Miguel Ferreira Pereira Gomes

Canal SCADA na WEB

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2002

Miguel Ferreira Pereira Gomes

Canal SCADA na WEB

Tese submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Mestre em Informática

Dissertação realizada sob a supervisão do
Professor Doutor João Correia Lopes,
Professor Auxiliar da
Faculdade de Engenharia da Universidade do Porto

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2002

Communicate! It can't make things any worse.

Gore Vidal

Para o meu pai, António Gomes

Agradecimentos

Em primeiro lugar gostava de agradecer ao professor João Correia Lopes, pela disponibilidade que sempre demonstrou, pela pertinência das questões que foi levantando, pelas meticulosas revisões deste texto e sobretudo por me ter transmitido a "visão de túnel" sem a qual ainda estaria, certamente, a perseguir melhorias inúteis para o objectivo final.

À empresa onde trabalho, "EFACEC-Sistemas de Electrónica, SA", nas pessoas dos Eng. António Carrapatoso e Eng. Alberto Rodrigues, por acreditarem que vale a pena investir na investigação para que a indústria avance, espero sinceramente que este trabalho ultrapasse a recompensa pessoal da realização e que seja um contributo útil para a inovação dos nossos produtos.

Ao Dave e ao Viegas, por acreditarem que vale a pena pensar *software*, sem a sua arquitectura de componentes esta tese seria, inevitavelmente, outra.

Ao Ricardo, por ter participado activamente na construção do *framework*, foi um prazer enriquecedor trabalhar com ele.

A todos os colegas da divisão, que colaboraram e deram sugestões. Especialmente à Xana, ao Hugo, ao Paulo e ao Carlos por terem aguentado, pacientemente, o mau humor característico do "dia seguinte às noitadas de trabalho".

Um agradecimento muito especial à família e aos amigos, pelo apoio, compreensão e paciência.

E finalmente à Nicha, pelo amor, pela companhia, pela ajuda e pela paciência!

Abstract

For several reasons and since their appearance, the SCADA systems have always been a specific kind of software engineering. They normally are big and complex in the functionality they provide, but mostly, they have to follow very strict requirements in the matters of performance and availability. Traditionally, these proprietary applications are closed and difficult to inter operate with. Each software provider has its own, complete, SCADA system with a small degree of integration with others'.

It is, therefore, a very stabilized kind of software, somewhat insensible to broader changes, even with respect to the big revolution that the Web is operating on the software engineering in general.

In this thesis, we propose a different approach for the implementation of SCADA applications. We'll base ourselves in some of the Web principles, specifically, using "Web Services" as a transport layer in the communication between applications. This way, we'll try yo improve these applications in what concerns interoperability, portability and versatility in the geographical location of system' operators, but keeping in mind the strict requisites of this area.

We'll build a communications' framework, implementing a publish/subscribe paradigm, based on topics. We'll provide two transport implementations, the first one, based on a common middleware technology (RMI) and the other, based on Web Services (SOAP).

With this framework, we'll build an exploration set, with the necessary applications and scenarios to evaluate the fulfillment of the SCADA requisites.

The obtained results are satisfactory, both in the comparison between the Web and RMI implementations, as in the fulfillment of the referred requisites. This shows, that it is possible to adapt the traditional SCADA systems, to the, new era, Web technologies.

Resumo

As aplicações SCADA foram, por diversos motivos e desde o seu aparecimento, uma área especial da engenharia de *software*. Por um lado, são normalmente aplicações volumosas e complexas em termos funcionais, por outro, obedecem tipicamente a requisitos muito apertados de desempenho e de fiabilidade. São também, tradicionalmente, aplicações "fechadas" e pouco interoperáveis, sendo que, cada fabricante apresenta normalmente o espectro completo de aplicações que compõem um sistema SCADA.

Por estas razões, é uma área com alguma inércia à mudança, nomeadamente à grande revolução que a *Web* está a operar no modo de interagir com o *software* em geral e de o fazer em particular.

Nesta tese, propõe-se uma abordagem diferente para a construção de aplicações SCADA, introduzindo alguns conceitos relacionados com as tecnologias *Web* e mais especificamente com os serviços *Web* como meio de transporte na comunicação entre aplicações. Procura, desta forma, ganhar-se em termos de interoperabilidade, portabilidade e versatilidade na localização geográfica dos utilizadores, mas tentando manter-se os requisitos apertados que condicionam esta área.

Recorrendo a um paradigma de computação distribuída baseada em componentes, numa lógica de publicação e subscrição, desenvolveu-se um *framework* indicado para a construção de aplicações SCADA. São fornecidas duas alternativas para o transporte de eventos entre componentes, uma baseada numa tecnologia típica de *middleware* (o RMI) e outra baseada em serviços *Web* (SOAP).

Com base nessa arquitectura constrói-se um ambiente de exploração, com vista a avaliar o desempenho da solução *Web*, quer por comparação com a de referência, quer por comparação com os requisitos definidos no contexto SCADA.

Os resultados obtidos são satisfatórios, demonstrando, por um lado, que o desempenho de ambas as soluções não é, no essencial, muito diferente e por outro, que os requisitos SCADA em geral e os de latência e largura de banda em particular, são cumpridos com a solução *Web*.

Demonstra-se, assim, a possibilidade tecnológica de gozar de vantagens das tecnologias *Web* em sistemas SCADA.

Conteúdo

Abstract vii

Resumo viii

Índice de Tabelas xiii

Índice de Figuras xv

1 Introdução 1

1.1 Sistemas SCADA 1

1.1.1 Área de Interesse 2

1.1.2 Problemas Típicos 4

1.2 Importância da *Web* 4

1.3 Objectivo: Integração da *Web* com Sistemas SCADA 6

1.3.1 Motivação 6

1.4 Estrutura da Dissertação 7

1.5 Resumo 8

2 Sistemas SCADA 9

2.1 Arquitecturas Típicas para Sistemas SCADA 9

2.1.1 Funcionalidades em Detalhe 13

2.2 Requisitos de Sistemas SCADA 15

2.2.1 Dificuldades na Obtenção de Requisitos 15

2.2.2 Requisitos Típicos e Requisitos Limite 16

2.2.3 Enumeração de Requisitos SCADA 16
2.2.4 Casos Típicos de Operação 19
2.2.5 Tradução para Requisitos de Comunicação 20
2.3 A *Web* como uma Solução 23
2.3.1 Limitações dos Sistemas SCADA Clássicos 23
2.3.2 Objectivo: Colocar as Aplicações HMI na *Web* 23
2.4 Resumo 24

3 Tecnologias *Web* 25

3.1 Tecnologias *Web* 25
3.1.1 A *Internet* e a *World Wide Web* - Clarificação 26
3.1.2 HTTP — Protocolo Suporte 27
3.1.3 Serviços *Web* 30
3.1.4 Outros Protocolos 38
3.2 Linguagens de Programação 39
3.2.1 Arquitectura de Componentes 40
3.3 Aplicações SCADA na *Web* 40
3.3.1 Objectivo - TESE 40
3.3.2 Antevisão de Dificuldades 41
3.3.3 Receita Tecnológica 43
3.4 Resumo 44

4 Modelo de Computação Distribuída 46

4.1 Componentes Distribuídos 46
4.1.1 Componentes, O Que São? 46
4.1.2 Componentes, Publicadores e Subscritores de Tópicos 48
4.1.3 Ligações ponto a ponto 51
4.2 Framework comum de comunicações 53
4.2.1 Pacote `frk` 54

4.2.2 Pacote `cmp` . . . 57
4.2.3 Interface com o Pacote `comm` . . . 62
4.3 Resumo . . . 64

5 Implementações — RMI e *Web* **65**
5.1 Pacote `comm` . . . 65
5.1.1 Transporte de Eventos . . . 68
5.1.2 Identificação de Aplicações Remotas . . . 68
5.1.3 Conexões . . . 68
5.1.4 Sessões . . . 68
5.2 Implementação RMI . . . 69
5.2.1 Introdução Tecnológica . . . 69
5.2.2 Implementação . . . 71
5.2.3 Desenho do Pacote `comm.rmi` . . . 74
5.3 Implementação Baseada em Serviços *Web* . . . 74
5.3.1 Transporte de Eventos . . . 77
5.3.2 Gestão das Ligações por Eventos . . . 79
5.3.3 Identificação de Aplicações Remotas . . . 80
5.3.4 Separação — Servidor e Cliente . . . 80
5.3.5 Desenho do Pacote `comm.web` . . . 83
5.4 Configuração de Canais de Comunicação por Tópicos . . . 89
5.5 Resumo . . . 92

6 Exploração e Avaliação da Arquitectura **93**
6.1 Definição do Ambiente de simulação . . . 93
6.1.1 Cálculo das Grandezas . . . 94
6.2 Configurações de Teste . . . 97
6.2.1 Cenários de Teste . . . 99
6.2.2 Condições Físicas para os Ensaios . . . 100

6.3 Resultados 100
6.3.1 Latência 100
6.3.2 Largura de Banda 102
6.4 Conclusões 106

7 Conclusões **107**
7.1 Satisfação dos Objectivos 107
7.1.1 Opções Tomadas 107
7.1.2 Resultados Satisfatórios 109
7.2 Trabalhos Futuros 110
7.2.1 Oportunidades de Melhoria 110
7.2.2 Novas Funcionalidades 110
7.3 Conclusões 111

A Acrónimos **112**

Referências **114**

Lista de Tabelas

6.1 Resultados dos testes de latência 101

Lista de Figuras

1.1 Scada: Arquitectura Geral 3

2.1 Centro de Comando — CERN 10

3.1 Pilha de protocolos TCP/IP 28

3.2 Pilha de tecnologias dos serviços *Web* 31

3.3 A estrutura da mensagem SOAP 36

3.4 Solução *Web* para o canal — Diagrama de contexto 45

4.1 Pacote `frk` — Diagrama de Classes 56

4.2 Pacote `cmp` — Diagrama de Classes 58

4.3 Publicação de um evento — Diagrama de Sequência 60

4.4 Entrega de um evento a uma *Callback* — Diagrama de Sequência . . . 61

4.5 Interface `ComponentCommunication` 62

4.6 Configuração da lógica de publicação e subscrição em XML 63

4.7 Exemplo de um metamodelo em XML 64

5.1 Interface RMI básica — PUSH 71

5.2 Entrada de um novo publicador 74

5.3 Implementação RMI do canal 75

5.4 Transporte de um evento pelo canal RMI 76

5.5 Interface do serviço *Web* 81

5.6 Pacote `comm` — Classes comuns *WEB.server* e *WEB.client* 84

5.7 Pacote `comm` — Implementação *WEB.server* 86

5.8 Pacote comm — Implementação *WEB.client* 87
5.9 Novo publicador *Web* 88
5.10 Publicação de evento por parte do *WEB.client* 90
5.11 Publicação de evento por parte do *WEB.server* 91
5.12 Configuração dos tópicos da aplicação WebBridge 91

6.1 Classe PingCallback 95
6.2 Interface gráfica do Simulator 97
6.3 Tom & Jerry — Teste WEB 98
6.4 Tom & Tom — Teste RMI 98
6.5 Largura de banda — Configuração WEB_1 103
6.6 Largura de banda — Configuração WEB_2 103
6.7 Largura de banda — Configuração WEB_5 104
6.8 Largura de banda — Configuração WEB_10 104
6.9 Largura de banda — Configuração RMI 105

Capítulo 1

Introdução

Neste capítulo inicial, faremos uma contextualização dos temas suporte deste trabalho, os sistemas SCADA por um lado e a *Web* e respectivas consequências no *Software* em geral, por outro. Com base nesse contexto e explorando alguns problemas relacionados com o SCADA enquanto área da "Engenharia de *Software*", proporemos a *Web* como solução.

Finalmente, far-se-á uma descrição geral do conteúdo dos capítulos deste texto.

1.1 Sistemas SCADA

Supervisory, Control and Data Aquisition, o acrónimo SCADA poderá ser traduzido com fidelidade para "Sistemas de supervisão, controlo e aquisição de dados", que é, em si mesma, uma definição razoavelmente completa para estes sistemas.

Aprofundando um pouco o conceito, incluem-se nesta categoria, todos os sistemas de *Software*[1] que permitem supervisionar e controlar outros sistemas ou processos específicos, por exemplo: centrais de geração de energia (nucleares, térmicas ou hidráulicas), redes de distribuição (de energia, de gás ou de água), laboratórios complexos de pesquisa científica[2], entre outros. A uma escala mais pequena pode considerar-se a supervisão de processos fabris (controlo das máquinas, da temperatura, etc.), ou, para sistemas ainda mais pequenos, poderão considerar-se aplicações comuns de domótica com controlo e supervisão de aparelhos domésticos[3].

[1] Por sistemas de *Software* entenda-se um conjunto de aplicações colaborantes e potencialmente distribuídas por várias máquinas para fornecer uma ou um conjunto de funcionalidades.

[2] Tome-se para exemplo o CERN — *Centre Européenne de Resérche Nucleaire*, que é um cliente e fabricante de sistemas SCADA para consumo interno, assim como participante activo em consórcios para estabelecimento de protocolos abertos (ver [DS99]).

[3] Em rigor, como veremos, há um limite na dimensão do sistema e sobretudo na "localização" física do *Hardware* de aquisição a partir do qual os sistemas são considerados SCADA por oposição aos DCS - *Distributed Controlled Systems* (ver [Boy99]).

Não interessará, nesta introdução, aprofundar muito este conceito, essa discussão será feita no próximo capítulo, no âmbito da definição dos requisitos típicos destes sistemas. Interessa-nos, sim, constatar que estes sistemas são suportados invariavelmente em várias camadas de *Software*, que interactuam num dos extremos com *Hardware* de controlo e aquisição (sensores e actuadores de vários tipos, cartas de entradas e saídas, conversores A/D e D/A, protecções, etc.) e, no outro extremo, disponibilizam interfaces para os operadores com monitorização gráfica do estado do mesmo, visualização de alarmes, visualização gráfica de dados históricos, entre outras opções. Sendo que, a unir estes dois extremos, são necessárias aplicações de comunicação que implementam protocolos vários. Finalmente, é ainda comum conterem um repositório permanente de dados, que podem ser de configuração, históricos ou do estado actual do sistema, normalmente uma base de dados.

Apesar da natureza tão díspar destas camadas de *Software*, a característica que melhor as une é, sem dúvida, o grau de exigência a que estão sujeitas. Dito de forma simples, é exigido que o operador que controla um sistema, veja exactamente o que se passa nesse sistema em cada momento e que possa confiar (quase) absolutamente que pode efectuar com sucesso uma acção sobre o mesmo, como se estivesse no local da acção, por exemplo, abrir um disjuntor numa rede de distribuição de energia.

Noções como sistemas duplicados, sistemas em tempo real, sistemas disponíveis 99,X% do tempo são comuns nesta área e têm condicionado a evolução do *Software* que a suporta (estes e outros requisitos podem encontrar-se em [STE02] e em [LTE96]).

Finalmente, o *software* SCADA é, normalmente, composto de aplicações complexas e distribuídas!

1.1.1 Área de Interesse

Esta tese não se enquadra naturalmente em todo o espectro de camadas que compõem as aplicações SCADA, centra-se essencialmente no *Software* que "habita" um centro de comando. Ou seja, as aplicações distribuídas encarregues de fazer, por um lado, o tratamento e "interpretação" dos dados oriundos das camadas inferiores e que relatam o estado do sistema supervisionado, por outro, a disponibilização gráfica desses mesmos dados (devidamente tratados) para os operadores do sistema e, finalmente, o envio de mensagens de controlo para as camadas inferiores, se pedidas pelo operador.

Observando esquematicamente a arquitectura tradicional típica de um destes sistemas, de dimensão razoável, pode ver-se realçada a área onde nos focalizaremos (figura 1.1).

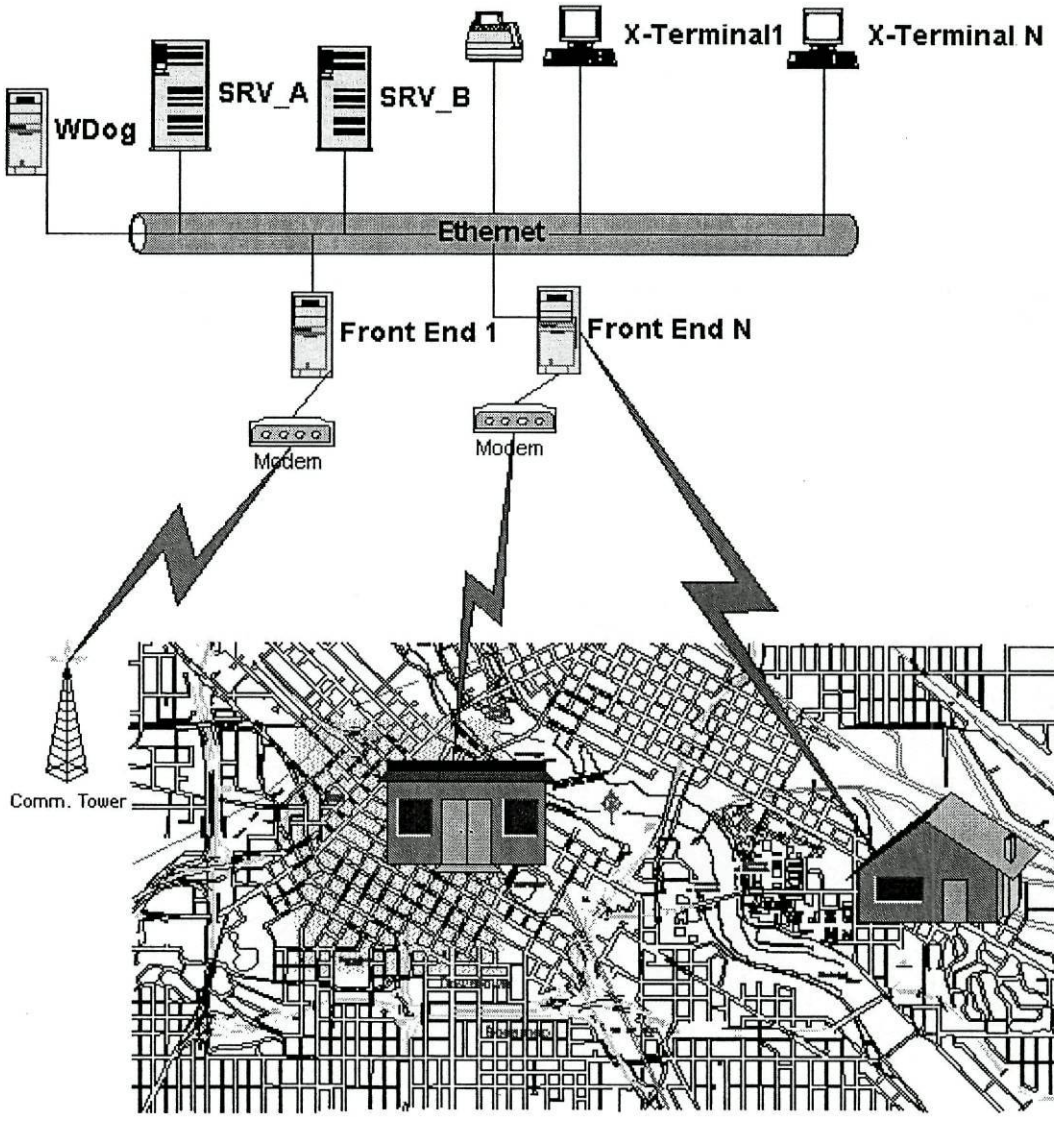

Figura 1.1: Scada: Arquitectura Geral

1.1.2 Problemas Típicos

Apesar das particularidades do contexto SCADA (algumas das quais já fomos aflorando), não deixa, naquilo que nos interessa neste trabalho, de ser uma área da **Engenharia de *Software***, com os problemas que lhe dizem respeito. Nomeadamente os processos de desenvolvimento e manutenção das aplicações, a evolução das mesmas para acompanhar a necessidade dos clientes em termos de funcionalidades e de escalabilidade, os custos das aplicações em geral e do licenciamento de *Software* de terceiras partes em particular, etc.

A este respeito, há algumas características tradicionalmente comuns às aplicações SCADA e que decorrem um pouco do mercado em que estão inseridas. Tipicamente o *Software* é vendido por fabricantes que fornecem também as camadas restantes da arquitectura, formando de alguma forma um *lobby* proteccionista que permite manter a marca em futuros contratos. Por outro lado, são normalmente aplicações suficientemente grandes e complexas para desencorajarem o desenvolvimento de sistemas completos por fabricantes "mais pequenos" e a tentativa de produzir pequenas peças de *Software*, que integrem com os sistemas maiores, é normalmente dificultada pelo uso de protocolos e tecnologias proprietários para as comunicações internas.

A principal consequência deste contexto, é o acentuar da resistência natural às mudanças tecnológicas e de arquitectura que foram surgindo em outras áreas do *Software*, nomeadamente com inovações nos padrões da computação distribuída (*Model-View-Controller* [SM02, pp 208-219], modelo de três ou mais camadas ver [Vin01]) e o aparecimento da *Web* como plataforma distribuída por excelência.

Como consequência destas limitações na interoperabilidade e do facto de serem sistemas complexos e evolutivos, os tempos de desenvolvimento são geralmente muito elevados, o que constitui um dos principais problemas com que os fabricantes nesta área se deparam actualmente.

1.2 Importância da *Web*

A evolução do *Software* nos últimos tempos tem sido enorme, motivada e sustentada em parte pela evolução do *Hardware*, quer a nível dos próprios computadores (a capacidade de processamento das máquinas duplica em média a cada 18 meses [Moo65]), quer pela evolução das tecnologias de comunicação (surge o conceito de comunicação sem fios, a Internet de banda larga [COM02]). Mas a "explosão" dos serviços WWW — *World Wide Web*[4] e a sua proliferação exponencial na *Internet*, foi a causa da maior revolução na forma de fazer aplicações, basta considerarmos o número de utilizadores de uma aplicação *Web* típica, a localização espacial das peças que compõem a aplicação (interface gráfica ou repositório de dados por exemplo) e,

[4]Os conceitos de internet, WWW, tecnologias *Web*, etc. são tradicionalmente objecto de diferentes interpretações; no capítulo 3 serão detalhadamente discutidos.

sobretudo, a adaptação das aplicações, tradicionalmente com estado (*statefull*) e com conexão permanente (*connection-full*), a protocolos sem manutenção de estado nem conexões permanentes como é o HTTP — *Hiper Text Transfer Protocol* ([BLFF96]).

Uma constatação óbvia, é a de que a disponibilização estática de documentos para consulta remota, foi substituída gradualmente pela disponibilização de serviços cada vez mais complexos, para clientes que variam entre o navegador (*browser*) comum e uma aplicação feita numa linguagem de alto nível (como por exemplo uma *applet* JAVA — ver [Vin01]).

A expansão da *Internet* a um universo cada vez maior de máquinas e a disponibilização de serviços úteis para os consumidores em geral, obrigou a que os fabricantes se preocupassem com que as suas aplicações (serviços) pudessem ser acedidas pelo maior número possível de pessoas e, consequentemente, pelo maior número de plataformas (sistemas operativos, linguagens, máquinas, etc.).

