUTEX_INITIALIZER;

/*****************************************************************************
   Funcoes externas
*****************************************************************************/

char *XmlRpcResponseManager (char *request_body,
                             char *endpoint,
                             int sock)
{
  unsigned char function_name[XMLRPC_FUNCTION_NAME_SIZE];
  unsigned char header[VALUE_SIZE];
  unsigned char key[KEY_SIZE];
  unsigned char value[KEY_SIZE];
  unsigned char *response_body;
  XML_NODE *request_tree, *response_tree;
  PFUNCTION foo;

  memset(function_name, '\0', XMLRPC_FUNCTION_NAME_SIZE);

  response_tree = NULL;

  pthread_mutex_lock(&_load_string);
```

```
request_tree = mxmlLoadString(NULL, request_body, MXML_TEXT_CALLBACK);
pthread_mutex_unlock(&_load_string);

if (request_tree == NULL) {
  syslog(LOG_ERR, "bad xml document");

  return (char *)NULL;
}

if (XmlRpcFunctionName(request_tree, function_name) != 0) {
  syslog(LOG_ERR, "rpc function not found.");

  mxmlDelete(request_tree);

  return (char *)NULL;
}

if ((foo = (PFUNCTION)GetFunction(function_name, endpoint)) == NULL) {
  syslog(LOG_ERR, "mapping function error");

  mxmlDelete(request_tree);

  return (char *)NULL;
}

response_tree = (*foo)(request_tree, response_tree);

if (response_tree != NULL) {
  response_body = mxmlSaveAllocString(response_tree->parent->parent, MXML_NO_CALLB
} else {
  syslog(LOG_ERR, "response build error.");

  mxmlDelete(request_tree);

  return (char *)NULL;
```

```
  }

  mxmlDelete(request_tree);
  mxmlDelete(response_tree->parent->parent);

  //syslog(LOG_DEBUG, "body: %s", response_body);

  if (response_body != NULL) {
    memset(header, '\0', KEY_SIZE);
    memset(key, '\0', KEY_SIZE);
    memset(value, '\0', KEY_SIZE);

    strcpy(header, "Content-length: ");
    sprintf(value, "%d", strlen(response_body));
    strcat(header, value);
    strcat(header, "\r\n");
    strcat(header, "\r\n");

    if (SendingMessage(sock, -1, header) != 0) {
      syslog(LOG_ERR, "sendind response body error ");
    } else if (SendingMessage(sock, -1, response_body) != 0) {
      syslog(LOG_ERR, "sendind response body error ");
    }
  }
  /*******************FIM DA RESPOSTA************************/

  free(response_body);
  response_body = (char *)NULL;


  return response_body;
}
```