A consequência foi o aparecimento do conceito da **interoperabilidade**, que já existia (por exemplo com o CORBA, como se pode ver no artigo [SV02]), mas que ganhou assim uma dimensão inédita, permitindo, por exemplo, o uso de tecnologias diferentes para fazer diferentes módulos de uma aplicação (como ter uma interface gráfica em Visual Basic e a lógica de negócio em C++ ou JAVA). Levando este conceito mais longe, passa a ser possível integrar mais facilmente sistemas de fabricantes diferentes[5]!

A segurança foi o outro conceito que ganhou uma nova dimensão com a disponibilização generalizada de aplicações mais ou menos sensíveis como operações bancárias, compra e venda de produtos, pagamento de impostos ou serviços. Também aqui houve uma revolução grande, sendo que é quase impossível hoje encontrar aplicações na *Web* sem algum tipo de autenticação e segurança do "cliente" (ver [CB94]) e está relativamente banalizado o recurso a canais "opacos" encriptados com SSL - *Secure Socket Layer* (ver [FKK96]).

A questão de segurança não é uma questão menor no contexto SCADA, basta pensarmos na importância estratégica das redes de abastecimento de água ou de energia para uma região, ou ainda no perigo real, para bens materiais e vidas humanas, de algumas operações relacionadas com os mesmos.

[5] O uso de protocolos ASCII é fundamental na expansão real do conceito de interoperabilidade, mais concretamente o aparecimento do XML (*Extensible Markup Language*) e do seu uso pelo SOAP (*Simple Object Access Protocol*) são a base do novo conceito chamado serviços *Web* (como veremos no capítulo 2).

1.3 Objectivo: Integração da *Web* com Sistemas SCADA

As duas secções anteriores pretenderam fazer uma introdução contextual aos dois mundos que esta tese relaciona e que tem como objectivo integrar. Ou seja, um mundo composto por aplicações SCADA típicas e que têm as suas especificidades, nomeadamente no que diz respeito às restrições de robustez e fiabilidade. E um outro em que a interoperabilidade é a palavra de ordem e a distribuição efectiva dos componentes do sistema com afastamento geográfico real, é um facto.

Esta tese focalizar-se-á assim, na integração das vantagens das tecnologias *Web* em sistemas SCADA, nomeadamente a possibilidade de "entrar" e operar um sistema a partir de qualquer máquina ligada à Internet[6].

1.3.1 Motivação

É comum as aplicações SCADA terem tempos de vida elevados quando comparadas com outro tipo de aplicações. No entanto esta característica enfrenta cada vez mais obstáculos, por exemplo, a duração das plataformas historicamente consideradas estáveis tem progressivamente diminuído (isto comprova-se, por exemplo, observando as fusões que têm existido entre fabricantes UNIX outrora concorrentes como a Digital e a Compaq), por outro lado, o próprio mercado tem fomentado o alargamento destes sistemas a outras plataformas, a ponto de hoje ser comum ver SCADAS a correr em arquitecturas INTEL com Linux ou Windows [DS99]. Estes factores levam a que os fabricantes deste tipo de aplicações tenham hoje preocupações de **portabilidade** e **interoperabilidade** que não tinham anteriormente.

Por outro lado, os elevados tempos de desenvolvimento, com consequentes custos directos ou de perda de mercado, fazem com que haja uma necessidade de reutilização progressiva, integração e interacção fácil de e com módulos de *Software* de outros fabricantes.

Finalmente, os custos de manutenção de *Software* tradicionalmente monolítico, ou a falência de algumas tecnologias utilizadas há relativamente poucos anos atrás[7], levam a que seja importante considerar a utilização de tecnologias abertas e com tempos de desenvolvimento consideravelmente mais baixos.

Esta mesma tese poderá eventualmente ser válida para outro âmbito da Engenharia de *Software*, em que *ainda* se verifique o mesmo afastamento das tecnologias *Web*, mas

[6]Este enunciado inicial da tese será substituído por outro mais claro e concreto, depois de, no capítulo 2 se fazer uma abordagem mais detalhada do contexto SCADA em que nos vamos inserir e de no capítulo 3 se descreverem as tecnologias disponíveis no âmbito *Web*.

[7]Refira-se como exemplo a utilização de XMotif para a construção de interfaces gráficas que tem sido progressivamente descontinuada por parte de alguns importantes fabricantes de *software* como acontece com o 4GL *UNIFACE* da Compuware ([UNI02]).

por (de)formação profissional do autor, é no âmbito SCADA que este texto é escrito.

Resta acrescentar que, apesar da adaptação deste tipo de computação distribuída à *Web* não ser trivial, os fabricantes SCADA começam a anunciar algumas soluções com essa vocação, embora ainda claramente mais por pressões de mercado, do que por trazerem vantagens visíveis para os operadores dos sistemas (veja-se o exemplo da ABB em [Ohl02]).

Com esta tese, pretende-se abordar o problema da adaptação da computação distribuída tradicional (*middleware*), às tecnologias *Web*, tentando avançar uma solução para um canal de comunicação que consiga um saldo positivo na avaliação das vantagens e desvantagens dessa adaptação.

1.4 Estrutura da Dissertação

- Capítulo 2: *Sistemas SCADA*

  Neste capítulo, começa por aprofundar-se a caracterização dos sistemas SCADA, centrando-nos nos pontos que interessam ao objectivo da tese. Abordam-se arquitecturas típicas, descrevem-se os principais actores destes sistemas e o seu papel. Analisa-se ainda a evolução recente do *Software* nesta área e as tendências que se avizinham. Finalmente, faz-se um levantamento dos requisitos comuns a estas aplicações, com, sempre que possível, uma quantificação precisa dos mesmos para que mais à frente se possa fazer uma análise avaliativa objectiva.

- Capítulo 3: *Tecnologias Web*

  Aqui, entraremos mais profundamente nas possibilidades tecnológicas no âmbito da *Web*. Fazem-se algumas clarificações sobre conceitos base e descrevem-se as principais tecnologias à nossa disposição actualmente. Abordam-se ainda os problemas essenciais que se levantam na adaptação destas tecnologias ao nosso problema concreto e justificam-se algumas opções tecnológicas. Ainda na discussão das melhores tecnologias, condiciona-se a escolha a uma discussão paralela sobre o modelo de computação distribuída mais interessante no nosso âmbito e consequentes necessidades tecnológicas para o implementar. Finalmente, enuncia-se a tese na sua forma definitiva e avança-se com uma *receita* tecnológica que serve os nossos propósitos e que será a base da implementação do trabalho de validação desta tese.

- Capítulo 4: *Modelo de Computação Distribuída*

  Partindo da opção por um modelo de computação distribuída baseado em componentes, clarificam-se aqui todos os conceitos envolvidos, nomeadamente os de componente, tópico, evento, publicação, subscrição. Com base nesse modelo e sem especialização do ponto de vista do canal de comunicação, desenha-se uma arquitectura suporte do mesmo. O objectivo será o de conseguir uma

arquitectura comum que permita ter mais tarde (no capítulo seguinte) diferentes implementações ao nível do transporte de eventos mas sempre de forma transparente para as aplicações.

- Capítulo 5: *Implementações — RMI e Web*

  Neste capítulo apresenta-se o desenho e implementação do pacote de comunicações em duas versões, uma implementação RMI (para referência e como representativa do *middleware* tradicional) e uma outra baseada em serviços *Web*, que será posteriormente comparada com a anterior, para validação da tese.

- Capítulo 6: *Exploração e Avaliação da Arquitectura*

  Neste capítulo faz-se uso do *framework* construído para a definição de um ambiente de testes apropriado para as medições necessárias à avaliação dos requisitos definidos no capítulo 2. Definem-se papéis, configurações e aplicações a implementar para por em prática os cenários de teste que se exigem. Finalmente, executam-se os cenários de teste, capturam-se os resultados e faz-se a correspondente análise.

- Capítulo 7: *Conclusões*

  Termina-se a tese fazendo uma avaliação global da mesma, sumariando os passos mais importantes na sua definição e prova. Sugerem-se ainda alguns aspectos que poderão ser melhorados e outros acrescentados para aprofundar o alcance desta tese.

1.5 Resumo

Neste capítulo introdutório fez-se uma contextualização do problema que vamos abordar, com uma enumeração breve dos requisitos das aplicações SCADA típicas de centro de comando.

Explicaram-se também alguns dos problemas com que as empresas que desenvolvem estes sistemas se deparam como motivo para sugerir algumas alterações tecnológicas, nomeadamente a introdução de tecnologias *Web* nos modelos tradicionais.

Capítulo 2

Sistemas SCADA

Neste capítulo aprofunda-se inicialmente a caracterização dos sistemas SCADA centrando-nos nos pontos que interessam ao objectivo da tese. Abordam-se arquitecturas típicas, descrevem-se os principais actores destes sistemas e o seu papel. Analisa-se ainda a evolução recente do *Software* nesta área e as tendências que se avizinham. Finalmente, faz-se um levantamento dos requisitos comuns a estas aplicações, e, sempre que possível, uma quantificação precisa dos mesmos para que mais à frente se possa fazer uma análise comparativa objectiva.

2.1 Arquitecturas Típicas para Sistemas SCADA

Como foi referido no capítulo introdutório, esta tese e os trabalhos efectuados para a sua prova, incidirão apenas na camada de *software* vulgarmente conhecida por "Centro de Comando". Que pudemos ver globalmente enquadrada na figura 1.1 e que podemos examinar agora em detalhe na figura 2.1.

Esta especialização clara por um contexto não significa que não existam requisitos comuns às restantes camadas de *software* dos sistemas SCADA. De facto, a ordenação cronológica dos eventos, a garantia de que todos os eventos são reportados (nada se perde), a exigência de elevados tempos de disponibilidade das aplicações são, todos eles, requisitos que existem quer num centro de comando quer associados aos *scanners*[1] de uma unidade de aquisição.

Apesar desses pontos comuns, focalizar-nos-emos de agora em diante, apenas na área que nos interessa e que discriminaremos mais detalhadamente da figura 2.1[2].

[1] Uma das filosofias mais usadas na interacção com *Hardware* de aquisição de dados é conhecida vulgarmente como *scanning*, ou seja, a consulta periódica às entradas para descobrir diferenças para o estado anterior (uma descrição mais extensa pode encontrar-se em [Boy99, Unit 4 — *Real Time Systems*]).

[2] Com a cortesia da EFACEC — Sistemas de Electrónica, SA.

Note-se que, quer a arquitectura, quer a nomenclatura aqui apresentada, pode não coincidir exactamente com os sistemas SCADA de todos os fabricantes. Usa-se como base a escassa literatura que existe neste âmbito ([Boy99]), arquitecturas publicitadas por vários fabricantes desta área e o modelo da empresa "EFACEC — Sistemas de Electrónica, SA" enquanto fornecedor internacional deste tipo de sistemas (do qual a figura é um exemplo) e local de trabalho do autor, com evidentes consequências no ponto de partida para esta tese.

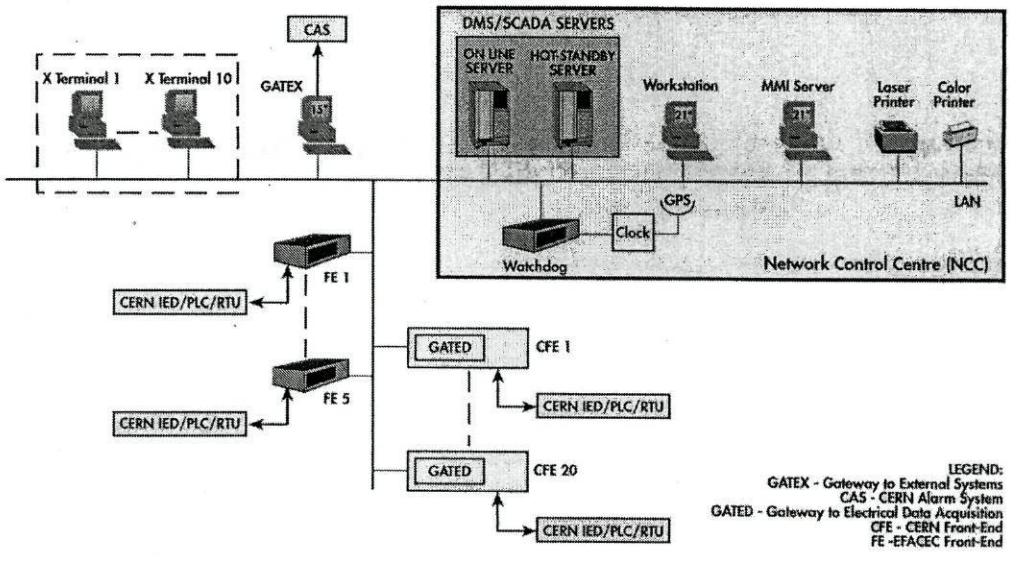

Figura 2.1: Centro de Comando — CERN

1. Unidades remotas (RTU - *Remote Terminal Unit*)

   Esta é uma das peças fundamentais dos sistemas SCADA, uma vez que é aqui que é directamente gerido o *hardware* de controlo e aquisição.

   Podem existir várias "unidades remotas terminais" no mesmo sistema SCADA (para um sistema grande podem ser centenas [Boy99]) e a sua função consiste essencialmente no seguinte:

   - adquirir em cada momento o estado dos equipamentos controlados pelas cartas de E/S;
   - transmitir essa informação para o servidor SCADA (MTU — *Master Terminal Unit* segundo a terminologia usada em [Boy99]);
   - executar comandos sobre esses mesmos equipamentos a pedido do centro de comando;

   As unidades remotas podem existir muito afastadas fisicamente do próprio centro de comando (dezenas ou centenas de quilómetros por vezes), sendo que o meio e protocolos de comunicação que usam para comunicar com este podem ser diversos. Comunicação série, rádio ou *Ethernet* como exemplos de meios comuns,

usando quer protocolos ANSI (como os IEEE), quer protocolos proprietários (como por exemplo, em Portugal o, "protocolo EDP").

2. *Front-End* de comunicações

   Também se poderiam chamar "concentradores de protocolos" porque uma das suas funções é concentrar e traduzir os diferentes protocolos falados pelas unidades remotas num só (o que facilitará a tarefa dos servidores SCADA), a esse protocolo chama-se por vezes "Meta-Protocolo".

   A segunda função é a de uniformizar os meios de comunicação, ou seja, comunica com as RTUs por diferentes meios (rádio, porta série, etc.) mas disponibiliza para os servidores os dados num só meio (*Ethernet* na maior parte dos casos), o que mais uma vez facilita largamente a tarefa, do ponto de vista das comunicações, aos servidores SCADA.

   Dependendo do tamanho do sistema, este papel pode ser desempenhado directamente pelo servidor SCADA (como um módulo deste), existir numa máquina separada para o efeito, ou ainda, como no caso representado na figura, ser desempenhado por diversas máquinas separadas fisicamente num sistema de múltiplos *front-ends*.

   No exemplo apresentado, pode ver-se que o centro de comando, além de controlar e recolher informação de *front-ends* EFACEC, comunica também com front-ends do próprio CERN.

3. *WatchDog*

   Mais uma vez, este papel pode não ser desempenhado por uma máquina fisicamente separada, mas por um módulo de software localizado em outras máquinas existentes (nomeadamente nos servidores[3]).

   Por um lado tem como responsabilidade saber em cada momento o estado de cada uma das outras máquinas que compõem o sistema, por outro, em sistemas com componentes duplicados, serve de árbitro na gestão de conflitos entre máquinas que podem assumir o mesmo papel. O recurso à redundância é comum nestes sistemas, nomeadamente nos próprios servidores SCADA, como veremos na descrição seguinte.

4. Servidor SCADA

   Os requisitos em termos de disponibilidade (quase) total do sistema ao longo do tempo, tornam praticamente impossível a sua implementação sem existir redundância em termos de papéis e de informação.

   Contornam-se assim problemas que ocorram por eventuais falhas de *hardware*, falhas do próprio *software* ou mesmo a necessidade de proceder à manutenção das máquinas, de Bases de Dados, proceder a reconfigurações, etc.

[3]Neste caso, e para servidores redundantes, é evidente que o *Watchdog* tem que ter mecanismos de desempate relativamente ao seu homólogo, que corre no outro servidor, o que pode tornar-se num problema complexo de computação.

Paralelamente à existência de máquinas com papéis redundantes é normal, para minorar os tempos de comutação, que o servidor que não está a controlar o sistema esteja num estado síncrono relativamente ao outro e, como tal, pronto a assumir o controlo num curto espaço de tempo (chama-se a esse estado "*HotStandBy*").

O servidor SCADA tem a seu cargo o tratamento imediato de toda a informação de estado do sistema supervisionado à medida que vai sendo recolhida, o que inclui:

(a) geração de alarmes (veremos à frente o significado do conceito no mundo SCADA);

(b) actualização das interfaces gráficas de monitorização (sejam elas diagramas de supervisão, listas com estados de entidades telemetradas, listas de alarmes ou gráficos em tempo-real);

(c) arquivo de estados de equipamentos e de ocorrências de eventos (*"journaling"* ou *"event-logging"*). Este arquivo pode ser periódico ou com base em regras;

É ainda aqui, que são traduzidas e processadas as acções dos operadores a partir das interfaces gráficas, nomeadamente na aceitação de alarmes ou no envio de controlos para os actuadores remotos.

Por outro lado, mantém a definição e configuração do próprio sistema a monitorizar. Por exemplo, se se tratar de uma rede eléctrica, a definição de todos os pontos de medida, entidades derivadas destas, acções disponíveis sobre o sistema (incluindo sequências destas acções).

Finalmente, é comum o servidor SCADA permitir uma análise da informação histórica arquivada, ou seja, uma análise da evolução do sistema que está a ser supervisionado utilizando listas de arquivos ou gráficos de evolução do estado histórico de entidades[4].

5. Posto de Operação

São interfaces com os operadores, já que é a partir destes postos que os operadores têm acesso às aplicações de supervisão e controlo do sistema pelo qual são responsáveis.

Como aplicações mais comuns, temos:

Listas de alarmes. A geração de um novo alarme corresponde normalmente ao aparecimento de um elemento novo na lista com um determinado nível de prioridade associado e o toque simultâneo de uma buzina no centro de comando.

[4]Em sistemas de grande dimensão, é vulgar estar associado ao SCADA um sistema de apoio à decisão relacionado com o contexto do SCADA, por exemplo um DMS (*Distribution Management System*) para redes de distribuição de energia. Esses sistemas permitem um tratamento analítico e de previsão que auxilia nas opções de gestão, por exemplo pela aplicação de algoritmos de aprendizagem para previsão de cargas no planeamento da evolução do sistema controlado.

Representações gráficas do estado das entidades do sistema. Podem ser representações tabulares com campos descritivos ou gráficas em sinópticos animados.

Gráficos de tendência. Podendo ser de tempo-real, a acompanhar a evolução de um ou mais pontos de medida, ou gráficos históricos sobre os dados arquivados.

Aplicações de edição (ou administração). Todas as operações de definição e administração do próprio sistema SCADA são feitas a partir destas ferramentas. Desde a definição dos próprios equipamentos de medida e mapeamento em entidades físicas (*hardware*), passando pela configuração e desenho das representações associadas a essas entidades (em sinópticos por exemplo) até ao estabelecimento das regras de arquivo e geração de alarmes para cada entidade.

2.1.1 Funcionalidades em Detalhe

Depois de uma descrição da arquitectura em termos do papel das diferentes máquinas que a compõem, e para melhor se perceberem os requisitos que apresentamos a seguir, faz-se agora uma descrição mais detalhada das funcionalidades e dos conceitos lógicos que estão associados ao centro de comando (as descrições que se apresentam estão de acordo com a pesquisa do CERN "O que é SCADA?" [DS99]).

Telemetria Define a possibilidade de se adquirir o estado de aparelhos remotos ou enviar-lhes comandos. A comunicação com esses aparelhos varia em termos de protocolos ou tipo de transporte (rádio, *Ethernet*, rede eléctrica entre outros).

Entidades telemetradas Correspondem ao mapeamento das grandezas físicas associadas aos equipamentos, em objectos lógicos que as representam no sistema SCADA.

Tipicamente são identificadas unicamente no sistema e têm associado um valor, que pode ser analógico ou digital. Por exemplo, a temperatura capturada por um sensor, ou a corrente eléctrica numa linha são exemplos de entidades analógicas; o estado de um disjuntor (aberto ou fechado) será uma entidade digital.

Além do valor que uma entidade tem num determinado momento, existe um conjunto de atributos que também caracterizam o seu estado. Como exemplos desses atributos temos a validade dessa medida, o cumprimento da gama de valores [5], o estado de alarme, valor imposto ou o estado de equipamentos a montante e dos quais esta medida dependa.

[5] A leitura do valor de uma entidade pode corresponder à gama de valores fora da qual o aparelho não garante a fiabilidade da leitura, por outro lado é comum permitir-se a definição de limites a partir dos quais a entidade entra em alarme, por exemplo porque está fora da gama de valores aceitável, do ponto de vista da operação do sistema, para aquela entidade.

Entidades derivadas Partindo das entidades telemetradas, é comum poderem derivar-se outras que são depois seguidas e tratadas, como se fossem efectivamente entidades telemetradas.

Correspondem à aplicação de uma fórmula de cálculo baseada em outras entidades (telemetradas ou derivadas) e permitem um processamento idêntico das suas mudanças de estado, nomeadamente no arquivamento, ou na entrada em alarme.

Um exemplo simples será a definição de uma entidade derivada com o valor de uma "potência activa" com base em entidades telemetradas com o valor da corrente, tensão e ângulo de fase, para o mesmo equipamento eléctrico.

Controlos Correspondem à representação lógica de acções que podem efectuar-se remotamente sobre os equipamentos que são controlados pelo sistema. Normalmente, podem ser agrupados em conjuntos ou sequências, para facilitar a operação do sistema. Por exemplo, para efectuar uma série numerosa de operações que tem que ser sequencial.

É comum poder associar-se a um controlo um "estado final" para uma entidade que lhe esteja relacionada, por exemplo, efectuar o controlo "abrir o disjuntor *A*" deve conduzir a uma situação em que a entidade que corresponda ao estado do disjuntor *A* esteja no estado "ABERTO". Desta forma é possível fazer a confirmação automática da boa execução do controlo.

Alarmes São um ponto essencial em qualquer sistema SCADA. É suposto que o operador seja avisado de qualquer situação anómala no sistema que supervisiona. Por exemplo, falhas na alimentação de uma determinada zona, medidas que ultrapassam os limites de segurança previstos ou qualquer outra informação que deva ter um tratamento (ou pelo menos atenção) especial, por parte do operador.

Os alarmes estão associados a entidades e aos valores e estados que estas vão tendo, sendo que têm níveis de prioridade (para facilidade de tratamento por parte do operador).

Finalmente e por uma questão do segurança, um alarme nunca desaparece da lista enquanto não for "aceite" pelo operador, garantindo-se assim que existe uma responsabilização do mesmo, por ter tomado conhecimento de uma determinada ocorrência.

Esta questão poderá parecer menor mas, se considerarmos por exemplo as implicações económicas de uma falha de abastecimento de electricidade por parte de uma empresa de distribuição de energia eléctrica, facilmente percebemos a importância do factor "responsabilização" nestes sistemas[6].

[6] O conceito de qualidade de serviço relativo à distribuição de energia eléctrica é hoje fortemente regulado legalmente sendo que existem, por exemplo em Portugal, normas que não só o definem como impõem compensações indeminizatórias aos clientes por incumprimento do distribuidor ([Sil02], [D1200], [D1000], [D1399]).

Áreas de responsabilidade Apesar de um sistema SCADA ser centralizador por definição, a separação de responsabilidades na operação das diferentes áreas, tal como aconteceria no terreno, é fundamental. Clarificando melhor, o acesso e controlo do sistema deve ser condicionado pelo conceito de área de responsabilidade a que o operador está ou não ligado, áreas essas às quais os diferentes equipamentos pertencem.

Este mecanismo decorre ainda da necessidade de responsabilização explicada na descrição dos "Alarmes". Por outro lado permite também que se reflicta a divisão lógica e funcional das diferentes responsabilidades dos operadores do sistema na operação do próprio SCADA, ou seja, reflecte a organização da própria empresa.

Em termos práticos, este conceito permite, por exemplo, que se autorize a um determinado operador a visualização e/ou aceitação de alarmes de uma determinada área e que se rejeitem essas mesmas operações noutra área pela qual o operador não seja *responsável.* O mesmo acontecendo, naturalmente, para as restantes aplicações de interface gráfica com o utilizador (visualização de sinópticos, de listas, etc.).

2.2 Requisitos de Sistemas SCADA

Apresenta-se a seguir a enumeração exaustiva dos requisitos que condicionam as aplicações que fomos referindo anteriormente, fazendo-se, sempre que possível uma quantificação dos mesmos.

Os requisitos serão orientados para a área que realmente interessa ao objectivo deste trabalho, ou seja, os requisitos que condicionam a quantidade de informação que "circula" entre os servidores SCADA e os postos de operação, incluindo características como os atrasos no seu transporte, entre outros.

Alguns desses requisitos serão traduzidos em termos de grandezas relativas à comunicação (parâmetros normais e parâmetros máximos), permitindo-nos mais tarde fazer uma melhor avaliação do cumprimento, ou não, dos objectivos a que nos propomos.

2.2.1 Dificuldades na Obtenção de Requisitos

A tradução de requisitos em números concretos que possam ser depois comparados objectivamente é, provavelmente, um dos problemas mais complexos em todos os trabalhos de investigação. Sobretudo quando as fontes não são necessariamente "científicas" ou quando não existe uma normalização dos sistemas que se pretende estudar.

Para os sistemas SCADA, este facto é tanto mais verdadeiro quanto estes sistemas são normalmente produtos comerciais, propriedade dos seus fabricantes. Como consequência disso, a sua especificação é normalmente a apresentada por esses mesmos

fabricantes, numa linguagem menos científica e mais "publicitária". Outra fonte de informação importante são os requisitos impostos pelos próprios clientes quando definem as regras dos concursos para a aquisição de novos sistemas.

A consequência desta situação é a de que todos os construtores vão evoluindo reactiva ou pro-activamente, mas em função do mercado e a informação objectiva de cada produto é escassa (que não a do *"marketing"*).

Não é por isso de admirar que uma das fontes para obtenção destes parâmetros sejam precisamente os cadernos de encargos relativos à venda de sistemas SCADA ([STE02], [LTE96]).

Para a definição de alguns parâmetros importantes em que a informação é ainda assim omissa, o autor reserva-se o direito de introduzir algumas características com base na sua própria experiência profissional.

2.2.2 Requisitos Típicos e Requisitos Limite

A primeira distinção que é necessário fazer e que aparece discriminada nas "descrições" destes sistemas é a das características de funcionamento normal, nas quais é suposto o sistema "aguentar" permanentemente sem que se deteriore[7] o seu comportamento.

Por oposição a estas, temos as características de funcionamento extremo ou vulgarmente, "picos de carga". Sendo que estas últimas correspondem a situações temporárias de carga extrema que podem provocar alguma deterioração momentânea no funcionamento do SCADA, mas das quais ele tem que recuperar sem intervenção nem perda de quaisquer dados.

2.2.3 Enumeração de Requisitos SCADA

2.2.3.1 Requisitos de configuração (ou estáticos)

1. Número de pontos de medida (mais de 100000):

   Corresponde ao número de entidades telemetradas que poderão fazer parte de um mesmo SCADA. Em Portugal, os maiores sistemas SCADA andam na ordem das dezenas de milhares de entidades, embora se preveja que, num futuro breve, este número aumente radicalmente (ver [RR02]).

2. Número de RTUs (mais de 200):

   Corresponde ao número de unidades remotas e pode ascender às centenas para sistemas grandes. No âmbito desta tese, este parâmetro torna-se irrelevante uma

[7] O sintoma mais típico de "deterioração" de um destes sistemas, consiste no atraso progressivo dos dados reportados relativamente à sua ocorrência real, que conduz a um desfasamento relativamente à realidade.

vez que só estamos preocupados com as transacções entre servidor SCADA e os postos de operação e vice-versa.

2.2.3.2 Requisitos não mensuráveis

1. Disponibilidade (99.9%).

   Este é um termo muito usado hoje em dia a propósito de serviços na Internet, nomeadamente nas aplicações de comércio electrónico, dadas as implicações económicas que têm os períodos de indisponibilidade das mesmas.

   Foi no entanto um requisito desde sempre associado às aplicações SCADA, também por motivos económicos (como já referimos a respeito das compensações indeminizatórias [Sil02]), mas sobretudo pela natureza quase sempre "sensível" dos sistemas e processos controlados com risco para pessoas e equipamentos nalguns casos.

   Este requisito está no entanto intimamente ligado ao desenho do próprio sistema SCADA, nomeadamente aos seus mecanismos de redundância e tolerância a falhas e não exclusivamente ao canal de comunicação entre aplicações. Daí que não se lhe possa dar uma resposta completa no contexto do canal de comunicações, a não ser no factor de condicionar algumas opções como sejam as tecnologias a usar ou o desenho das diferentes peças.

2. Não se pode perder informação.

   Esta é uma característica fundamental nos sistemas SCADA. Todos os eventos reportados têm que ser tratados, quer para controlo em tempo real dos processos, quer para posterior estudo do comportamento de determinados equipamentos.

   Ao canal de comunicações exige-se precisamente que seja fiável e garanta a entrega efectiva de todas as mensagens transmitidas.

3. A informação é tratada pela ordem de ocorrência.

   Ou pelo menos, pela ordem de relato da ocorrência por parte das camadas inferiores.

   Este é outro requisito que condiciona claramente o comportamento do canal de comunicações, uma vez que do ponto de vista de um par emissor/receptor de uma comunicação, tem que ser garantida a ordem na transmissão.

4. Portabilidade, escalabalidade e interoperabilidade.

   O tempo de desenvolvimento destes sistemas é normalmente longo por serem aplicações volumosas e complexas. Este facto obriga a que os fabricantes façam aplicações tão portáveis quanto possível, para evitar que, ao satisfazerem os requisitos de sistema operativo, *hardware* e servidor de bases de dados de um determinado cliente, percam competitividade relativamente a outros requisitos específicos para outros clientes. Como exemplo das necessidades de portabilidade

destas aplicações, refira-se que a maior parte dos fabricantes começou por ter sistemas a correr em VMS, UNIX e mesmo MsDOS e que, ultimamente, quase todos têm versões Windows ou mesmo Linux dos seus SCADA ([DS99]).

O preço destes sistemas (sobretudo no que diz respeito ao *hardware* de aquisição e controlo) obriga a que, por vezes, os clientes façam um investimento faseado na sua aquisição e isto implica, naturalmente, que a dimensão dos mesmos tenda a aumentar com o tempo. Até porque, a tendência do cliente é incluir cada vez mais elementos controláveis no seu sistema (ver [RR02]), justificando-se assim as necessidades de escalabilidade.

Finalmente, neste cenário de evolução constante quer por parte dos fabricantes quer por parte dos clientes, é exigida cada vez mais uma integração das diferentes aplicações à disposição do cliente ou seja, o conceito de gestão integrada tão popular na actualidade, também presente em [RR02]. Por exemplo, é normal que os dados históricos do SCADA sejam necessários para uma ferramenta de apoio à decisão usada pelo gestor de um sistema e é aceitável que essa ferramenta seja independente e até de um fabricante diferente do próprio SCADA. A interoperabilidade é, por isso, cada vez mais um requisito importante no desenho e implementação de qualquer sistema SCADA.

5. Interfaces gráficas ricas.

   A representação "rica" graficamente dos estados dos equipamentos é tradicionalmente uma preocupação nos sistemas SCADA, para mais simples utilização dos mesmos e por imposição do mercado concorrencial.

   Por outro lado, essas mesmas interfaces gráficas, além de terem que espelhar de forma simples e intuitiva a realidade do sistema controlado, têm também que permitir que se actue sobre ele. Isto implica que as aplicações de interface com os operadores, não sejam apenas receptores de dados relativos aos equipamentos visualizados, mas permitam também actuação sobre eles, com o consequente **envio de mensagens** de controlo para os servidores SCADA (aceitação de alarmes, envio de controlos, etc.).

   Finalmente, tal como com as aplicações gráficas tradicionais, é expectável que o refrescamento das entidades representadas seja automático e tão próximo da realidade quanto possível. Isto por oposição às aplicações *web* comuns em que é normalmente necessário que o utilizador provoque um refrescamento.

6. Sendo a *Web* um meio de comunicação instável por natureza, seria ainda desejável que o canal fosse auto-regulado e tivesse a noção da sua qualidade em cada momento!

2.2.3.3 Requisitos quantificáveis

1. Alarmes

(a) Número máximo numa lista: 2000.

(b) Atraso máximo na "chegada" do alarme à lista (depois de ser reportado no SCADA): 5 segundos.

(c) Tempo limite para a aceitação máxima de todos os alarmes da lista: 5 segundos.

2. Telemetria

(a) Variação média das entidades telemetradas[8]: 100/segundo.

(b) Picos de variações de entidades telemetradas: 200/segundo.

3. Número máximo de registos em tabelas de arquivos históricos: podem atingir vários milhões.

   Estes valores dependem fortemente da configuração e da manutenção destas tabelas históricas, baseiam-se em dados recolhidos informalmente em alguns clientes da EFACEC, nomeadamente no CERN e na CERJ (*Companhia Eléctrica do Rio de Janeiro*).

4. Limite máximo de registos históricos consultáveis em cada "vista": 2000.

5. Número médio e número máximo de entidades telemetradas visualizáveis em cada diagrama gráfico: 200 / 1000.

   É previsível que um operador possa ter mais do que um diagrama aberto ao mesmo tempo, o que fará crescer este parâmetro linearmente (com o número de diagramas abertos). Este parâmetro é muito importante porque permite que se faça uma gestão eficiente do canal de comunicação, fazendo circular apenas informação útil.

6. Número máximo de entidades visualizáveis em gráficos de tendência em tempo real: 8 (o período mínimo de amostragem é normalmente de 1 segundo).

2.2.4 Casos Típicos de Operação

Dado que esta tese se insere no contexto exclusivo do canal de comunicação no centro de comando, nomeadamente na comunicação que terá que existir entre os servidores SCADA e os postos de operação, far-se-á agora uma breve descrição de exemplos típicos do uso do sistema com análise da troca de mensagens que aí ocorre.

Operador faz login no sistema Tipicamente são trocadas mensagens de validação do operador, sendo que depois é enviada para o posto de operação a informação

[8]Este parâmetro corresponde ao número médio de variações para uma entidade telemetrada reportada para os servidores SCADA. Há normalmente uma filtragem fina na origem, de variações muito pequenas, para limitar o tráfego e processamento inúteis.

necessária para a construção da sua interface básica, ou seja, alguns dados gerais sobre o estado do sistema do ponto de vista da validade das máquinas, do número de alarmes em cada área, etc.

Neste cenário, a troca de mensagens não é grande mas exige-se alguma rapidez na mesma. Não é aceitável que o operador, depois de introduzir os seus dados, tenha que esperar muito mais do que alguns segundos para que a interface gráfica se estabeleça com o estado geral do sistema.

Operador abre uma lista de alarmes As listas de alarmes podem ser filtradas (por exemplo por área de responsabilidade) ou não filtradas de todo.

Dependendo de onde é que é feita a filtragem, esta questão condicionará o número de mensagens trocadas com o cliente.

Senão vejamos, se a filtragem for feita no cliente e a construção de cada lista for feita com base num conjunto comum actualizado pelo servidor, então não existe qualquer influência no número de mensagens trocadas com o servidor, quer a lista aberta seja filtrada, quer não seja.

Já se a filtragem for feita no servidor, para cada pedido diferente oriundo do posto de operação, podem circular para o posto de operação apenas os alarmes que o operador está interessado em visualizar. Claro que se forem abertas várias outras listas de alarmes baseadas em filtros diferentes, o servidor acabará por enviar muitas mais mensagens com alarmes "repetidos" para os postos de operação.

Admitindo o pior dos cenários e com base nos números apresentados na secção anterior, pode ser exigido ao canal que propague 2000 novos alarmes no espaço de 5 segundos.

Operador abre um quadro de rede Neste caso, o problema de comunicações é semelhante ao da lista de alarmes, mas com uma diferença de dimensão. Se olharmos para os números relativos às variações de entidades telemetradas e para o número típico de entidades por diagrama, chegamos à conclusão que podem chegar a um posto de operação várias dezenas de actualizações por segundo úteis (100/s). Isto considerando que o operador tem 3 ou 4 diagramas abertos com uma dimensão razoável e que só essas mensagens é que são enviadas para o posto de operação respectivo.

2.2.5 Tradução para Requisitos de Comunicação

Vejamos agora como é que estes requisitos todos, típicos do mundo SCADA, condicionam o canal de comunicações, impondo-lhe requisitos derivados desses.

2.2.5.1 Latência máxima

A latência é condicionada por vários dos requisitos descritos, sendo que terá que ser sempre menor do que os 5 segundos relativos aos atrasos máximos para alarmes ou telemetria.

Esta grandeza não pode ser significativamente alterada para mensagens que correspondam ao limite dos próprios 2000 alarmes. Isto implica que, idealmente, se os 2000 alarmes ocorressem praticamente ao mesmo tempo, deveriam ser todos transmitidos em conjunto e, por isso, com o mesmo atraso que um só alarme teria.

2.2.5.2 Largura de Banda

A largura de banda é a outra grandeza fortemente presente nos requisitos SCADA enumerados até aqui.

Existem imposições à largura de banda quando se prevêem, por exemplo, que os 2000 alarmes sejam transmitidos todos "ao mesmo tempo", sem aumento da latência. Existem também imposições à largura de banda quando se prevê um tráfego de telemetria para os diagramas abertos, da ordem das 100 mensagens por segundo.

Fazendo algumas contas, chegamos à conclusão que para a transmissão de 100 variações de telemetria por segundo para um posto de operação, a largura de banda utilizada não chega a 64kbytes.

Senão vejamos:

- Imaginando que se usam estruturas binárias para a transmissão das variações de telemetria e que essas estruturas ocupam 50 bytes.
- Considerando que se usa um protocolo básico de comunicação entre o servidor e o posto de operação e que o aumento de bytes transmitidos pelo próprio protocolo é de 10%.
- Considerando ainda que se usa o protocolo TCP/IP e que também esse introduz um acréscimo de 10% no número de bytes efectivamente transmitidos.

Temos que a largura de banda necessária é de:

$$100msg * 50b * 1.1 * 1.1 = 6050bytes/segundo \tag{2.1}$$

Se, por ventura, não se transmitirem estruturas, mas sim pedaços de texto legível com os mesmos valores e isso for traduzido em mensagens de 200 bytes (em vez dos 50 originais) e se, além disso, for usado um protocolo de aplicação mais esbanjador que use também ele texto na negociação da transmissão, contribuindo com um excesso de 100% sobre a transmissão original, teremos:

$$100msg * 200 * 2 * 1.1 = 44000bytes/segundo \tag{2.2}$$

2.2.5.3 Fiabilidade

Como podemos ver, o requisito de fiabilidade é derivado directamente de um equivalente para o sistema SCADA: o canal de comunicação tem que garantir a entrega de todas as mensagens enviadas.

2.2.5.4 Bidireccionalidade

O requisito de bidireccionalidade decorre claramente dos requisitos relacionados com a interacção fornecida pelas interfaces gráficas. É necessário que circule informação nos dois sentidos.

2.2.5.5 Segurança

Apesar deste tema não ter sido abordado enquanto requisito SCADA, faz todo sentido incluí-lo aqui, por causa das diferenças impostas pelas tecnologias ligadas à Internet.

A possibilidade, amplificada pela Internet, de se fazerem ataques do tipo homem-no-meio ("Man-In-The-Miedle" [Sch96, pp. 48-49]), ou outros que permitam, não só a consulta das mensagens transaccionadas, como também a alteração dessas mesmas mensagens, torna a segurança como um requisito muito importante.

2.2.5.6 Escalabilidade

O requisito de escalabilidade é adaptado directamente ao canal de comunicação; é necessário que o canal suporte não só os incrementos nas necessidades de "largura de banda" para transmissão de dados, mas também que se adapte à evolução dos conteúdos a transmitir, ou seja à evolução das próprias interfaces disponibilizadas pelas aplicações.

2.2.5.7 Portabilidade

O requisito de portabilidade obriga a que o canal de comunicação garanta a comunicação entre diferentes plataformas, de forma transparente para as aplicações.

Este requisito condicionará naturalmente as escolhas tecnológicas, nomeadamente no que diz respeito a protocolos e linguagens de programação.

2.2.5.8 Interoperabilidade

A interoperabilidade só poderá ser conseguida, ou pelo menos reforçada pelo canal de comunicação, se este usar protocolos de comunicação standard ou suficientemente divulgados, para que seja provável que outro sistema também "conheça".

2.2.5.9 Auto–Regulação

Apesar da evolução da própria Internet com um aumento generalizado da largura de banda disponibilizada a todos os seus utilizadores e uma melhoria significativa na qualidade dos acessos, é ainda assim um meio não fiável e instável de comunicação.

Daí que seja desejável que o canal de comunicação possua mecanismos de avaliação do seu estado e se possível capacidade de adaptação às condições instantâneas do mesmo.

2.3 A *Web* como uma Solução

Depois da introdução detalhada ao mundo SCADA e aos requisitos de comunicação entre aplicações, materializa-se agora a tese num primeiro objectivo para fazer face aos problemas que afectam os sistemas SCADA tradicionais.

2.3.1 Limitações dos Sistemas SCADA Clássicos

Como foi referido no capítulo inicial a propósito das motivações para esta tese (secção 1.3.1), há algumas limitações na arquitectura tradicional para os centros de comando, descrita na figura 2.1, e que seriam parcial ou totalmente resolvidas com a possibilidade de deslocar para a *Web* as aplicações que correm nos postos de operação, ou seja, as aplicações de interacção Homem–Máquina (HMI).

Esta ideia não é aliás original, sendo que alguns dos principais fabricante de *Software* SCADA começam a anunciar soluções baseadas na *Web* ([Ohl02]). O caminho parece, no entanto, estar a ser conduzido com base mais numa lógica de *marketing* do que numa lógica de mais valias concretas para este tipo de *software* e, em última análise, para o utilizador final do mesmo.

2.3.2 Objectivo: Colocar as Aplicações HMI na *Web*

Para resolver as limitações descritas na secção 2.3.1 e cumprindo os requisitos enumerados anteriormente para a comunicação entre postos de operação e servidores SCADA, propõe-se a construção de uma arquitectura que permita a implementação de aplicações HMI acessíveis pela *Web*. Pretende-se no entanto, que esta solução seja

suficientemente flexível, para que as mesmas aplicações transitem de forma simples, de um contexto *Web* (com as suas condicionantes de segurança, latência, desempenho, entre outras) para a rede local. Essa transição deverá ser tão simples quanto possível, idealmente deve resumir-se à configuração das aplicações, não obrigando a qualquer recompilação das mesmas.

O *Software* que tem suportado os sistemas SCADA tem características mais ou menos singulares, que o tornam numa área altamente específica da *Engenharia de Software*. Estas especificidades estão ligadas essencialmente à segurança, disponibilidade e fiabilidade destes sistemas.

É por isso expectável, que fenómenos recentes e evoluções na própria forma de fazer *Software*, tenham alguma resistência no mundo SCADA, por outro lado, essas mesmas revoluções podem não ser de integração "óbvia" num mundo em que a estabilidade e a fiabilidade são características essenciais.

Como adaptar este tipo de aplicações, em que o estado é uma característica fundamental, por exemplo a uma tecnologia tão eminentemente "sem estado" (*stateless*) como é o protocolo HTTP ([BLFF96])que suporta o mundo *Web*?

Esta e outras questões vão ser discutidas neste texto, depois de o objectivo aqui apresentado ser traduzido para um enunciado de tese mais concreto no próximo capítulo.

2.4 Resumo

Neste capítulo, depois de se aprofundarem alguns conceitos do mundo SCADA, enumeraram-se os requisitos típicos destes mesmos sistemas. Com base nestes últimos construiu-se uma base de requisitos de comunicação que terão que ser cumpridos paralelamente ao objectivo de colocar as aplicações HMI na *Web*. Este objectivo contextualiza a tese, mas não a enuncia, o enunciado da mesma virá depois da descrição do estado da arte, no que diz respeito ao mundo *Web*, que será feito no próximo capítulo.

Capítulo 3

Tecnologias *Web*

Depois de ter sido feita uma contextualização do problema que vamos abordar, de terem sido descritos, de forma mais ou menos exaustiva, os requisitos que teremos como referência no resto do trabalho e de ter sido justificada a necessidade de adaptar os sistemas SCADA ao mundo *Web*, entraremos agora mais profundamente nas possibilidades tecnológicas desta área. Enunciam-se os problemas essenciais que se levantam na sua adaptação ao nosso problema concreto e as opções tomadas, com a devida justificação.

Termina-se o capítulo com o enunciado da tese e uma *receita* tecnológica que será a base da implementação do trabalho suporte da mesma.

3.1 Tecnologias *Web*

A palavra *WEB* foi conscientemente inserida no objectivo a que nos propomos com este trabalho (ver 2.3.2) e sê-lo-á novamente no enunciado da tese que é feita no final deste capítulo (3.3.1). Justifica-se, por isso, que se faça uma clarificação desse e de outros conceitos, paralelamente à apresentação do estado da arte, no contexto das tecnologias disponíveis neste momento e nesta área.

A segunda nota que se impõe antes de se falar de tecnologias, é a constatação de que esta é uma área em profunda expansão; este facto é facilmente comprovável, por exemplo pela velocidade a que aparecem novos acrónimos ou extensões de outros existentes[1]. Neste contexto, um capítulo sobre o estado da arte tende sempre a ser um pouco fastidioso, procuraremos, por isso, cingir-nos às tecnologias realmente relevantes para o trabalho que vamos efectuar.

[1]Veja-se como exemplo o XML, cuja primeira especificação data de Fevereiro de 1998 ([HM01]) e seus derivados actuais: XHTML ([Pem00]), XLink ([CDD01]), XMLNS ([CDD01]), ebXML, XML-RPC, XSL, XSLT, SVG...

3.1.1 A *Internet* e a *World Wide Web* - Clarificação

3.1.1.1 Internet

Em 24 de Outubro de 1995, o FNC — *Federal Networking Council*[2] aprovou uma resolução que definia a Internet como um sistema de informação global que ([Lei00, *History of the Future*]):

1. está logicamente ligado entre si por um espaço global de endereçamento comum, baseado no protocolo IP - *Internet Protocol* ([RFC81]);

2. permite que as aplicações comuniquem usando a *stack* de protocolos TCP/IP ([Ste94]), uma extensão desta ou outros protocolos compatíveis com o IP;

3. disponibiliza e/ou usa de forma pública ou privada, serviços de alto nível assentes na estrutura de comunicações referida.

A definição de W. Richard Stevens ([Ste94]), distingue os conceitos de internet (com '*i*' minúsculo) e Internet (com '*I*' maiúsculo), afirmando que a primeira consiste num conjunto de redes interligadas e que partilham um mesmo tipo de protocolos e que a **Internet** consiste no conjunto de servidores interligados mundialmente (milhões) e que comunicam entre si usando a pilha protocolos TCP/IP.

O que se retém de ambas as definições e que nos servirá para o texto restante, é que Internet, está ligada à comunicação entre redes (e consequentemente entre máquinas), com tudo o que isso acarreta em termos de gestão de endereços, de domínios ou arquitectura de comunicações à escala mundial e, naturalmente, a pilha de protocolos TCP/IP desempenha um papel fundamental como base de toda a comunicação[3].

Se recuarmos um pouco no tempo (até 1962) chegamos às primeiras ideias de Licklider [Lei00, *Origins of the Internet*], sobre uma rede global de máquinas ligadas entre si e que permitisse a qualquer um a fácil e rápida consulta de dados e execução remota de programas que, como vemos, não anda muito longe daquilo que temos hoje à disposição na Internet. Isto porque, sobre essa estrutura gigante de comunicações, foram-se desenvolvendo protocolos aplicacionais para fornecer os mais diversos serviços.

---

[2] A entidade foi entretanto dissolvida e substituída nas suas funções por sub-grupos da CCIC — *Committee on Computing, Information and Communications* e pelo grupo LSN — *Large Scale Networking.*

[3] Este contexto está de acordo com os objectivos da comunidade de projectistas, operadores, fornecedores de serviços ou investigadores na área das telecomunicações em geral e das redes em particular, a IETF — *Internet Engineering Task Force* (ver [IET]). Nos seus objectivos está a garantia da melhor evolução arquitectural da Internet e a sua boa operação, criando para isso grupos de trabalho para discutir tópicos como roteamento, transporte, segurança... ou seja os temas que se inserem no âmbito da definição de Internet apresentada.

3.1.1.2 *World Wide Web*

O conceito de hipertexto é um conceito antigo, cuja característica fundamental corresponde, precisamente, à possibilidade de se "navegar" de um documento para outro (ou parte específica de outro), podendo o documento alvo, estar localizado numa máquina diferente.

A *globalização* desse conceito foi, sem dúvida, o motivo fundamental para o sucesso quase imediato da *"World Wide Web"*. E foi precisamente essa a primeira definição para um novo mundo, proposta pelo Tim Berners-Lee, nos seus esforços para convencer a gestão do CERN, segundo a qual, um "sistema de hipertexto global" seria do interesse da instituição (ver [BL89])[4].

Alargando este conceito para recursos genéricos, sejam eles documentos, dados, aplicações ou mais recentemente serviços, chegamos à definição actual para a *Web*, ou seja e segundo o próprio consórcio W3C, **um conjunto global de recursos** (veja-se no mais recente "esboço" da "Arquitectura para a *World Wide Web*"[BLBC+02a][5]).

Esta generalização e uniformização na "navegação" por recursos de tipos diferentes, é feita pela definição do conceito de URI — *Uniform Resource Identifier* ([BLFM98]), que define o modo comum para aplicações, ou utilizadores, referirem recursos. Incluindo-se nesse modo, toda a informação necessária para a identificação exacta do recurso pedido: protocolo aplicacional a usar (SMTP [Pos82], HTTP [BLFF96], FTP [PR85]), porto (TCP/IP) onde o serviço se encontra disponível no servidor[6] e, finalmente, a identificação do recurso propriamente dito. Eis alguns exemplos:

- `mailto:mpgomes@se.efacec.pt`
- `http://www.efacec.pt/index.html?with=arg1&and=arg2`
- `news:comp.infosystems.www`
- `ftp://se.efacec.pt/`
- `ldap://ldap.itd.umich.edu/c=GB?objectClass?one`

3.1.2 HTTP — Protocolo Suporte

Situando o protocolo HTTP — *Hyper Text Transfer Protocol*, na pilha de protocolos TCP/IP simplificada do modelo OSI, para 4 camadas apenas (de acordo com [Ste94],

[4] O nome provisório com que o autor da proposta baptizou o sistema foi "Mesh", tendo-o mudado mais tarde para *"World Wide Web"*.

[5] W3C — *World Wide Web Consortium*; refere-se aqui um documento que ainda não se encontra na versão final para realçar o carácter evolutivo da própria *Web*.

[6] A omissão deste "parâmetro" faz com que seja usado o porto pré-definido para o tipo de serviço pretendido (ex: 80 para o protocolo HTTP).

ver figura 3.1[7]), observamos que ele é, indubitavelmente, um protocolo ao nível da aplicação e que utiliza o TCP/IP para "transporte".

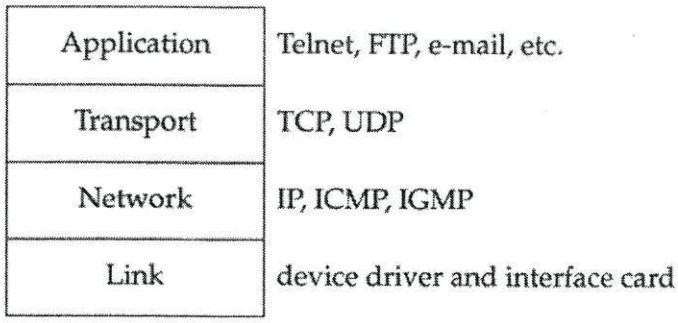

Figura 3.1: Pilha de protocolos TCP/IP

O protocolo HTTP, foi um dos principais responsáveis pelo crescimento exponencial da *Web* e ainda é, nos dias que correm, o mais utilizado na troca de informação entre máquinas da Internet. Isto porque, ele é essencialmente vocacionado para a obtenção de recursos estáticos, através de pedidos e respostas sem qualquer relação entre si (sem estado) e com total desacoplamento entre o cliente (que faz o pedido) e o servidor (que responde ao pedido enviando o documento pedido na resposta).

O servidor não mantém qualquer informação de acompanhamento histórico dos pedidos sucessivos de um mesmo cliente e a conexão não se mantém: é reatada para cada pedido.

Deve distinguir-se a versão HTTP/1.0 ([FG+99]) do protocolo, da versão HTTP/1.1 ([BLFF96])). A primeira introduziu os pedidos GET, POST e HEAD — GET e POST especificando duas formas diferentes de "pedir" um recurso e o HEAD para obter apenas o cabeçalho da resposta. A mais recente, acrescentou os métodos OPTIONS, PUT, TRACE, DELETE e CONNECT, prevendo a possibilidade de se manter a mesma conexão entre diferentes pedidos (método CONNECT), mas mantendo-se a inexistência de "memória" entre esses pedidos (os restantes novos métodos permitem a gestão remota dos recursos, acrescentar novos ou apagar existentes, usando o próprio protocolo HTTP).

Assim, na sua esmagadora maioria, as mensagens HTTP trocadas entre clientes e servidores, são pedidos GET ou POST e correspondentes respostas. É por isso e pela simplicidade do protocolo, que se explica que se consigam implementar servidores *Web* tão eficientes, capazes de responder a milhares de pedidos por minuto ([Fer02]).

A reter desta tecnologia:

1. o facto de a iniciativa para que haja comunicação ter que partir sempre do cliente, com um pedido;

[7]Imagem retirada de [Ste94].

2. depois da resposta enviada, o servidor não mantém qualquer informação sobre o estado desse cliente (referem-se na secção 3.1.2.1 algumas formas de contornar esta questão);

3. a conexão TCP/IP fecha-se depois do ciclo pedido/resposta terminado (exceptuando as utilizações raras da possibilidade de manter a conexão referidas para o HTTP/1.1 [BLFF96]);

3.1.2.1 Métodos para seguir a sessão com HTTP

Com a evolução dos portais da Internet no sentido de serem cada vez mais "personalizados" e permitirem, por exemplo, uma sequência de acções relacionadas entre si (ex: autenticação, escolha de um produto e ordem de compra), surgiram mecanismos para manter a noção da sessão e consequente manutenção de estado entre diferentes pedidos HTTP (ver [KM97]).

- Re-escrita de URLs

  A informação extra relativa à opção do utilizador numa determinada página e necessária para produzir a página seguinte coerentemente, é passada como "parâmetro" do pedido HTTP (o que leva à existência por vezes de URLs muito confusos com diversos parâmetros de sessão lá incluídos), por exemplo:
  `http://www.google.com/search?hl=en&lr=&ie=UTF-8&oe=UTF-8&q=HTTP%2F1.0+RFC`

- Campos escondidos

  São campos de formulários em HTML, que não aparecem ao operador, mas que, no entanto, estão lá e serão submetidos juntamente com as opções visíveis do formulário. Esta técnica permite, por exemplo, a geração de uma chave de sessão que é associada inicialmente à primeira "resposta" e que é devolvida depois pelo cliente nos pedidos seguintes.

- *Cookies* ([KM97])

  São ficheiros escritos na própria máquina do cliente, que permitem guardar informação sobre a sessão e que serão depois lidos nos pedidos subsequentes (obriga a que o navegador esteja configurado de forma a permitir que os "*cookies*" sejam escritos!).

- Sessões usando *Secure Sockets Layer* (SSL)

  Este protocolo utiliza chaves de sessão, sendo que, a chave da sessão do cliente é sempre enviada nas suas mensagens, o que permite a identificação de cada cliente por parte do servidor e a consequente possibilidade deste manter estado ([FKK96]).

3.1.3 Serviços *Web*

Apesar de ser possível encontrar várias definições para este conceito, o único aspecto, realmente polémico, reside no facto de se fazer incluir, ou não, o protocolo SOAP (ver secção 3.1.3.3) na própria definição. Ou seja, a questão reside em se é ou não concebível que existam serviços *Web* que não usam esse protocolo.

Relativamente a esta questão optamos por responder que sim, que é concebível embora, na prática, quase não existam e isto porque o SOAP se tornou o *standard* de facto para serviços *Web*. Nesse sentido, a definição deste tipo de serviços passa por serem um *framework*[8] de comunicação para troca de mensagens, em que o único requisito é que o serviço seja capaz de enviar e receber mensagens usando uma combinação qualquer de protocolos Internet *standard*. Sendo que a sua forma mais comum, corresponde à execução remota (estilo RPC - *Remote Procedure Call*) de um procedimento num servidor (onde o serviço *Web* reside).

A questão fundamental nos serviços *Web*, reside precisamente na definição das mensagens a trocar com o serviço e nunca nas tecnologias a usar por qualquer das partes, ou seja, qualquer combinação tecnológica (seja um PDA a correr *Palm* ou um computador pessoal a correr uma aplicação em JAVA no ambiente Linux), que consiga construir mensagens conforme a definição, que as compreenda e finalmente que suporte as tecnologias *standard* da Internet, pode ser fornecedor, ou cliente, de serviços *Web* (ver [STK02]).

Tendo feito esta distinção, resta acrescentar que o conceito nasceu e continua a evoluir com o protocolo SOAP, de que falaremos na secção 3.1.3.3.

Mas o conceito "serviços *Web*" não termina na definição do formato das mensagens, acrescenta duas novas áreas que estão muito ligadas ao contexto *Web*: a "descrição" (*description*) e a "busca" (*discovery*).

Podemos fazer um paralelo com a utilização tradicional da *Web*, ou seja a navegação que fazemos para encontrar documentos com conteúdos que nos interessem usando URLs. É comum usarmos um motor de busca para encontrarmos, com base em palavras chave, conteúdos interessantes; depois de termos alguns endereços, fazemos normalmente uma leitura superficial do documento encontrado (ou do resumo se existir) para verificar se é o que procurávamos. É exactamente isso que se passa com estas novas áreas dos serviços *Web*, ou seja, dotar os próprios serviços de mecanismos

[8]Opta-se pelo termo original em inglês, uma vez que não existe um equivalente em português. No âmbito do desenvolvimento de *Software*, a palavra *framework* corresponde a uma estrutura comum que facilita a construção de aplicações. Por exemplo, facilitando a construção de interfaces gráficas, a comunicação entre processos, *logging*, etc. Na maior parte das vezes inclui um conjunto de bibliotecas e documentação associada, que encapsulam sequências complexas de processamento. Alguns exemplos de *frameworks* de computação conhecidos: MFC — *Microsoft Foundation Classes*, jini (para facilitar a construção de aplicações distribuídas em JAVA), *Swing* (para aplicações gráficas em JAVA), PVM — *Parallel Virtual Machine* e MPI — *Message Passing Interface* (também para processamento paralelo), entre outros. Para alguns autores, o *framework* corresponde a uma receita (*pattern*) do desenvolvimento de *Software* (veja-se [MM97, pp. 149-152]).

para se auto-descreverem e "serem encontrados".

Pode observar-se esquematicamente a pilha de tecnologias associada aos serviços *Web* na figura 3.2 (retirada de [STK02]).

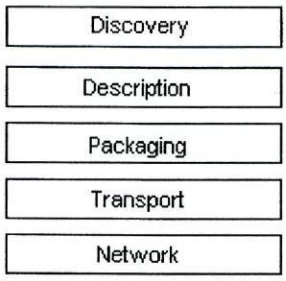

Figura 3.2: Pilha de tecnologias dos serviços *Web*

É interessante estabelecer uma comparação com a pilha de protocolos apresentada anteriormente (ver figura 3.1) e verificar o aparecimento das três novas camadas que já fomos referindo:

- empacotamento de mensagens (*packaging*), que abordaremos na secção 3.1.3.3;
- descrição (*description*), que abordaremos na secção 3.1.3.4;
- busca (*discovery*), que abordaremos na secção 3.1.3.4.

3.1.3.1 XML — *standard* "de facto" para estruturação de dados

O acrónimo XML significa *"Extensible Markup Language"* e nasce de uma necessidade, mais ou menos imperativa, que deriva da evolução da própria *Web*, sendo a palavra **extensível** (*extensible*) a palavra chave. Ou seja, é uma linguagem suficientemente simples mas extensível, para poder ser usada na estruturação de quaisquer tipos de dados a trocar entre aplicações na *Web*[9].

À medida que a *Web* evoluia no sentido da disponibilização de conteúdos com serviços inter-dependentes (antecipando o conceito de serviços *Web*), incluindo-se por exemplo num determinado sítio conteúdos obtidos directamente de outro sítio[10], começaram a aparecer cada vez mais aplicações cuja função era tratar, filtrar ou procurar conteúdos inevitavelmente formatados em HTML.

Como facilmente se depreende, o HTML não é uma linguagem interessante para a estruturação de dados (conteúdos), mas sim para a formatação dos mesmos (hipertexto). Por outro lado, goza de pouco rigor enquanto *standard*, o comportamento

[9] A linguagem surgiu de necessidades específicas de aplicações *Web*, o que não quer dizer que não se tenha tornado numa linguagem de estruturação de dados universal.

[10] Passou a ser comum encontrar referências a cotações de acções, às condições atmosféricas ou mesmo a notícias de última hora, em sítios que não têm essa informação original, mas vão buscá-la automaticamente a outros recursos da *Web*.

dos navegadores não é sempre idêntico face a alguns tipos de marcas, nem sempre é necessário "fechar" uma marca e os construtores, no passado, foram estendendo unilateralmente a linguagem original, utilizando novas marcas para novas situações.

No extremo oposto, o SGML — *Standard Generalized Markup Language* (do qual o HTML é uma especialização), apesar de genérico, é demasiado complexo (a sua especificação tem mais de 150 páginas), sendo rara a aplicação que a implementa na totalidade, o que leva a que aplicações diferentes usem sub-conjuntos diferentes da especificação e, consequentemente, sejam incompatíveis entre si (ver [HM01, *The Evolution of XML*]).

Assim se explica que, em 1996, um grupo em que se incluiam Jon Bosak, Tim Bray, C. M. Sperberg-McQueen e James Clark, tenha começado a trabalhar numa versão "leve" do SGML, chegando à versão 1.0 do XML em Fevereiro de 1998 (ver [Boy01]).

Depois dessa primeira versão, houve algumas evoluções importantes das quais destacamos:

- XML Namespaces (XMLNS - [W3C99])

  Veio resolver o problema, cedo detectado, de potenciais conflitos entre o nome dado às marcas de um determinado documento XML com as de outro incluído por esse (uma das características do XML é a possibilidade de num documento referir ou incluir outros).

- XSL — *Extensible Stylesheet Language* ([ABC+01])

  Dessa tecnologia derivaram a XSLT e a XSL-FO, sendo que a primeira permite definir regras de *transformação*, e a segunda de *formatação*[11]. Esta tecnologia é fulcral, por permitir uma separação forte entre a apresentação (formatação, vistas, filtros) e o conteúdo (enquanto dados estruturados em XML).

- XML Schemas

  Depois da primeira abordagem com DTD — *Document Type Definitions* ([HM01, cap. 3]), foi necessária uma abordagem que permitisse uma tipagem mais forte, ou seja, para lá da definição do bom encadeamento de marcas, era necessário também ter uma noção do tipo de dados a serem trocados, o que aconteceu com o aparecimento dos esquemas XML (XML-schemas).

A importância do XML tem-se revelado fundamental para ajudar a resolver o problema (já antigo) da interoperabilidade, mas que o avanço da *Web* tornou mais importante que nunca. Seria hoje impossível para um gestor de aplicações, pôr uma serviço de vendas disponível na *Web* sabendo que, por exemplo, os utilizadores de computadores *Mackintosh* não iriam poder utilizar o serviço. Como o XML define uma estruturação

[11] A combinação XSLT com CSS — *Cascading Stylesheet Language*, tem sido cada vez mais utilizada para construir HTML, por exemplo, para ser mostrado por um navegador, sendo que os navegadores mais recentes já processam eles próprios o XSL, construindo o documento resultado no cliente.

de dados em texto, com uma formatação estanque conseguida pelo uso de marcas para o início e fim para cada elemento, facilmente se percebe porque é que rapidamente surgiram diversas implementações para diferentes plataformas de *parsers* XML[12], sem ambiguidades ou incompatibilidades entre si (voltaremos a este ponto à frente na discussão sobre o uso de SOAP contra, por exemplo, o uso de uma tecnologia tão "amadurecida" como o CORBA, [SV02]).

3.1.3.2 Protocolos baseados em XML

Como o XML nasceu precisamente dessa necessidade de estruturação de dados, para troca de informação entre aplicações independentemente da plataforma em que cada uma corre, seria de esperar que rapidamente surgissem protocolos de comunicação baseados em XML e, necessariamente, orientados para a disponibilização de serviços (serviços *Web*, claro).

A evolução destes protocolos pode ser dividida em duas gerações (segundo [GSB+01, cap. 3: SOAP]), uma primeira intimamente ligada à primeira versão do XML (XML/1.0), que comunga naturalmente com esta as vantagens da sua simplicidade, mas também as desvantagens das suas limitações. E uma segunda geração marcada pelo amadurecimento do XML, que resolveu, como vimos, alguns problemas importantes como os da extensibilidade e tipagem (com os "namespaces" e os esquemas XML) e que se traduz, quase exclusivamente, no protocolo SOAP que abordaremos, dada a sua importância para este trabalho, na secção seguinte).

A primeira geração caracteriza-se, como vimos, pela simplicidade e marca o "arranque" dos serviços *Web*; realçam-se dessa geração dois protocolos que tiveram (e ainda têm) uma expressão importante:

- WDDX ([WDD])

  Web Distributed Data Exchange, ou, tentando traduzir, troca distribuída de dados na *Web*. Foi criado pela então "Allaire Corporation" (agora "Macromedia, Inc.") e define apenas uma forma de troca de dados em XML, pode ser usado sobre qualquer transporte e está ligado intimamente à troca de informação que existe por exemplo numa base de dados relacional[13].

- XML-RPC

  O XML-RPC está relacionado, como o acrónimo facilmente faz deduzir, com a invocação de um procedimento no servidor, sendo que os dados passados como argumento ao procedimento e o resultado da sua execução, são naturalmente

[12] A implementação mais usual é sem dúvida o "Xerces" com implementações de uso livre por exemplo para C++ ou JAVA.

[13] A conversão de dados de uma base de dados relacional para documentos XML foi, desde logo, abordada e tornou-se tão comum que os próprios fabricantes de bases de dados (como a Oracle por exemplo) fornecem procedimentos para ir buscar dados directamente formatados em XML.

formatados em XML. Este protocolo usa como transporte o HTTP[14] e está na origem do SOAP.

Com as limitações associadas à primeira versão do XML, também os protocolos da primeira geração se revelaram algo limitados, sobretudo no que diz respeito à sua deficiente extensibilidade. Quando foi necessário estender os protocolos para o transporte de dados binários, isso obrigou a uma alteração nas suas especificações, com consequente substituição das implementações, contrariamente à segunda geração de protocolos XML em que o problema da extensibilidade é resolvido com "namespaces".

O outro problema com estes protocolos está naturalmente ligado à outra limitação da primeira versão do XML, a falta de uma tipagem forte, pelo uso apenas de DTDs e não de esquemas XML, como acontece com a segunda geração.

3.1.3.3 O protocolo SOAP

Simple Object Access Protocol — protocolo (simples) para acesso a objectos. O acrónimo não foi provavelmente feliz uma vez que foca ainda o acesso a objectos como se tratasse da evolução de uma tecnologia procedimental (como o RPC — para chamadas remotas a procedimentos) para uma tecnologia orientada a objectos (como o CORBA — *"Common Object Request Broker Architecture"*) e não dá ênfase ao facto de ser essencialmente um protocolo de comunicação, suficientemente extensível para permitir qualquer tipo de troca de dados, mas ao mesmo tempo *interoperável*, sendo essa, como vimos, a palavra chave no contexto *Web*.

O facto do "gigante" *Microsoft*, ter estado na origem das primeiras especificações deste protocolo (SOAP 0.9 e SOAP 1.0 em Setembro e Dezembro de 1999, respectivamente), assim como o facto de outro "gigante", que é a IBM, ter participado na construção do SOAP 1.1, com evidentes melhorias relativamente às anteriores relacionadas com a extensibilidade e a remoção de alguns aspectos de tecnologia específica da Microsoft, explicam um pouco que uma tecnologia tão recente, tivesse uma aceitação e uma divulgação tão rápidas.

Como factos essenciais nesse alastramento rápido da tecnologia, retenham-se os seguintes ([GSB+01, cap. 3 - SOAP]):

- Em Maio de 2000 a versão 1.1 é proposta (pela Microsoft e pela IBM) ao consórcio W3C;

[14] É interessante verificar como o conceito clássico para camada de transporte e camada de aplicação da pilha de protocolos TCP/IP (ver figura 3.1) foi sendo progressivamente substituído (pelo menos em termos informais) por um conceito em que o transporte é agora associado a protocolos aplicacionais como o HTTP e o SMTP. Sendo esta evolução aceitável uma vez que, em termos lógicos, o conteúdo das mensagens deixa de estar necessariamente ligado ao protocolo original (hipertexto num caso e mensagens de correio noutro), para passar a estar ligado ao novo protocolo aplicacional que usa estes como transporte, ou seja, mensagens XML (dada esta explicação, passaremos também a usar esta abordagem).

- No mesmo ano a IBM doa uma implementação em Java-XML ao projecto *"Apache XML"* (`http://xml.apache.org`), para desenvolvimento pela comunidade *"OpenSource"*;
- A *Sun* reage começando a integração do protocolo com a sua plataforma J2EE ([SCD+01, cap. 22: *J2EE and Web Services*]);
- Em Setembro de 2000, o consórcio W3C anuncia a criação de um grupo de trabalho em protocolos baseados em XML, para resolver o problema potencial de fragmentação das implementações e para potenciar as excelentes oportunidades relativas a este tipo de protocolos, ainda em fase inicial de evolução (comunicado da W3C [BLBC+02b]);
- Este mesmo grupo, produz o primeiro esboço da versão 1.2 do protocolo em Julho de 2001;
- A especificação é dividida em duas partes, a primeira relativa ao "Framework de comunicação" e a segunda relativa a extensões opcionais ao protocolo base definido na parte 1, datando as propostas mais recentes, para ambas as partes, de Junho de 2002 ([W3C02a] e [W3C02b] para a parte 1 e 2 respectivamente);
- Em Novembro de 2002, o W3C divulga um esboço da "Arquitectura para os *Web Services*" ([CFNO02]);

Pela enumeração dos factos referidos, facilmente se depreende que o SOAP veio para ficar e, apesar de ainda não ser oficialmente um *standard Web* (o consórcio W3C ainda não divulgou uma especificação definitiva e oficial para a versão 1.2), na prática já o é e o envolvimento dos principais fabricantes de software nesta tecnologia, garante-a por si só.

Sem entrar na especificação do protocolo, apresenta-se na figura 3.3 a estrutura básica das mensagens SOAP (retirado de [STK02]).

Objectivamente, o que o protocolo SOAP nos fornece é:

- uma estrutura para a definição da "unidade mínima de comunicação", ou seja todas as mensagens neste protocolo vão dentro de um "envelope" com cabeçalhos (*header*) e um corpo (*body*), de acordo com o esquema da figura 3.3;
- um mecanismo para tratamento de erros que permite a identificação da origem e da causa do erro (noção de *"SOAP Fault"*);
- extensibilidade, pelo uso de *namespaces* e de esquemas XML;
- um mecanismo flexível para transmissão de dados, já serializados nalgum formato (XML ou outro), assim como uma convenção para a representação de tipos

[14] Retirado de [STK02].

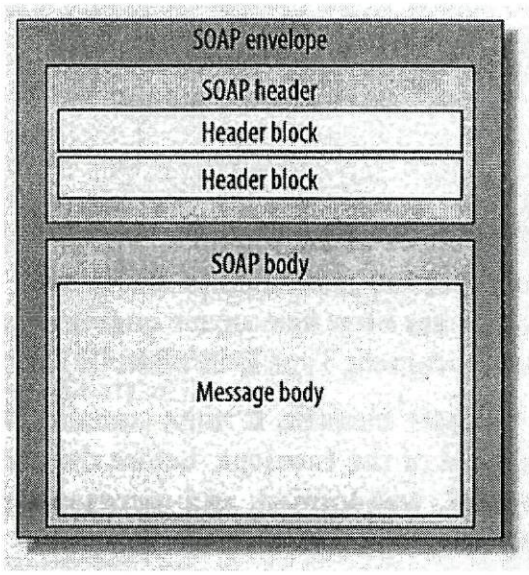

Figura 3.3: A estrutura da mensagem SOAP

abstractos de dados (como as estruturas de uma linguagem de programação, por exemplo);

- uma convenção para a definição de serviços tipo RPC (que são naturalmente a forma mais comum de computação distribuída);
- por oposição aos serviços RPC, quando os tipos de dados a transferir são tão variados, numerosos ou complexos que esse esquema não é apropriado, permite definir o serviço orientado à transferência de documentos XML;
- finalmente, fornece um mecanismo de adaptação a protocolos de transporte, sendo naturalmente o HTTP o mais usado (a especificação do SOAP não obriga a que assim seja).

Tentando enquadrar este protocolo no contexto SCADA, levantam-se logo à partida algumas questões (mais do que respostas):

1. Sendo as aplicações SCADA[15], aplicações com comunicação eminentemente bi-direccional, como é que se adapta a essa realidade um protocolo como o SOAP, que necessita de um "invocador" do serviço e de um "fornecedor" do mesmo?
2. Será necessário ter "servidores" SOAP em todas as máquinas?
3. Como é que um componente a correr no servidor toma, por exemplo, a iniciativa de informar uma lista de alarmes a correr remotamente num navegador, de um novo alarme?

[15] O sub-conjunto sobre o qual nos estamos a centrar, ver 1.1.1.

4. Como e quem é que mantém o estado das aplicações remotas?

5. O aumento de dimensão nas mensagens provocado pelos cabeçalhos SOAP, por um lado e HTTP por outro, não torna demasiado ineficiente a comunicação?

Por outro lado, facilmente se percebem também alguns elementos práticos que o uso desta tecnologia resolveria:

1. Todo o tratamento de comunicações é garantido à partida pela pilha de protocolos TCP/IP - HTTP - SOAP, basta implementar o serviço e invocá-lo por parte do cliente.

2. O uso de HTTP garante a facilidade de configuração de *firewalls* para permitirem a "passagem" das mensagens relativas à nossa aplicação. Note-se que este é um problema importante uma vez que, por motivos de segurança, é comum existirem filtros para a troca de pacotes TCP/IP que não sejam os conhecidos para serviço HTTP ou SMTP por exemplo.

3. Se e quando a segurança for um problema, o uso de SSL ([FKK96]) seria sempre uma opção, dada a sua fácil integração com o SOAP. Passaria a ser apenas um problema de configuração, mas com eventuais efeitos no desempenho do canal de comunicação.

   Veremos na secção 3.3.2.3 que esta não é nem a única, nem, provavelmente, a melhor solução para tratar a questão da segurança no nosso problema específico.

3.1.3.4 WSDL e UDDI — Descrição e busca de serviços *Web*

A descrição (*description*) do serviço refere-se à "descrição" da interface disponibilizada pelo mesmo, URIs disponíveis, formato das mensagens, protocolos usados no transporte, etc. O WSDL tornou-se a tecnologia mais usada embora haja outros como o RDF — *Resource Description Framework* da W3C ([RDF]) ou o DAML — *DARPA Agent Markup Language* ([DAM])).

A busca (*discovery*), trata da localização do serviço pretendido, implementando um mecanismo que permite ir buscar as "descrições" de serviços *Web* a um repositório bem conhecido (estilo índice global de serviços). A UDDI é neste campo a tecnologia mais usada embora, também aqui, haja alternativas. A *Microsoft* e a *IBM*, propuseram em conjunto uma alternativa ao UDDI, a *WS-Inspection* que simplifica a indexação dos serviços mais simples, tipicamente os mais comuns são descritos apenas por um endereço.

3.1.3.5 Conjugação de várias tecnologias — interoperabilidade

O conjunto de tecnologias que descrevemos na secção 3.1.3, forma uma aplicação prática do XML fortemente vocacionada para a implementação dos serviços *Web*, mais

ou menos complexos e encadeados, mas dentro de um contexto tipicamente *Web*, ou seja, associados normalmente a uma transacção estilo pedido – resposta e com origem, na maior parte das vezes, numa acção (consulta) de um utilizador.

Para este tipo de aplicações, os serviços *Web* têm-se revelado apropriados, sobretudo pela possibilidade simples de integrar tecnologias. É comum ter serviços implementados usando a implementação de Pavel Kulchenko em Perl do SOAP[16] a serem invocados por *applets* JAVA por exemplo.

Esse conceito de interoperabilidade, é a grande razão pela qual os próprios "programadores" de aplicações para a *Web*, aderiram tão rapidamente aos serviços *Web*. Desta forma, podem usar facilmente a linguagem que melhor dominam, ou que melhor se apropria, para cada módulo de um qualquer sistema. Este factor torna-se especialmente notório quando as aplicações são desenvolvidas numa arquitectura de três camadas, com uma separação clara entre lógica de negócio, repositório de dados e interfaces gráficas.

A questão que se coloca e que condicionará este trabalho, é a da possibilidade de se utilizar este tipo de tecnologias numa área perfeitamente distinta, em que as aplicações existem persistentemente, em que a computação distribuída já é uma realidade (embora usando tecnologias e protocolos menos inter-operáveis) e, sobretudo, em que as aplicações são concebidas numa lógica de máquina de estados e a sua interacção depende fortemente do seu estado e do dos outros componentes.

Usando um exemplo concreto, um operador deve conseguir monitorizar, na sua estação de trabalho, as diversas fases de actualização de um servidor SCADA sobre o seu servidor redundante; ou, noutro exemplo, quando um operador invoca um comando sobre o sistema supervisionado, é expectável que não o possa fazer (logo à partida) se o equipamento sobre o qual ele vai actuar não está válido no sistema ou se não tiver permissões para o fazer. Estes exemplos demonstram que a adaptação deste tipo de aplicações distribuídas, não é óbvia quando usando tecnologias *Web* ou, mais especificamente, serviços *Web*.

3.1.4 Outros Protocolos

- Jabber (`http://www.jabber.org/`)

  É um protocolo simultaneamente de "empacotamento" e de "transporte" e tem como característica essencial o facto de permitir a transmissão assíncrona de mensagens, o que lhe tem conferido alguns adeptos. Não é, no entanto, um *standard* e não usa o HTTP como transporte, o que lhe retira alguma da flexibilidade (de configuração) relativamente ao SOAP, por exemplo.

- BEEP (`http://www.beep.org/`)

[16] `http://soaplite.com/`.

Ainda está em fase de amadurecimento pela IETF, mas poderá vir a ser interessante, um vez que permite troca de mensagens bidireccional. Também incorre no problema do Jabber por não usar o HTTP como transporte.

3.2 Linguagens de Programação

Observando melhor os requisitos que definimos para o canal (secção 2.2), verificamos que, embora não seja um requisito explícito, não é possível prescindir do uso de uma linguagem de alto nível, quer no cliente, quer no servidor.

Relativamente ao cliente, a justificação é simples, é que o objectivo é *"... ter interfaces gráficas com o mesmo grau de interactividade..."* e funcionalidades gráficas que num posto de operação SCADA comum (listas interactivas de alarmes, sinópticos animados com possibilidade de execução de controlos, etc.). A implementação deste tipo de interfaces recorrendo a linguagens de "scripting" como "JavaScript" ou o "VBScript", seria uma tarefa muito mais complexa e com um resultado menos interessante e mais difícil de manter. Por outro lado, será de esperar que os fabricantes de software SCADA queiram reutilizar a análise e desenho que já tenham das versões originais dessas aplicações e, do ponto de vista dos clientes, não é aceitável nesta área de negócio que haja mudanças "radicais" na forma de operar o SCADA.

Relativamente ao servidor, parece mais ou menos evidente que é necessário implementar um ou mais componentes que façam a ponte entre os componentes remotos e os componentes locais (do sistema legado já existente), sendo da sua responsabilidade a manutenção de estado relativamente aos canais de comunicação activos, filtragem de mensagens (para reduzir o uso da largura de banda) o que parece ser uma tarefa mais simples numa linguagem de alto nível. Por outro lado, tendo um dos extremos implementados numa linguagem, há vantagens óbvias na utilização da mesma tecnologia no outro extremo, porque há uma boa probabilidade de poderem reutilizar-se módulos.

Justificada a necessidade de incluir linguagens de programação de alto nível, de preferência com orientação a objectos e com facilidade de integração com tecnologias *web*, enumeram-se agora as possibilidades mais evidentes.

- Tecnologia Sun (JAVA, RMI, J2EE)

  A característica mais relevante do JAVA no âmbito desta tese é a portabilidade. Caracteriza-se ainda pela facilidade de uso, excelente documentação e pela adequação à implementação de receitas típicas da orientação por objectos.

  O RMI enquanto tecnologia de computação distribuída é muito simples de usar e configurar quer no desenvolvimento das aplicações, quer na sua instalação (faremos uma descrição mais exaustiva do seu funcionamento na secção 5.2.1).

  O desempenho menos interessante é provavelmente a maior desvantagem desta linguagem.

É importante referir ainda as tecnologias ligadas à arquitectura J2EE, com a possibilidade da fácil implementação de componentes desacoplados de vários tipos, os EJBs (*Enterprise Java Beans*). Ainda na arquitectura J2EE, o serviço de mensagens JMS (*Java Messaging Service*) permite a implementação de uma comunicação por subscrição de tópicos (veremos no capítulo 4 a importância deste tipo de arquitectura na solução adoptada).

- Tecnologia Microsoft (C++, C#, COM)

  O ambiente *".NET"* responde à questão da portabilidade de uma linguagem com a fácil integração ou interoperabilidade entre linguagens, permitindo que se implementem diferentes componentes em diferentes linguagens com a comunicação simplificada. Esta solução não permite ainda assim o uso de outras plataformas (como o Linux) para correr essas mesmas aplicações.

  O desempenho do C++ e do próprio C# é várias vezes superior ao JAVA.

3.2.1 Arquitectura de Componentes

Apesar de este assunto fazer parte do estado da arte, dada a sua relevância para a arquitectura implementada ele será abordado de forma mais profunda no capítulo 4.

Para já, refira-se apenas que a utilização de uma arquitectura é fulcral para fazer face às questões que se levantam no desenvolvimento do *software* em geral e do *software* SCADA em particular ([MV01]). A opção por uma arquitectura de componentes, publicadores e subscritores de eventos de forma assíncrona, com recurso à receita genérica da *Callback* (referida por Vinoski em [Vin01] e descrita em [SM02, pp 238-247]), é um dos paradigmas disponíveis para a construção de aplicações distribuídas, revelando ter interessantes vantagens ao nível da simplificação do desenho e da implementação para as aplicações do contexto SCADA.

3.3 Aplicações SCADA na *Web*

Como pudemos ver até agora, o objectivo genérico a que nos propusemos levanta vários problemas, pelo menos mantendo como requisito a possibilidade de se correrem as mesmas aplicações de interface com o operador, na *web* ou na rede local, apenas mudando a configuração, mas podemos agora definir de forma mais objectiva e tecnológica a tese a que nos propomos.

3.3.1 Objectivo - TESE

É possível implementar um canal de comunicações que permita ter aplicações em pontos diferentes da Internet (por exemplo as interfaces gráficas separadas geografica-

mente do centro de comando com uma firewall no meio), mas cumprindo os requisitos descritos para aplicações SCADA. Pretende-se poder ter clientes computacionalmente complexos (implementados numa linguagem de alto nível) e com interfaces ricas, a comunicar com as aplicações que correm no centro do comando.

Paralelamente, pretende-se cumprir um conjunto de condicionantes derivadas dos requisitos definidos para as aplicações SCADA:

- as aplicações têm que ser multi-plataforma (devem correr em várias plataformas directamente ou com um trabalho de adaptação insignificante);
- as interfaces gráficas devem poder correr num navegador, ou fora dele, sem recompilações (ou seja apenas por configuração)[17];
- a implementação que usaremos deve ser compatível com um sistema de publicação e subscrição de eventos;
- deve ser possível comparar o funcionamento na *web* com o funcionamento local das aplicações ditas clientes, idealmente deve conseguir-se implementar a maior parte do framework de comunicações, independente da tecnologia a usar como "transporte", mudando-se apenas a camada de transporte por configuração. Dessa forma, as comparações de funcionamento e desempenho, traduzirão mais fielmente as diferenças que realmente interessam, ou seja, as decorrentes do uso da tecnologia *Web* para comunicação;
- finalmente, sendo o resultado final um protótipo a usar numa área industrial que tem, como vimos, um grau de exigência muito grande relativamente à qualidade do *Software*, pretende-se aplicar, sempre que possível, receitas bem conhecidas para problemas típicos (*patterns*), sejam elas receitas típicas de computação distribuída por componentes ([MM97], [GHJV95]) ou receitas específicas deste âmbito tecnológico ([SM02], [Mar02]).

3.3.2 Antevisão de Dificuldades

Nas secções seguintes encaram-se as principais dificuldades que se prevê, tenham de ser ultrapassadas, com este trabalho.

3.3.2.1 Manutenção do estado

Pretende-se ter computação de alto nível quer no cliente, quer no servidor, o que implica que poderão existir objectos persistentes durante a existência de cada componente. A manutenção de estado relativo à própria comunicação levanta problemas,

[17] Trataremos apenas a questão relativa ao canal de comunicações neste trabalho e não as implicações gráficas desta condicionante.

isto porque, o uso de serviços *Web*, faz com que a gestão das instâncias que atendem cada pedido (dos clientes) seja feita pelo servidor de *Web*, sendo que este serviço tem depois que comunicar com os componentes que correm no servidor.

É necessário implementar um mecanismo de autenticação da sessão de cada cliente e fazer no servidor uma associação com o estado respectivo, em termos de comunicações (tópicos publicados, tópicos subscritos, componentes no servidor interessados nos eventos dos componentes remotos, etc. ver secção 4.1.2).

3.3.2.2 FULL-Duplex

Esta questão parece ser das mais complicadas, sendo que aparentemente só há duas soluções para que o servidor possa tomar a iniciativa de enviar mensagens para o cliente:

- Todas as máquinas têm o pacote de serviços *Web* necessários instalado. O que é desinteressante do ponto de vista da flexibilidade para o cliente (que só deveria precisar de um navegador e de uma ligação à Internet) e das dificuldades de configuração, manutenção, etc.
- Simula-se o full-duplex por uma técnica semelhante à tecnologia "*pull*" sendo que o cliente faz ciclicamente pedidos ao servidor e na resposta a esses pedidos periódicos, o servidor "entrega-lhe" as mensagens que tem guardadas para si.

3.3.2.3 Segurança

Não existem muitas variantes para resolver esta questão com as tecnologias disponíveis hoje em dia, o uso de encriptação num canal seguro por "Secure Socket Layer (SSL)" é, sem dúvida, a forma mais comum e ao mesmo tempo a mais segura para a maior parte das aplicações *Web* actuais. No entanto, esta opção apresenta uma desvantagem importante: não permite que se encriptem apenas parte (ou partes) da mensagem, cada invocação de um pedido (por exemplo dum serviço *Web*), será, opcionalmente, ou totalmente encriptada, ou não encriptada de todo. Esta condicionante tira alguma flexibilidade a um canal de comunicação, que se pretende o mais eficiente possível, para fazer face aos requisitos apertados do contexto SCADA.

O **XML Encription** [IDS02], passou já em Dezembro de 2002, do estado de proposta para recomendação (*Proposed Recommendations (PR)*), para recomendação estável (*Recommendation (REC)*). A especificação desta tecnologia, parece ser muito interessante para o tipo de comunicação que se pretende implementar (veremos na "Receita Tecnológica", secção 3.3.3, que as mensagens a transaccionar entre aplicações serão sempre em XML). A tecnologia prevê a possibilidade de se encriptarem apenas parte da *stream* XML: elementos XML, ou mesmo conteúdos de elementos XML. Esta característica, permite que se façam esquemas muito simples para implementação da

segurança no canal. Podem, por exemplo, configurar-se os elementos XML que devem ser encriptados e, a partir daí, o canal só teria que substituir automaticamente os elementos originais pela sua representação encriptada.

Quer com uma tecnologia, quer com a outra, a sua utilização carece sempre de algum estudo do impacto no desempenho global do canal.

3.3.2.4 Requisitos de largura de banda

Pretende-se comparar resultados para diferentes situações de disponibilidade de largura de banda à partida, mas convém referir a este respeito, que os avanços tecnológicos em termos de aumento da largura de banda disponibilizada para um utilizador da Internet comum são notáveis.

Sendo hoje banal um utilizador ter à sua disposição 256kb ou mesmo 512kb por segundo, o que é várias vezes mais do que alguns dos tipos de comunicação feita tradicionalmente para sistemas SCADA (comunicação rádio ou ligações telefónicas de menos de 64kb por segundo)[18].

3.3.3 Receita Tecnológica

Como linguagem de programação, opta-se pelo JAVA, por uma razão essencial, a portabilidade. Assumiu-se que esse era um requisito para as aplicações SCADA modernas, o que condicionou a escolha da linguagem. Recorde-se que se pretendia utilizar uma linguagem de alto nível, orientada a objectos, com facilidade de integração das tecnologias de serviços *Web* e portável (o C# num framework *".NET"* cumpria todos os requisitos menos o da portabilidade).

Escolhendo-se JAVA para a linguagem de programação, é natural que a escolha para tecnologia de comunicação entre componentes na mesma rede fosse o RMI (*Remote Method Invocation*), implementando-se um canal com essa tecnologia para comparação posterior com a implementação *Web* do canal.

Finalmente, para "transporte" e empacotamento de mensagens opta-se por JAX-RPC, que é a implementação do protocolo XML-RPC (ver secção 3.1.3.2) que é disponibilizada no pacote de serviços *Web* distribuído com o JAVA (JWSDP — *Java Web Services Development Pack*), o que nos resolve todos os problemas relacionados com o transporte na *Web*.

A grande questão, associada a esta opção, pôs-se relativamente ao uso do protocolo SOAP (por exemplo recorrendo à especificação JAXM do mesmo pacote); as razões que levaram a preferir o XML-RPC (na implementação JAX-RPC) foram as seguintes:

[18] É, naturalmente, preciso entrar em linha de conta com a utilização, no contexto *Web*, de protocolos mais "gulosos" em termos de dimensão das mensagens como é o caso do SOAP sobre HTTP.

- Pretende-se que a invocação do serviço seja o mais rápida possível (por causa dos apertados requisitos SCADA) e esta implementação usa um *parser* SAX ao contrário da implementação SOAP, que usa o DOM e que é consideravelmente mais pesado que o primeiro;

- Foi assumido que a troca de mensagens na rede local, também seria feita usando eventos formatados em XML, o que obrigaria a que fosse da responsabilidade da aplicação o empacotamento e "desempacotamento" das mensagem em XML.

  Isto permitiu que se considerasse uma interface muito simples, quer para o serviço *Web*, quer para as invocações *RMI*: a passagem de um pedaço de texto para o outro lado (método "push ()"). Este mecanismo tem vantagens óbvias ao nível da escalabilidade e da extensibilidade, como veremos na discussão sobre a arquitectura de componentes (4.1.1.1) e facilita obviamente a adaptação entre os dois mundos que pretendemos misturar.

  Nessa situação, os problemas enumerados relativos ao XML-RPC comparativamente com o SOAP, perdem a importância e a facilidade de uso do primeiro aliada à sua rapidez de resposta levam-nos a optar por esse;

A comunicação entre o serviço *Web* e os componentes locais, faz-se novamente por RMI, ou seja, sempre que o serviço for invocado pelo cliente, são também invocados métodos RMI em componentes locais.

O objectivo final, será o de conseguir uma arquitectura de comunicação, que permita a localização dos diferentes actores de acordo com o diagrama de contexto da figura 3.4.

3.4 Resumo

Com os três primeiros capítulos, fica concluída a apresentação exaustiva do tema de tese devidamente contextualizado, a proposta de um trabalho que fundamente a tese apresentada e a discussão de algumas opções gerais como a escolha de tecnologias e arquitecturas a utilizar na sua realização. Os capítulos que se seguirão, mostrarão mais aprofundadamente o trabalho efectuado, ou seja o desenho e implementação do canal (SCADA) de comunicações.

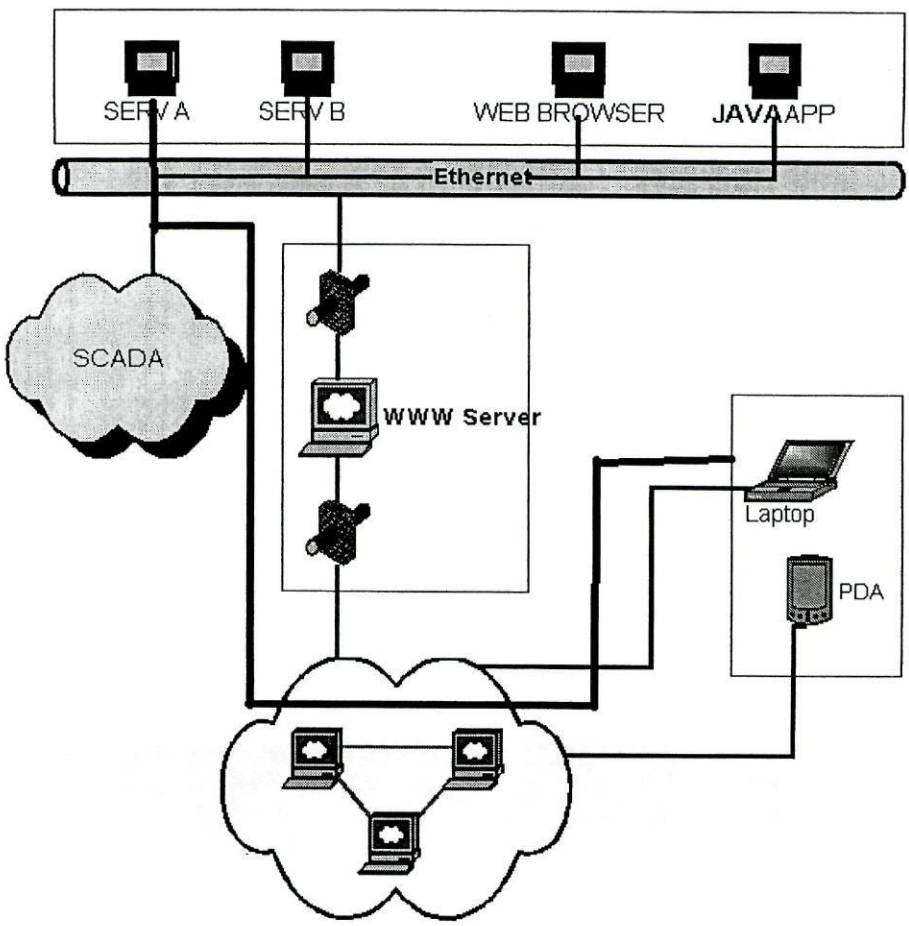

Figura 3.4: Solução *Web* para o canal — Diagrama de contexto

Capítulo 4

Modelo de Computação Distribuída

Depois de apresentada a tese, definido o trabalho a efectuar e escolhido o conjunto de tecnologias para a sua implementação, far-se-á agora uma descrição exaustiva do modelo de computação distribuída a usar, clarificando-se os diferentes papéis e conceitos envolvidos no mesmo.

Apresenta-se o desenho da arquitectura, independentemente do tipo de comunicação a ser usado, o que facilitará posteriormente a adaptação da mesma a diferentes tipos de comunicação (a tecnologia RMI, os serviços *Web* ou outro tipo de tecnologias no futuro).

4.1 Componentes Distribuídos

A arquitectura que se descreve nesta secção baseia-se nos conceitos usados pelo departamento de desenvolvimento de "Automação de Sistemas de Energia" da "EFACEC — Sistemas de Electrónica, SA", especificamente na arquitectura do "BUS" que é proprietária, mas cujos fundamentos gerais são do domínio público ([MV01]). Faz-se no entanto, um paralelo com a tecnologia de comunicação que faz parte da especificação J2EE 1.3 da Sun, o JMS — *"Java Messaging Service"* ([JMS02]) por serem duas abordagens muito semelhantes e ambas apropriadas ao tipo de computação distribuída que se pretende implementar.

4.1.1 Componentes, O Que São?

Do ponto de vista do sistema, os componentes são unidades funcionais do mesmo, ou seja, fornecem uma funcionalidade ou parte dela, colaborando com outros componentes para fornecerem o todo. Um sistema pode assim ser definido como um conjunto de componentes que colaboram entre si, para fornecer o pacote completo de funcionalidades, que se espera do mesmo.

Do ponto de vista de quem desenvolve, o componente é, em primeiro lugar, uma máquina de estados que é "alimentada" por uma interface bem definida, à custa de eventos (conceito que definiremos na secção 4.1.2.2). Desenvolver um novo componente, passa sempre por definir muito bem o seu comportamento em cada estado possível e a sua interface com o resto do mundo: que eventos recebe e que eventos publica[1].

Em termos práticos, os componentes existem dentro das aplicações (processos) que compõem um sistema, podendo as aplicações ser vistas como repositórios de componentes, à semelhança da arquitectura J2EE em que o servidor de aplicações é um repositório de EJBs, ou mesmo dos contentores *Web* enquanto repositórios de *servlets*.

Como a comunicação com o mundo exterior é sempre feita usando eventos via "BUS", um componente deve ser visto como um módulo que pode transitar entre aplicações sem alterações para o sistema global.

Finalmente, o canal de comunicação garante que a publicação de um evento é assíncrona, o publicador não bloqueia enquanto os subscritores não receberem ou processarem o evento.

A escolha dos componentes a quem "entregar" cada evento, é decidida pelo canal, com base na publicação e subscrição de tópicos por parte destes, como veremos à frente (secção 4.1.2.1).

4.1.1.1 Modelo PUSH

Um dos motivos de maior preocupação em aplicações distribuídas razoavelmente complexas, aparece com a "evolução" das comunicações entre aplicações, decorrente da evolução das próprias aplicações. Numa implementação clássica, em que se enviam estruturas binárias, essa questão é particularmente problemática uma vez que a instanciação do bloco recebido é perfeitamente livre (com um simples *cast*).

Primeiro o RPC (com o versionamento da invocação) e depois o CORBA (com a possibilidade de se passarem objectos como argumento), vieram minorar um pouco esses problemas. Obrigam, ainda assim, à recompilação consistente dos *"stubs and ties"*, e ligação (*link*) com todos os processos envolvidos, em duas situações:

- sempre que se alterar uma interface já existente, por exemplo para acrescentar, alterar ou retirar um parâmetro;
- sempre que se inserir uma interface nova ou apagar uma existente.

[1]Esta abordagem pode ser comparada, sem grandes concessões, com a sugestão de Selic e Rumbaugh para o uso de UML na modelação de sistemas complexos de tempo real (ver [SR98]), na qual os componentes correspondem a cápsulas (*"capsules"*), as *callbacks* correspondem aos portos (*"ports"*), o nosso BUS, enquanto agregador de canais de comunicação (RMI, *Web*, CORBA), corresponde aos conectores (*"conectors"*) e finalmente a interface de eventos, corresponde aos protocolos de sinais (*"signals"*) previstos por aqueles autores.

Esta questão não é menosprezável quando tratamos com sistemas suficientemente complexos, para terem múltiplas aplicações a correr de forma modular[2] (por exemplo numa arquitectura de componentes), quando se pretende permitir a interoperabilidade com sistemas de outros fabricantes e, sobretudo, quando o próprio sistema está constantemente em evolução, o que é muito comum para sistemas SCADA/DMS de dimensão considerável e tempo de vida longo.

Estes dois problemas são solucionados em parte com a adopção de dois paradigmas na comunicação entre componentes:

- o primeiro obriga a que só exista uma interface fixa, o "PUSH" que permite a um componente entregar um evento a outro[3];
- o segundo confere interoperabilidade e extensibilidade idênticas à que o SOAP trouxe aos serviços *Web*, ou seja, o argumento dessa interface única tem que ser um pedaço de texto formatado de acordo com um *standard*, que todos conheçam e possam interpretar – O XML é a resposta actual[4];

Além das vantagens referidas a respeito da extensibilidade e interoperabilidade, por exemplo com módulos de outros fabricantes, esta solução (ter só uma interface, definida computacionalmente como um método, com um argumento que é um "pedaço" de texto), é muito mais simples de adaptar a várias tecnologias. Permitindo-nos, a curto prazo, implementar no âmbito deste trabalho uma versão de referência baseada em RMI e uma versão baseada em serviços *Web*, mas mais tarde, estender esta arquitectura a outros canais de comunicação que sejam interessantes (voltaremos a este tópico nas sugestões para trabalhos futuros, secção 7.2).

4.1.2 Componentes, Publicadores e Subscritores de Tópicos

O mecanismo que faz com que uma mensagem (evento) seja, ou não, entregue a um determinado componente, corresponde à lógica de publicação e subscrição de tópicos. Assim, se um determinado componente está interessado em "ouvir" mensagens de um determinado tipo (tópico), tem que o subscrever, para as mensagens desse tipo lhe serem entregues. O contrário também é verdade, se um componente está interessado em "publicar" mensagens num determinado tópico, então deve registar-se (na arquitectura) como publicador desse tópico.

---

[2]É vulgar em sistemas SCADA de grande dimensão, existirem módulos opcionais sujeitos a esquemas de licenciamento e versionamento separados, ainda que oriundos do mesmo fabricante. Por exemplo, módulos EMS (*Electrical Management System*) a colocar "em cima de" um SCADA que supervisione uma rede eléctrica.

[3]Este é um dos fundamentos da "Arquitectura de Componentes" enquanto receita típica para aplicações CORBA ([MM97, cap. 8 - Component Architecture].

[4]É evidente que os dois extremos da comunicação têm que partilhar da mesma interpretação relativamente ao significado dos eventos que são transportados (se os eventos "obedecerem" a um XML-Schema conhecido por ambos, a tarefa fica facilitada).

4.1.2.1 Tópicos e Metamodelo

À semelhança da arquitectura JMS, o conceito de tópico será usado para que, na prática, o canal decida se um evento publicado por um componente deve ou não ser entregue a um outro componente. Mas ao contrário dessa arquitectura, é o próprio evento a chave dessa decisão, bastando o acto de publicar o evento para desencadear, de forma transparente, tudo o resto[5]. O algoritmo envolvido nessa acção pode resumir-se nos seguintes pontos:

1. um evento pertence a um tópico; essa informação é guardada num metamodelo — na nossa implementação será um documento XML a definir o próprio metamodelo (pode ver-se um exemplo na figura 4.7);

2. os componentes publicam ou subscrevem tópicos específicos, quer por configuração (ver figura 4.6), quer recorrendo à API de funções fornecida pela arquitectura;

3. durante a publicação de um evento, o metamodelo é consultado para a obtenção do tópico correspondente e o evento é entregue apenas aos subscritores desse tópico;

Desta forma, quem desenvolve aplicações não "necessita" de interagir directamente com objectos do tipo tópico; deve sim, definir um metamodelo de tópicos e eventos que sirva esse mesmo sistema e definir os publicadores e subscritores dos tópicos com base na *interface* pretendida para cada componente.

Este modelo, publicação e subscrição assíncrona de eventos sem recibos de entrega, serve uma boa parte da comunicação num sistema distribuído modelizado num paradigma orientado a objectos e modular; isso é especialmente verdade, nos sistemas típicos, que estamos a implementar.

Por exemplo, o comportamento de um componente que disponibiliza o estado das entidades telemetradas de um sistema SCADA, não deve ser alterado se o "cliente" desses estados for uma aplicação que faz cálculos de energia, ou se for uma aplicação HMI que mostra num diagrama, essas mesmas medidas. Por outro lado, também não deve alterar o seu comportamento se tiver um ou mais desses componentes gráficos, a "ouvirem" essas medidas. Na prática, a sua função normal deverá ser a de publicar continuamente eventos de telemetria, pertencentes a um tópico logicamente definido para o efeito.

Há, no entanto, situações em que este desacoplamento total entre "emissores" e "receptores" não é possível, por exemplo, quando um componente sozinho não implementa uma funcionalidade fechada, cooperando com outros para o efeito, ou ainda, para os

[5]Na tecnologia JMS, as mensagens não pertencem a um tópico, nem existe um metamodelo (fornecido pela arquitectura); a publicação num ou noutro tópico, é definida pelo publicador uma vez que é feita directamente no tópico (ver [JMS02]).

componentes que passam por um ou mais estados de "sincronização" de algum tipo, até atingirem o seu estado "normal". Nessas duas situações, é normal que sejam necessárias trocas de eventos dirigidos. Veremos na secção 4.1.3 como é que essa questão é abordada na arquitectura que pretendemos implementar (à semelhança da referida em [MV01]), comparando com a solução especificada pelo JMS.

4.1.2.2 Eventos

A unidade lógica mínima de informação, que dois componentes podem trocar entre si, é um evento, sendo que um evento é em primeiro lugar um objecto com estado, ou seja, com dados relevantes.

Os dados de um evento podem ser resumidos num conjunto de pares atributo/valor e combinações destes (como vectores de atributos, por exemplo).

O evento deve ter uma representação serializada em texto, para poder ser passado de acordo com o modelo PUSH (ver 4.1.1.1), sendo que os objectos computacionais que representam o evento têm que saber construí-lo a partir dessa representação textual e, por outro lado, saber gerar essa mesma representação.

Este conceito pode ser comparado ao da serialização de objectos em JAVA (que deu origem ao conceito de "Beans", ver [Eck00, cap. 13]), que permite uma flexibilidade muito grande na transferência de eventos entre componentes, uma vez que essa operação corresponde à serialização, transmissão (através de uma mecanismo muito simples para transmissão de texto) e desserialização no componente remoto.

4.1.2.3 *Callbacks*

A arquitectura encarrega-se de fazer chegar os eventos aos componentes interessados; a *Callback* é a "caixa de correio" que permite que os eventos que chegam sejam tratados pelos interessados.

Esta é uma receita de computação conhecida (utilizada desde há muito para diversas situações como eventos gráficos em X11, recepção de sinais em UNIX, temporizadores, etc.). A sua adaptação para o mundo dos objectos e da programação em JAVA especificamente pode ser encontrada em [SM02, pp 238-247]; esta receita de desenho está normalmente ligada a outra, a do "thread operário"[6] que descreveremos à frente na secção 4.2.

O componente interessado, regista um objecto (que implementa uma interface bem conhecida), para receber eventos específicos ou, alternativamente, todos os eventos de um tópico. A partir desse momento, o *Framework* chamará o método de processamento, sempre que chegar um novo evento para esse componente que cumpra o critério de entrega para essa *Callback*.

[6]Tradução livre pelo autor, do original em inglês: "*Worker Thread*".

Não existe qualquer restrição no número de *Callbacks*, que um, ou mais componentes da mesma aplicação, possam registar para a recepção do mesmo evento, sendo que ele tem que ser entregue a todas.

Esta filosofia é idêntica, por exemplo, à dos componentes EJB — *"Enterprise Java Beans"* da arquitectura J2EE ([SCD+01]), no sentido em que toda a computação é efectuada na implementação de *Callbacks* (no caso dos EJBs, os seus serviços são sempre invocados por aplicações exteriores, ou por outros EJBs). Este facto faz com que o processamento principal dos componentes não seja mantido por elas, mas sim pela arquitectura, sendo que apenas lhes compete implementar as *callbacks* de reacção, que fizerem sentido na sua lógica de negócio[7].

A arquitectura deve permitir a anulação do registo de uma *callback* por parte do componente.

4.1.3 Ligações ponto a ponto

Como já foi referido anteriormente, este modelo de computação, apesar de permitir a construção de sistemas muito modulares e facilmente extensíveis, não cobre todas as situações comuns do desenho de componentes como parte de um sistema. Em algumas situações é necessário menos desacoplamento entre o emissor de um evento e o receptor do mesmo ou, por vezes, é necessário relacionar a recepção de um evento com o envio de um outro, por exemplo como resposta a esse.

A implementação de protocolos é um exemplo claro desta necessidade, a necessidade dos componentes se "configurarem" com base em algo (por exemplo conteúdos de uma base de dados) é outro. Vejamos um exemplo:

- Um componente é responsável por validar os pedidos de entrada no sistema (*login*) por parte dos operadores. Na prática, é simples imaginar um componente destes a correr no servidor (onde encontra informação sobre os operadores e terminais autorizados) e os seus "clientes" a correrem nos postos de operação.

- Um cliente típico será uma caixa de diálogo para entrada no sistema. Corre no posto de operação e depois do operador introduzir os seus dados, enviá-los-à para o servidor para validação.

- Entretanto, o componente servidor recebe o pedido (num evento específico), valida-o de alguma forma e constrói a resposta, que envia de volta.

- Quando a resposta chegar ao componente cliente, este analisa-a e avança para o lançamento de outras aplicações, ou informa o operador do erro.

[7] A arquitectura tem sempre que fornecer um mecanismo de execução periódica de algum código (um temporizador), mas mesmo aí, os componentes são "acordados" numa *callback*, tal como na recepção de um evento.

O mecanismo descrito é banal, para múltiplas aplicações em que há um desacoplamento entre camadas de *software* (lógica de negócio separada da apresentação). O problema, que a arquitectura que descrevemos até agora não resolveu ainda, traduz-se na questão: *como é que a resposta a um pedido específico encontra o cliente que o pediu e só esse?*

A questão coloca-se, porque não é expectável que o componente servidor trabalhe em vários tópicos, um para cada potencial cliente e use eventos diferentes para clientes diferentes. Por outro lado, também não é interessante obrigar a que, cliente e servidor desta operação, implementem na sua lógica de negócio o mecanismo de identificação de pedido e respostas correspondentes (introduzindo um *token* no pedido que seria devolvido na resposta), isto porque, esta questão afecta muitos outros casos comuns num sistema, sendo portanto largamente vantajoso se a arquitectura resolver o problema à partida.

A especificação do JMS, permite a utilização de filas específicas para comunicação ponto a ponto (por oposição aos tópicos que são naturalmente "multiponto"). Aplicando esta solução ao exemplo que descrevemos, teríamos por exemplo, o servidor a "abrir" uma nova fila para cada diálogo específico com um dos seus clientes (uma solução em tudo idêntica ao servidor tradicional TCP/IP descrito pelo Stevens no clássico [Ste90, cap. 6 - Berkeley Sockets]).

A solução adoptada pela arquitectura do *BUS*, introduz o conceito de pedido (*Data Request*), muito semelhante à de um serviço *Web*, e de respostas direccionadas para o emissor de um pedido. Mantendo, desta forma, o desacoplamento entre componentes e orientando a questão para ligações lógicas, baseadas em serviços que cada componente pode fornecer.

4.1.3.1 Pedidos (*Data Requests*) e Respostas (*Data Replies*)

O *DataRequest* é uma variante do evento básico, na medida em que contém um evento e mantém as suas características de objecto transaccionável entre componentes. Introduz, no entanto, duas características novas ligadas ao conceito de serviço:

- o URN — *Universal Request Name* (não confundir com o conceito homónimo nas tecnologias *Web*), que identifica um serviço universalmente;
- o nome do pedido (*Request Name*), que serve para diferenciar, se for necessário, pedidos do mesmo tipo uns dos outros;

Assim, um servidor de um determinado tipo de serviço, regista-se como publicador do tópico respectivo (a que pertencem os eventos relativos a esse serviço) e regista uma *callback* para *DataRequests* identificada por um URN. No exemplo que demos anteriormente, o componente servidor poderia, por exemplo, definir um serviço identificado como "LOGIN_SERVICE" e registar uma callback para *DataRequests* desse tipo.

O cliente, quando pretende utilizar um serviço, subscreve o tópico respectivo, constrói um *DataRequest* com o URN correspondente e envia o pedido. Note-se que esta publicação é idêntica à de um evento usando o modelo PUSH, mas com uma lógica de publicação/subscrição inversa, ou seja, para que o cliente possa recorrer a um serviço, tem que subscrever um tópico, o que lhe permite receber eventos e publicar pedidos (*DataRequests*).

Finalmente, o servidor tem na arquitectura a possibilidade de publicar respostas relacionadas com cada pedido que recebe, sendo que essas respostas só serão entregues ao invocador do pedido.

Este mecanismo confere grande flexibilidade na localização dos serviços, na mesma aplicação, em aplicações diferentes, ou mesmo em máquinas diferentes e resolve a limitação descrita anteriormente.

4.2 Framework comum de comunicações

Descreve-se agora, o desenho e implementação do *framework*, executado no âmbito desta tese. Neste capítulo, tratar-se-á apenas dos módulos comuns a todos os canais de comunicação, realçando-se a interface que as implementações específicas terão de implementar. No próximo capítulo, entraremos então nos pormenores da implementação de referência em RMI e, naturalmente, da implementação *Web* como objectivo último desta tese.

Esta separação clara entre implementação da lógica de negócio, relativa à arquitectura de componentes descrita na secção anterior e as diferentes implementações tecnológicas do "modelo PUSH" (ver secção 4.1.1.1), é uma opção condicionada em primeiro lugar, por ser uma boa prática em termos de desenho (a separação modular dos problemas), mas também por ser imposta pelos requisitos do nosso problema. Pretendia-se poder, com este canal de comunicações, colocar a mesma aplicação a correr na *Web* ou numa rede local sem grandes alterações, de preferência apenas por configuração (ver secção 2.3.2).

Ora, a melhor forma de conseguir essa separação, é ter uma abordagem orientada a objectos, com a maior parte da funcionalidade do canal, implementada nas camadas comuns e apenas o estritamente indispensável implementado nas camadas específicas, sendo que mesmo estas, obedecendo a uma interface rígida, imposta pela camada superior.

Em termos concretos, o que se pretendeu foi separar, claramente, tudo o que tivesse a ver com o arranque e paragem de uma aplicação, a ligação da aplicação aos componentes, o metamodelo, os tópicos (incluindo publicação e subscrição), os eventos e *datarequests* e naturalmente as *callbacks*. Separar todos estes aspectos para que fossem tratados de forma comum (nos pacotes `frk` e `cmp` como veremos à frente, secções 4.2.1 e 4.2.2) e as diferentes implementações do "PUSH" ficassem noutro pacote (o pacote

`comm` que abordaremos no próximo capítulo).

Para além dos aspectos já referidos, há nesta separação, duas vantagens particularmente importantes para o nosso trabalho; especificamente:

- a comparação final entre implementações, será menos adulterada pelas próprias implementações, uma vez que o código que corre é, em grande parte, comum;
- se se pretender expandir o *framework* para outros tipos de comunicação, a tarefa será simplificada, porque basta implementar uma interface bem definida sem preocupações com a "lógica" de negócio *publish/subscribe*[8];

Resta acrescentar uma última nota, relativamente ao desenho e implementação efectuados no âmbito deste trabalho. Sempre que foi possível, foram usadas receitas (*patterns*) conhecidas para a implementação de partes do *framework*, sendo por isso natural, que as principais referências deste e do próximo capítulos sejam, precisamente, a receitas estruturais e de sistema na literatura apropriada.

4.2.1 Pacote `frk`

Este pacote trata de questões gerais relativas às aplicações, independentemente dos aspectos relacionados com os componentes e a sua lógica de negócio. Trata-se, tanto quanto possível, de um pacote independente do modelo de componentes e que poderia vir a ser utilizado por qualquer outro.

Os seus objectivos fundamentais são:

`Fornecer uma forma comum para as aplicações arrancarem e pararem.`

Este aspecto, apesar de simples, tem a vantagem de levar a que os diferentes programadores estruturem o arranque e paragem das suas aplicações da mesma forma (alocação e limpeza de recursos por exemplo), mas também permite que o próprio *framework* proceda à sua própria inicialização, o que será muito importante para as camadas de comunicação, como veremos;

`Fornecer uma API para que a aplicação faça` *`logging`*`.`

O facto de todas as aplicações fazerem *logging* utilizando a mesma API tem a vantagem imediata do encapsulamento, que permite que a sua implementação mude se for necessário, mas também que os *logs* produzidos obedeçam todos a um formato comum (facilitando naturalmente a sua análise);

[8] É evidente que a implementação de um canal para um sistema legado, que não use este modelo (publicação e subscrição), obrigará sempre à implementação de um módulo adaptador, dessa lógica para a do sistema.

`Fornecer mecanismos genéricos de configuração das aplicações.`

Sendo os ficheiros de configuração baseados em URLs, serão válidos para aplicações na *Web* ou não. Os primeiros "clientes" destes mecanismos serão os outros pacotes do próprio *framework*, uma vez que, como veremos, quer a publicação/subscrição de tópicos, quer a escolha do tipo de canal a utilizar por cada tópico, serão feitas por configuração;

`Implementar o ciclo principal de processamento.`

Finalmente, um dos aspectos mais importantes que este pacote "resolve", é o da implementação do ciclo principal de processamento, com o despacho de todos os eventos, à medida que vão chegando a uma fila comum. Este mecanismo é implementado de forma a permitir a inclusão de eventos "publicados", juntamente com os restantes do sistema (por exemplo, os eventos gráficos para as aplicações gráficas).

Este último aspecto é talvez o mais significativo deste pacote e será mais aprofundado ao longo da explicação do diagrama de classes, que apresentamos na figura 4.1.

4.2.1.1 Interfaces

Há três interfaces importantes, definidas neste pacote e que passamos a explicar.

Application:

Não define muito, apenas obriga a que cada instância da aplicação tenha um identificador (nome) e agregue uma fila de processamento.

Queue (Fila):

Esta interface tem duas funções importantes, a primeira é a possibilidade de se adicionar um evento à fila (*post*) e a outra é a possibilidade de se adicionarem clientes à mesma, clientes esses, a quem os eventos serão mais tarde entregues, como veremos na implementação desta interface (classe `ApplicationQueue`). Mantém, por isso, uma associação um para muitos com clientes (objectos `QueueClient`).

QueueClient (cliente da fila):

Os clientes da fila serão obrigados a implementar um método de processamento (`void process (String event)`) onde receberão os eventos, que são, nesta fase, apenas pedaços de texto como se pode ver pelo argumento do método.

As classes existentes estão, naturalmente, relacionadas com estas interfaces.

ApplicationQueue:

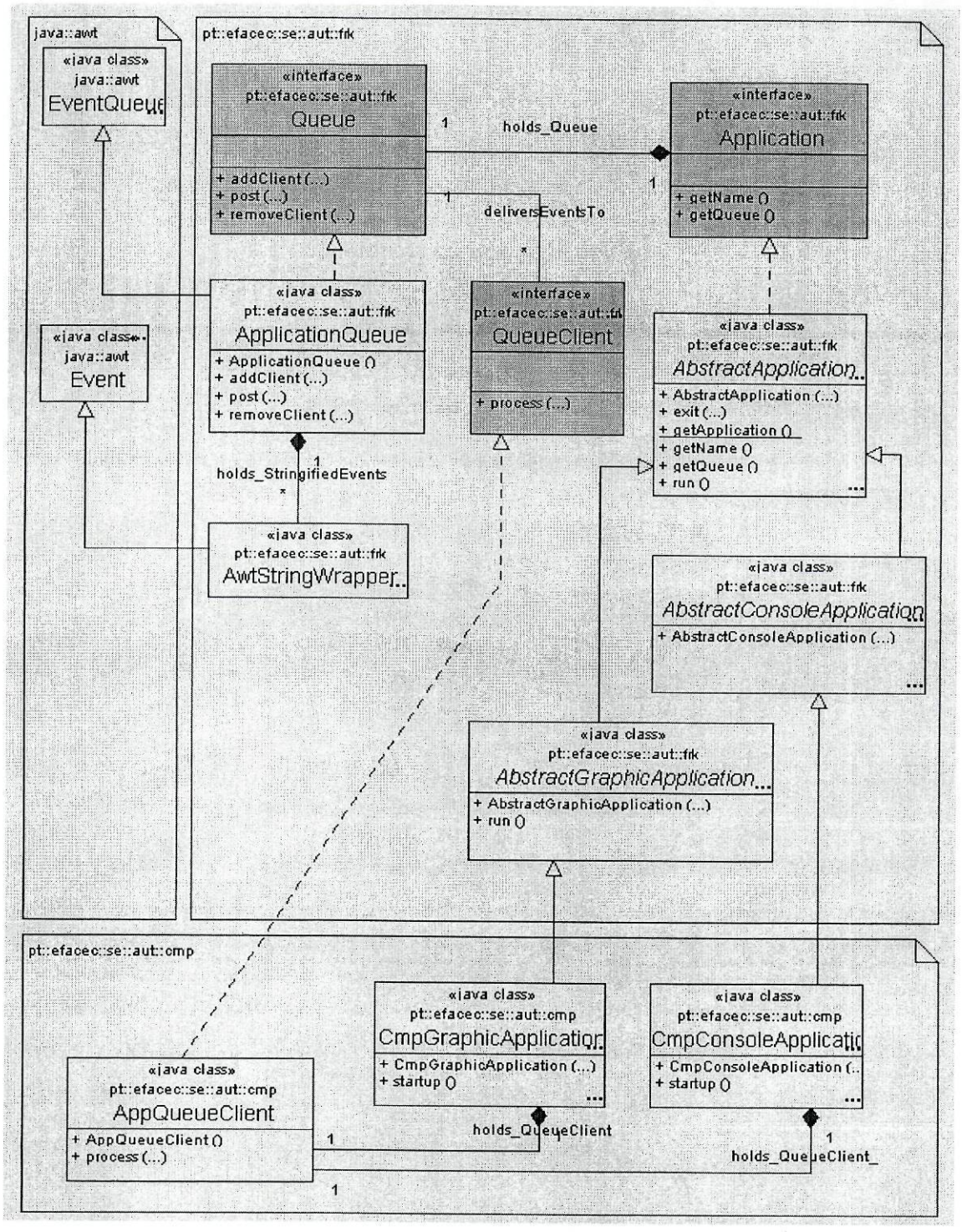

Figura 4.1: Pacote `frk` — Diagrama de Classes

O objectivo desta classe é implementar a gestão da fila de eventos, permitindo por um lado que diferentes "actores" aí coloquem eventos concorrentemente e, por outro, a entrega dos mesmos aos clientes "registados" (os `QueueClients`).

Esta funcionalidade, só por si, traduz fielmente a aplicação de uma receita típica da computação concorrente: o "*thread* operário" (que pode ser encontrado em [SM02, pp. 231-237]). Sendo, que aqui, é utilizada juntamente com outra receita já referida, a "*Callback*", uma vez que a função do *thread* operário é a de invocar um método (*callback*) nos clientes registados sempre que existam eventos novos na fila.

O facto da classe ser derivada da *EventQueue* do pacote AWT, tem duas vantagens importantes: a primeira é que o thread que controla a chegada de novos eventos já está implementado e a segunda é que permite tratar os eventos gráficos como usualmente (por delegação no pai), ocupando-se apenas dos eventos que dizem respeito ao canal, os objectos "*AwtStringWrapper*" que são entregues aos clientes.

AwtStringWrapper:

Esta classe serve apenas para distinguir os eventos do canal de comunicação (no seu formato serializado), dos outros eventos de processamento gráfico. Permite o tratamento diferenciado por parte da `ApplicationQueue`.

AbstractApplication:

Esta classe implementa os mecanismos comuns de inicialização e paragem da própria aplicação, bem como a criação da própria fila de eventos.

AbstractConsoleApplication:

Não traz nada de novo relativamente à anterior, existe apenas por simetria lógica relativamente à aplicação gráfica.

AbstractGraphicApplication:

Esta classe substitui a fila de eventos gráficos comum, pela fila criada pela sua classe pai (`AbstractApplication`).

Como se pode ver no diagrama apresentado na figura 4.1, o pacote `cmp` implementará as suas próprias aplicações derivadas destes dois tipos abstractos (aplicação gráfica e consola), implementando também o cliente da fila de eventos, para que os eventos serializados, uma vez chegados à fila principal da aplicação, lhe sejam entregues.

4.2.2 Pacote `cmp`

Este pacote implementa, como já referimos, toda a lógica do modelo de publicação e subscrição. A sua análise será dividida em várias secções.

A figura 4.2 representa num-o num diagrama de classes.

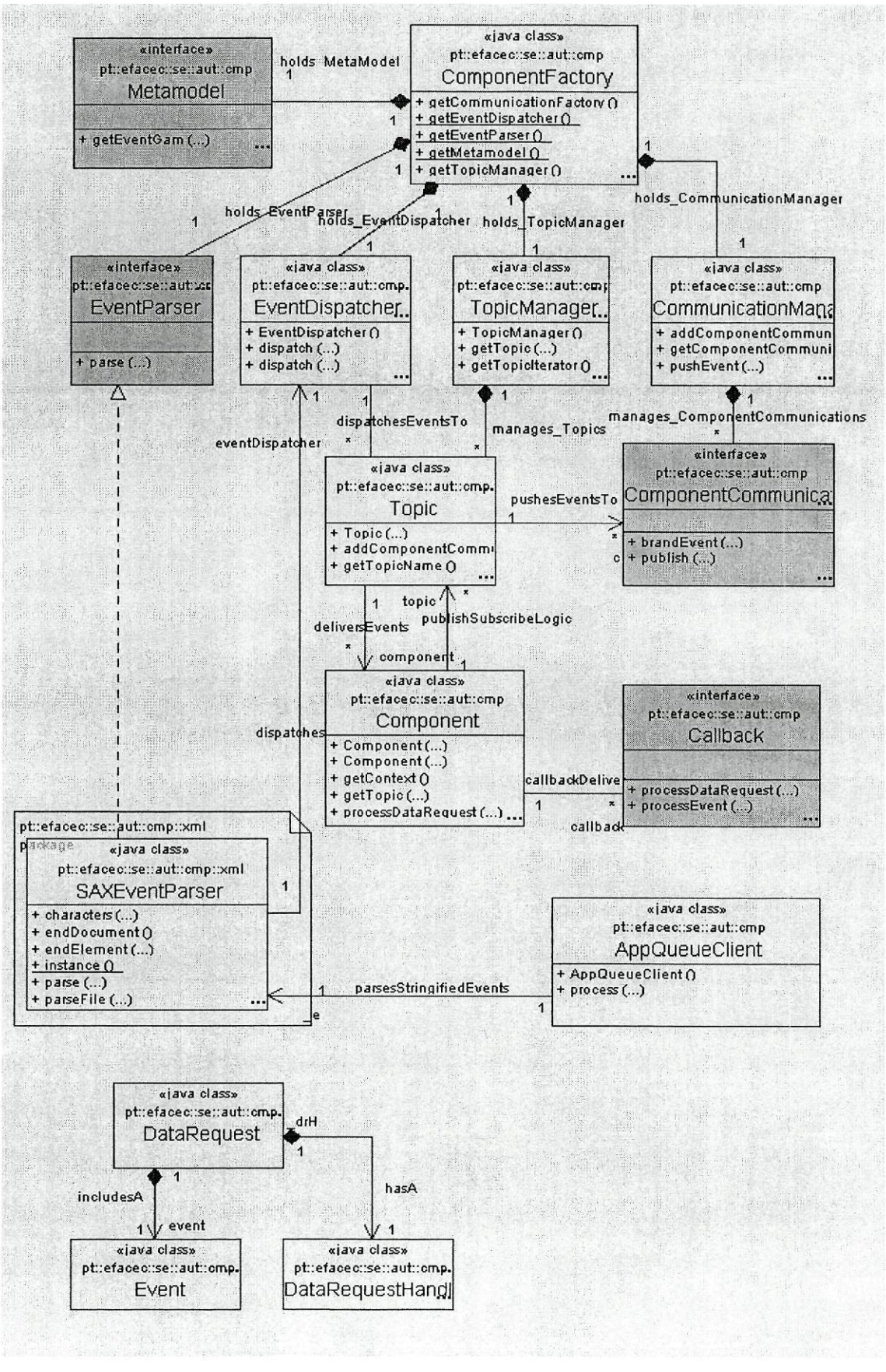

Figura 4.2: Pacote `cmp` — Diagrama de Classes

4.2.2.1 Metamodelo

Define as classes `Event`, `DataRequest` e `DataRequestHandle` que implementam precisamente os conceitos descritos na secção 4.1.2 para eventos e pedidos (*data requests*), sendo que o `DataRequestHandle`, faz parte de um `DataRequest` e será usado pelos componentes para dirigirem a publicação de um evento para o emissor de um pedido (recebem um `DataRequest`, extraem-lhe o `DataRequestHandle` e publicam um evento juntamente com esse *handle*).

4.2.2.2 Componentes, Tópicos e Callbacks

Component:

Esta classe implementa o conceito de componente tal como descrito anteriormente, sendo a interface mais visível para quem desenvolve aplicações usando este *framework*.

Permite a configuração dinâmica em termos de publicação e subscrição de tópicos, permite o registo de objectos `Callback` e finalmente permite a publicação de eventos e pedidos.

Está associada a um conjunto de *callbacks* (instâncias da classe `Callback`), que estejam registadas e a tópicos (classe `Topic`) que publica ou subscreve. Às primeiras, entrega eventos quando chegam e, aos segundos, entrega eventos para enviar ("PUSH").

Topic:

Esta classe é muito importante na interacção com o componente, os tópicos são simultaneamente a porta de saída e de entrada dos eventos, do ponto de vista dos componentes.

Como podemos ver no diagrama de classes da figura 4.2, esta classe está associada ao `EventDispatcher` que "traz" os eventos e à interface `ComponentCommunication` que, como veremos, tratará de "empurrar" os eventos para outros componentes[9].

Resta acrescentar que só existe uma instância de um determinado tópico em cada aplicação independentemente do número de componentes que sejam criados, isto porque os tópicos são geridos pelo `TopicManager`, seguindo o padrão típico da "fábrica" (*factory*) com reutilização dos objectos do mesmo tipo [GHJV95, pp 87–98].

A interacção entre estas classes durante a publicação de um evento por parte de um componente pode ser analisada no diagrama de sequência da figura 4.3.

[9]No próximo capítulo veremos como o tópico é a entidade lógica que vai decidir, por configuração, que implementações do canal serão usadas.

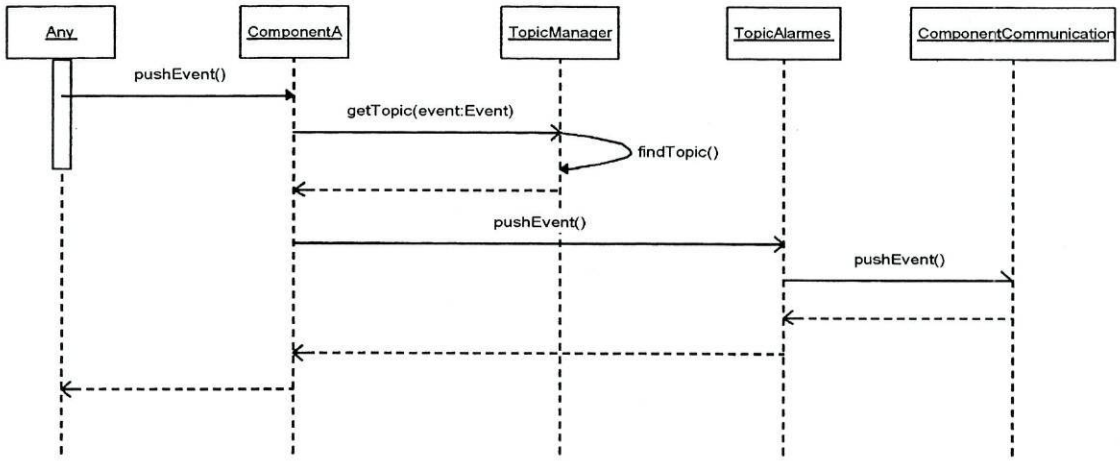

Figura 4.3: Publicação de um evento — Diagrama de Sequência

4.2.2.3 Despacho de Eventos

Como vimos a respeito do pacote `frk`, os eventos que cheguem à fila principal da aplicação, são disponibilizados na sua forma serializada para clientes aí registados. Esse, é o mecanismo de entrada no pacote `cmp` de um evento que foi recebido por uma aplicação[10].

Daí que, cada aplicação com consola ou gráfica (implementadas respectivamente pelas classes `CmpConsoleApplication` ou `CmpGraphicApplication`), registe exactamente um cliente da fila principal de eventos (classe `AppQueueClient`). Para melhor descrever o papel de cada classe envolvida no processo de "despachar" os eventos recebidos, passamos a analisar diagrama de sequência da figura 4.4.

Descrevendo passo a passo:

1. O evento é entregue, como texto, ao cliente da fila principal de processamento;
2. Daí, ele é passado ao *parser* de eventos (neste caso usa-se uma implementação SAX, [HM01, cap. 17 - SAX]) uma vez que os eventos são XML e podem ser entregues agrupados ou isoladamente;
3. Sempre que o *parser* concluir a construção de um objecto evento a partir do texto em XML que lhe foi entregue, este é entregue à instância única do `EventDispatcher` existente na aplicação;
4. O `EventDispatcher` obtém o tópico a que o evento pertence (questionando

[10]Reserva-se a explicação sobre como é que os eventos são colocados na fila de processamento da aplicação para o próximo capítulo (5) a propósito do pacote `comm`.

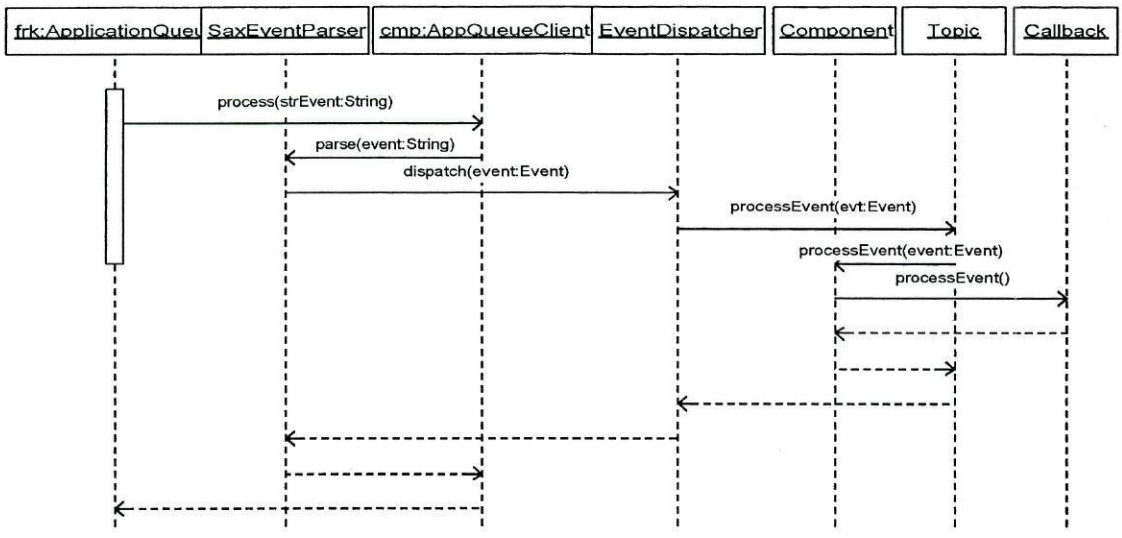

Figura 4.4: Entrega de um evento a uma *Callback* — Diagrama de Sequência

o próprio evento e depois pedindo ao *TopicManager* o objecto respectivo) e entrega-lhe o evento;

5. Finalmente o tópico percorre os componentes que registaram *callbacks* para esse evento e entrega-lhes o evento;

6. O componente faz a entrega final do evento às *callbacks* registadas para o efeito;

4.2.2.4 Classes de gestão

Há, como vimos, algumas classes "solteiras", ou seja classes das quais só existe uma instância em cada aplicação. Este mecanismo consiste na aplicação da receita *singleton*([GHJV95, pp 127-134]), uma das mais usadas na orientação por objectos, sendo que o objecto único de cada uma destas classes (`Metamodel`, `TopicManager`, `CommunicationManager`, `EventParser` e `EventDispatcher`) é obtido a partir da classe `ComponentFactory`, por aplicação dessa receita.

Resta ainda comentar a utilização mais uma vez da "fábrica de objectos", usada pelo `TopicManager` para a construção e gestão dos diferentes tipos de tópicos e pelo `CommunicationManager`, para os objectos `ComponentCommunication`. Em ambos os casos, existe uma classe gestora, encarregue de garantir a construção de objectos específicos de uma determinada classe base ou interface. Neste caso, garantem também a reutilização de objectos do mesmo tipo por oposição à criação de múltiplas instâncias.

4.2.3 Interface com o Pacote `comm`

O "contrato" para interacção com o pacote `comm` para implementação efectiva da transacção de eventos, é definido pela interface `ComponentCommunication`. Todos os métodos importantes da lógica de negócio de publicação e subscrição são aí incluídos, para permitir que as diferentes implementações utilizem toda a informação necessária.

A ideia é permitir que o pacote `comm` tenha a informação suficiente para estabelecer conexões entre aplicações, se for necessário, ou efectue filtragens na publicação e entrega de eventos (para poupar largura de banda por exemplo).

Esta interface pode analisar-se na figura 4.5.

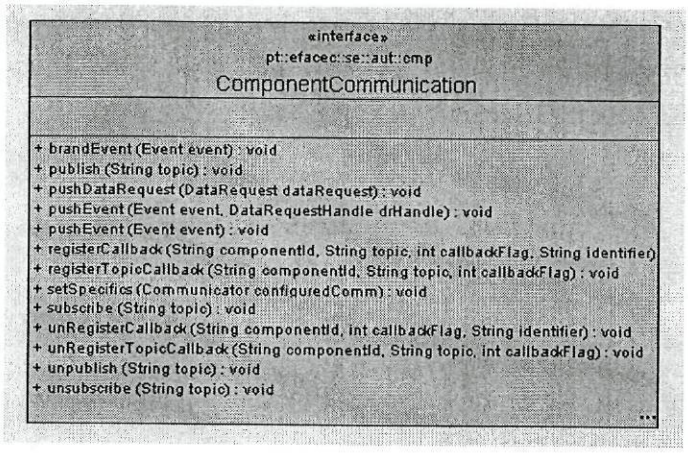

Figura 4.5: Interface `ComponentCommunication`

4.2.3.1 Configuração da lógica de publicação e subscrição

Como foi referido anteriormente, a publicação e subscrição do tópicos por parte de cada componente, pode ser feita por configuração através de um ficheiro XML que obedeça a um esquema muito simples, veja-se um exemplo na figura 4.6.

Source of: file:///home/mpg/jdevhome/mywork/JFRK/SWB/public_html/component_config

File Edit View Help

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
        <component name="Mickey">
                <topic name="Scada" publish="true" subscribe="true"/>
                <topic name="Areas" publish="false" subscribe="true"/>
        </component>
        <component name="Jerry">
                <topic name="Scada" publish="true" subscribe="true"/>
                <topic name="Areas" publish="false" subscribe="true"/>
        </component>
        <component name="Tom">
                <topic name="Scada" publish="true" subscribe="false"/>
                <topic name="Areas" publish="true" subscribe="true"/>
                <topic name="Construction" publish="true" subscribe="true"/>
        </component>
        <component name="Minnie">
                <topic name="Scada" publish="false" subscribe="true"/>
                <topic name="Areas" publish="true" subscribe="false"/>
        </component>
        <component name="Player">
                <topic name="Scada" publish="true" subscribe="true"/>
        </component>
        <component name="WebBridge">
                <topic name="Scada" publish="true" subscribe="true"/>
                <topic name="Areas" publish="true" subscribe="true"/>
                <topic name="Construction" publish="true" subscribe="true"/>
        </component>
</configuration>
```

Figura 4.6: Configuração da lógica de publicação e subscrição em XML

4.2.3.2 Exemplo de um Metamodelo

Como exemplo de um metamodelo definido em XML, pode ver-se a figura 4.7.

Source of: file:///home/mpg/jdevhome/mywork/JFRK/SWB/public_html/metamodel.xml - N

File Edit View Help

```
<?xml version="1.0" encoding="UTF-8"?>
<metamodel>
        <topic name="Scada">
                <event name="Measures">
                        <attribute name="Tag" type="EFAStringValue"/>
                        <attribute name="Value" type="EFALongValue"/>
                        <attribute name="Open" type="EFABoolValue"/>
                        <attribute name="Alarm" type="EFABoolValue"/>
                        <attribute name="Rpm" type="EFALongValue"/>
                </event>
                <event name="Digitals">
                        <attribute name="Tag" type="EFAStringValue"/>
                        <attribute name="Value" type="EFALongValue"/>
                        <attribute name="Open" type="EFABoolValue"/>
                        <attribute name="Alarm" type="EFABoolValue"/>
                </event>
                <event name="Alarm">
                        <attribute name="Tag" type="EFAStringValue"/>
                        <attribute name="Alarm" type="EFABoolValue"/>
                </event>
        </topic>
        <topic name="Construction">
                <event name="Position"/>
                <event name="Equipment"/>
        </topic>
        <topic name="Areas">
                <event name="Login"/>
                <event name="Permission"/>
        </topic>
        <topic name="CommManagement">
                <event name="Ping"/>
                <event name="Repetition"/>
                <event name="WebCmpAction"/>
        </topic>
</metamodel>
```

Figura 4.7: Exemplo de um metamodelo em XML

4.3 Resumo

Neste capítulo fizemos a clarificação do modelo que pretendemos implementar, quer no contexto de aplicações locais, quer no contexto de aplicações na *Web*. Descrevemos o desenho e implementação, feitos ao nível do framework, nos pacotes independentes do canal de comunicação a usar. No próximo capítulo, apresentaremos o desenho e implementação dos canais específicos, RMI e *Web*, sendo que só nos preocuparemos, daqui para a frente, com as especificidades dessas duas abordagens.

Capítulo 5

Implementações — RMI e *Web*

Neste capítulo apresentamos o desenho e implementação do pacote `comm`. Abordaremos as soluções usadas para a implementação de referência em RMI e que usaremos para validação dos pacotes apresentados no capítulo anterior. Finalmente, descrevemos a solução encontrada para a implementação da versão baseada em serviços *Web*, com integração total entre as duas.

5.1 Pacote `comm`

O objectivo deste pacote, decorre claramente do definido para os outros dois descritos até agora (`cmp` e `frk`, capítulo 4). Ou seja, fornecer implementações para a interface `ComponentCommunication`, que sejam coerentes na gestão das ligações (fixas ou virtuais) por forma a que as transacções de eventos serializados cumpram os requisitos da arquitectura de componentes.

Entrando um pouco mais no problema, pede-se às implementações desta camada que dêem resposta a questões como a seguinte:

- se existe numa aplicação, pelo menos um componente publicador, por exemplo, do tópico "Medidas" e se existe, pelo menos, um outro noutra aplicação, subscritor desse mesmo tópico, então todos os eventos pertencentes a este tópico e publicados pela primeira aplicação, devem ser "transportados" até à outra e colocados na sua fila principal de eventos (dando origem ao cenário descrito para "Despacho de um Evento" secção 4.2.2.3)[1].

Numa abordagem simplista e assumindo que todas as aplicações se encontram ligadas entre si (ou seja que é possível "empurrar" eventos de e para qualquer aplicação), a

[1] O cenário descrito deve ser invertido para a publicação de um *data request*, ou seja, as aplicações com componentes subscritores de um tópico entregam à camada `comm` `DataRequests` nesse tópico, para serem transportados até à aplicação publicadora do mesmo.

implementação desta camada, poderia resumir-se ao transporte de cada evento para todas as outras aplicações, uma vez que, como vimos, o pacote `cmp` filtraria os eventos por não encontrar tópicos e finalmente *callbacks* a quem os entregar.

Facilmente se conclui, no entanto, que para um sistema complexo com muitos componentes, essa abordagem rapidamente inundaria a rede de eventos reduzindo largamente a largura de banda útil. Por outro lado, teria consequências drásticas no desempenho das aplicações, por passarem grande parte do seu tempo de processamento a fazerem o despacho de eventos inúteis.

É fácil compreender que, como veremos à frente, o objectivo fundamental desta camada é precisamente a gestão eficiente dessas ligações entre aplicações, por forma a que por um lado a publicação de um evento por parte de uma aplicação seja uma operação "leve", sem passar por muitos critérios de filtragem, mas por outro que a ocupação da largura de banda disponível seja, também ela, eficaz e tanto quanto possível contendo apenas transacções de eventos úteis, ou seja, de eventos que têm um destinatário concreto.

Resolvidas essas questões, será depois necessário gerir os diferentes tipos de comunicadores que poderão existir associados ao mesmo tópico (voltaremos a este assunto mais tarde na descrição das implementações).

Observando pormenorizadamente a interface que tem que ter "implementações" neste pacote (figura 4.5) e relembrando a partir do diagrama de classes (figura 4.2) que as invocações de métodos desta interface serão feitas por objectos do tipo `Topic`, descrevem-se a seguir, um a um, os métodos aí presentes:

`publish`

> Decorre da invocação de um método homónimo por parte de um componente da aplicação e significa que a partir dessa invocação esse componente é um publicador nesse tópico. Do ponto de vista do pacote `comm`, despoletará as providências necessárias para tornar este comunicador num publicador desse tópico (o que pode passar pelo estabelecimento de conexões fixas ou de circuitos virtuais).

`unpublish`

> Invalida as conexões, baseadas na invocação anterior do *publish*, para o mesmo tópico. Deve considerar-se o facto destes comunicadores existirem ao nível da aplicação e de existirem vários componentes por aplicação que reutilizarão os canais de comunicação do mesmo tipo. Isto implica que não se possa invalidar uma ligação directamente após uma chamada a *unpublish* (ou *unsubscribe* como veremos), é preciso fazer uma gestão do número de chamadas a *publish* e anulá-las, uma a uma, até se desfazer uma ligação.

`subscribe`

Idêntico ao *publish* mas com sentido inverso, ou seja, serve para despoletar o estabelecimento de uma ligação com outras aplicações, com componentes publicadores deste tópico. Repare-se que, nesta situação, o pacote `comm` tem que permitir a transacção de *data requests* desta aplicação para outras, publicadoras deste tópico.

`unsubscribe`

São válidas as considerações feitas para o *unpublish*, embora devidamente adaptadas à lógica de subscrição.

`registerCallback`

Permite, se for interessante, tomar medidas específicas quanto à transacção ou não de eventos entre aplicações, com base não só na publicação e subscrição de tópicos, mas também no registo de *callbacks* específicas para eventos, que traduzem se o evento vai ou não efectivamente ser processado no destinatário.

Dependendo da uma *flag* "callbackFlag", a *callback* servirá para a entrega de eventos ou *data requests*.

`unregisterCallback`

Anula o registo anterior de uma *callback* pela invocação de *registerCallback*.

`registerTopicCallback` Este método é equivalente ao *registerCallback*, mas permite tratar desta vez a informação de que há um componente interessado em "ouvir" todos os eventos de um determinado tópico (ou *data requests* dependendo da "callbackFlag").

`unregisterTopicCallback`

Anula o anterior.

`brandEvent`

É chamado antes da publicação efectiva de qualquer evento para que, cada canal de comunicação específico "marque" o evento por forma a que, no destinatário, se possa saber por que canal de comunicação ele circulou.

Esta questão é fulcral na gestão de diferentes canais de comunicação, configurados para o mesmo tópico, numa determinada aplicação. Por exemplo, para uma aplicação, que no tópico "Alarmes" seja simultaneamente um componente RMI e *Web*, o mecanismo de "marcação" dos eventos pelo canal em que foram transmitidos, permite implementar a funcionalidade de reenvio pelo outro canal.

As secções que se seguem, discriminam logicamente os problemas que terão que ser tratados por cada implementação. Faz-se uma apresentação sucinta de cada problema, mostrando-se depois, nas secções relativas às duas implementações, as soluções concretas encontradas para cada um.

5.1.1 Transporte de Eventos

A primeira questão que as implementações `comm` terão que resolver é a do transporte de cada evento serializado (num pedaço de texto) entre aplicações.

No fundo, corresponde à implementação do modelo PUSH (descrito em 4.1.1.1) por cada canal, com as suas especificidades tecnológicas.

5.1.2 Identificação de Aplicações Remotas

A forma de "descobrir" ou referenciar aplicações remotas, identificando-as, é outro dos problemas que tem de ser resolvido por cada canal. Isto porque, é necessário para a implementação da lógica de publicação e subscrição, na manutenção das ligações entre aplicações, que elas comecem por se identificarem umas às outras.

5.1.3 Conexões

Decorrente do problema anterior, está o estabelecimento, ou não, de ligações fixas entre aplicações, conforme as tecnologias o permitam. É evidente que esta tarefa é facilitada por tecnologias que permitam, à partida, implementar directamente estas conexões fixas, mas em todas as implementações, tem que existir a noção de circuito "virtual" entre duas aplicações, como sendo aquele que permite que um evento parta de uma aplicação e chegue à outra (com recibo de recepção).

5.1.4 Sessões

Finalmente, associada a cada uma dessas ligações, surge imediatamente a noção de estado e consequentemente de sessão. Ou seja, é necessário, não só que se tenha um circuito estabelecido para transporte de eventos entre aplicações, mas também que se saiba o suficiente sobre essa outra aplicação para tomar a decisão de lhe enviar, ou não, um determinado evento.

No âmbito específico do problema da comunicação, o estado associado a cada sessão, pode estar nas próprias conexões, ou seja, se uma aplicação não está "ligada" a outra é porque não existe uma intersecção de interesses na configuração por tópicos de ambas (ou seja, nenhuma é publicadora de um tópico que outra subscreva).

5.2 Implementação RMI

Nesta secção descreve-se a implementação efectuada para a tecnologia RMI, representativa de um modelo de computação distribuída tradicional (*middleware*). Respondem-se ás questões levantadas na secção anterior, a propósito dos problemas que têm que ser resolvidos pelas implementações específicas da camada de transporte.

Esta implementação será útil em duas fases do trabalho, numa primeira fase, para comparação com a implementação baseada em serviços *Web*, do ponto de vista das soluções de desenho e, mais tarde, para a realização de testes comparativos do desempenho de ambas as soluções.

5.2.1 Introdução Tecnológica

Antes de avançarmos para os detalhes da implementação, convém clarificar alguns aspectos da tecnologia RMI, para que melhor se compreenda a abordagem tomada[2].

Antes de mais convém esclarecer que a tecnologia RMI é definida na especificação da própria linguagem JAVA (ver [Eck00]) como parte do pacote básico, estando portanto disponível em qualquer ambiente de desenvolvimento JAVA (JDK— *Java Development Kit*) ou em ambiente de instalação (JRE — *Java Runtime Environment*). Isto por oposição, por exemplo, às especificações de módulos relacionados com o desenvolvimento empresarial, como os EJB — *Enterprise Java Beans* ou o *framework* de comunicações JMS — *Java Messaging Service*, ambos pertencentes à especificação do pacote empresarial J2EE na sua versão 1.3.

O facto da tecnologia pertencer ao pacote básico, é importante por dois motivos essenciais:

- Em primeiro lugar, tem existido alguma instabilidade à volta da evolução da própria especificação da linguagem, com métodos a prescreverem e a serem substituídos por outros. Ora, é sabido que a utilização de APIs estáveis dá mais garantias para a manutenção do código.

- Em segundo lugar, nem todos os fabricantes de *software* que fornecem "ambientes" JAVA, fornecem a totalidade da especificação.

  Por exemplo, alguns fabricantes de servidores *Web*, só implementam a especificação relacionada com as *servlets*, ou seja, só implementam o repositório *Web* e não o de aplicações (para os EJBs); veja-se, como exemplo, o popular *Tomcat*. Por outro lado, mesmo os que implementam servidores de aplicações (arquitecturas J2EE) nem sempre implementam a totalidade das tecnologias

[2]O leitor que esteja suficientemente familiarizado com a tecnologia, pode, sem prejuízo para a sequência do texto, passar para a secção 5.2.2 onde se inicia a descrição da solução baseada em RMI.

especificadas, por exemplo, o *JBoss* não fornece a tecnologia JMS, apesar de fornecer um repositório para EJBs.

A especificação da linguagem adianta dois conceitos fundamentais no seu "modelo de objectos distribuídos": o conceito de objecto remoto e o de interface remota (ver [SCD+01, cap. 3 — Distributed Computing Using RMI]).

Um *objecto remoto* é um objecto cujos métodos podem ser invocados a partir de outro objecto, a correr numa máquina virtual diferente (eventualmente noutra máquina). A definição de objecto remoto é feita a partir de *interfaces remotas*, que não são mais do que interfaces JAVA comuns, que declaram os métodos do objecto, invocáveis remotamente.

Assim, a tecnologia RMI consiste na invocação de métodos em objectos remotos, pertencentes a interfaces remotas. Em termos práticos, é dada ao programador a possibilidade de obter uma referência para um objecto remoto (usando um *stub* desse mesmo objecto) e invocar métodos nele como se de um objecto local se tratasse (na mesma máquina virtual).

Esta tecnologia é muito semelhante à definida no CORBA — *Common Object Request Broker Architecture* ([MM97]). Do facto de o RMI estar "agarrado" a uma linguagem específica (o JAVA) e o CORBA não, decorre uma vantagem e uma desvantagem imediatas:

- O RMI é muito simples de usar, uma vez que se usa a própria linguagem de programação para a definição das interfaces remotas (não é necessário recorrer a uma notação própria como o IDL — *Interface Definition Language*).

- a interoperabilidade (entre aplicações desenvolvidas em linguagens de programação diferentes) não era conseguida, em RMI. Este facto foi minimizado pela adopção do protocolo especificado pelo OMG — *Object Management Group*, o IIOP[3] como alternativa ao JRMP — *Java Remote Method Protocol* original, o que veio permitir a integração com aplicações distribuídas baseadas em interfaces CORBA.

Para concluir esta introdução ao RMI, refira-se apenas que, tal como para o CORBA, o programador não tem que se preocupar com os pormenores da implementação das invocações remotas, quer por parte do cliente quer por parte do servidor das mesmas, uma vez que esse código é gerado a partir da interface remota com uma ferramenta de pré-compilação, que faz parte do ambiente de desenvolvimento (neste caso o *rmic*).

[3] *Internet Inter-ORB protocol*; o aparecimento deste protocolo ([SV02]) veio, aliás, colmatar algumas falhas na especificação do CORBA para a interoperabilidade total entre fabricantes.

5.2.2 Implementação

Feita uma introdução à tecnologia a usar nesta implementação, para comunicação entre componentes e consequentemente, entre aplicações, passamos agora à descrição da implementação baseada nessa tecnologia.

5.2.2.1 Transporte de eventos

Da descrição da tecnologia RMI, a primeira ideia que fica é que o "problema" de fazer passar um pedaço de texto entre duas aplicações, com o devido controlo de erros, é trivial. Isto porque, a tecnologia trata precisamente de invocação remota de métodos.

Assim, o problema do transporte de eventos em RMI resolve-se pela definição de uma interface remota única, que será usada para toda a comunicação entre aplicações, a interface "PUSH".

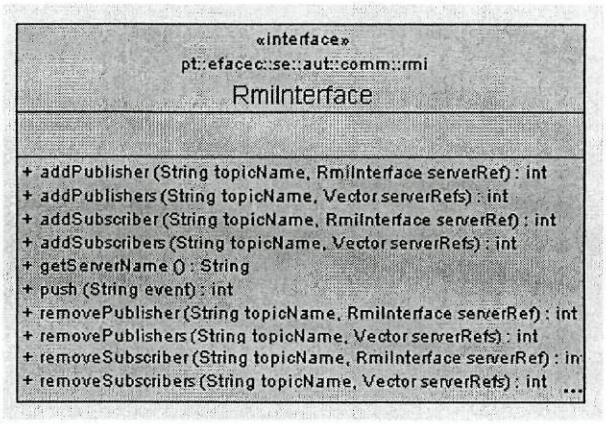

Figura 5.1: Interface RMI básica — PUSH

Todos os eventos publicados por componentes, serão transportados na sua forma serializada, em texto, através da invocação remota deste método entre aplicações, o que está em sintonia total com o modelo PUSH defendido na secção 4.1.1.1.

A figura 5.1 apresenta a interface `RmiInterface`, que define precisamente os métodos invocáveis remotamente por outras aplicações. Para além do método `push`, esta interface inclui ainda outros métodos que serão explicados mais tarde, a propósito da descrição da aplicação *Navigator* (secção 5.2.2.3).

5.2.2.2 Identificação de Aplicações Remotas

Na tecnologia RMI, para que os objectos JAVA possam invocar métodos remotamente uns nos outros, têm que "encontrar" primeiro uma referência local que represente o objecto remoto (através de um *stub*).

A forma clássica de identificar aplicações num contexto RMI e obter referências para os seus objectos remotos, consiste na sequência seguinte:

1. Uma aplicação que implementa uma interface remota, ou seja um servidor RMI, arranca e regista-se (*bind*) no contexto RMI[4]. Esse registo é feito com um nome que tem que ser único em cada *Registry*.

2. Uma segunda aplicação interessada em invocar métodos da interface disponibilizada pela primeira, arranca a seguir e pede (*lookup*) ao *Registry* uma referência para um objecto que represente o servidor registado.

   Este pedido é feito com base no nome usado pelo servidor para registo, ou seja, o cliente tem que conhecer esse nome.

3. Usando essa referência, o cliente invoca métodos no servidor, como se o estivesse a fazer localmente.

O grande problema com esta sequência está, evidentemente, no facto da aplicação cliente ter que conhecer à partida o identificador de cada servidor em que estará interessada em invocar métodos remotamente.

Transpondo esta questão para o nosso problema específico, todas as aplicações serão simultaneamente servidoras (no sentido em que vão implementar a interface remota apresentada em cima) e clientes. Uma mesma aplicação, enquanto publicadora de um evento, é cliente do destinatário (invoca o método *push* neste) e enquanto receptora de um evento é servidora.

Pretendia-se que estas ligações fossem efectuadas com base na publicação/subscrição de tópicos, por parte dos componentes de cada aplicação e não no "nome" das aplicações.

A solução encontrada baseia-se no facto da tecnologia RMI permitir usar como argumentos de um método remoto, referências para objectos remotos. Exemplificando, se uma aplicação A, conhecer o nome com que uma aplicação B se registou, pode pedir ao *registry* uma referência para essa aplicação. A partir desse momento poderá invocar métodos em A, podendo, por exemplo, passar uma referência para si própria, numa dessas invocações. Dessa forma a aplicação A, poderia ficar com uma referência para B mesmo sem conhecer, à partida, o "nome" desta.

Na nossa solução, existe sempre e antes de qualquer outra aplicação arrancar, uma aplicação especial a correr. Esta aplicação tem um "nome" fixo e bem conhecido por todos "*Navigator*" (ou navegador). Assim, com base no princípio descrito atrás, todas as outras aplicações só terão que pedir ao registry uma referência para o navegador, entregando-lhe depois uma referência para si próprias e a partir desse momento, o navegador poderá gerir as ligações, fornecendo essas mesmas referências, sempre que necessário.

[4]Tal como com a tecnologia CORBA, também aqui é necessário ter um *daemon* a correr para a gestão dos servidores (interfaces remotas) disponíveis. Nesta tecnologia o *daemon* chama-se "rmid" ou simplesmente "*Registry*".

É esse mecanismo que descrevemos na próxima secção.

5.2.2.3 Gestão das ligações

Para este tipo de tecnologia, uma conexão existe quando um objecto tem uma referência válida para outro objecto remoto. Dentro do nosso problema, as diferentes aplicações terão que possuir nos seus módulos de comunicação RMI, referências para objectos semelhantes de outras aplicações, nos quais seja expectável virem a invocar o método `push`.

Isto quer dizer que, para que o canal cumpra a sua função de acordo com a arquitectura de componentes, basta que possua o conhecimento de dois grupos de associações entre tópicos e referências para objectos remotos. Um primeiro grupo que associe objectos remotos a tópicos subscritos, ou seja, para todos os tópicos publicados pela aplicação, quais são as aplicações que têm componentes que os subscrevem. E um segundo grupo, equivalente a este, mas que associe objectos remotos a tópicos publicados, ou seja, para todos os tópicos subscritos, que aplicações remotas os publicam.

Como veremos no diagrama de classes (figura 5.3), estas duas associações correspondem às duas associações entre a classe `RmiServer` (que implementa a interface remota) e a classe `TopicMemberTable` (que mantém as associações entre objectos remotos e tópicos)[5].

A gestão destas duas associações é feita pelo *navegador*, da seguinte forma:

- Sempre que uma aplicação arranca, pede uma referência para o *navegador* ao *registry*;
- Sempre que os métodos *publish* ou *subscribe* forem chamados (em `RmiComponentCommuni`
os métodos *registerPublishedTopic* ou *registerSubscribedTopic* do navegador serão chamados com a referência para a aplicação como argumento (para além do nome do tópico);
- Na implementação desses métodos, o navegador procura o mapa de subscritores (ou publicadores se o método invocado tiver sido o *subscribe*) para esse tópico;
- Para cada subscritor encontrado, o navegador invoca o método *addPublisherOnTopic* no objecto correspondente, passando como argumento a referência para o novo objecto publicador e o nome do tópico;
- Desta forma, cada um dos subscritores de um determinado tópico é actualizado quando um novo publicador para esse tópico informa o navegador desse facto. Essa actualização é feita pela adição da referência para o novo objecto, à sua lista de publicadores;

[5]Essas associações estão identificadas no diagrama como "*publishers*" e "*subscribers*", publicadores e subscritores!

[6]Esta classe implementa a interface `ComponentCommunication` para a versão RMI.

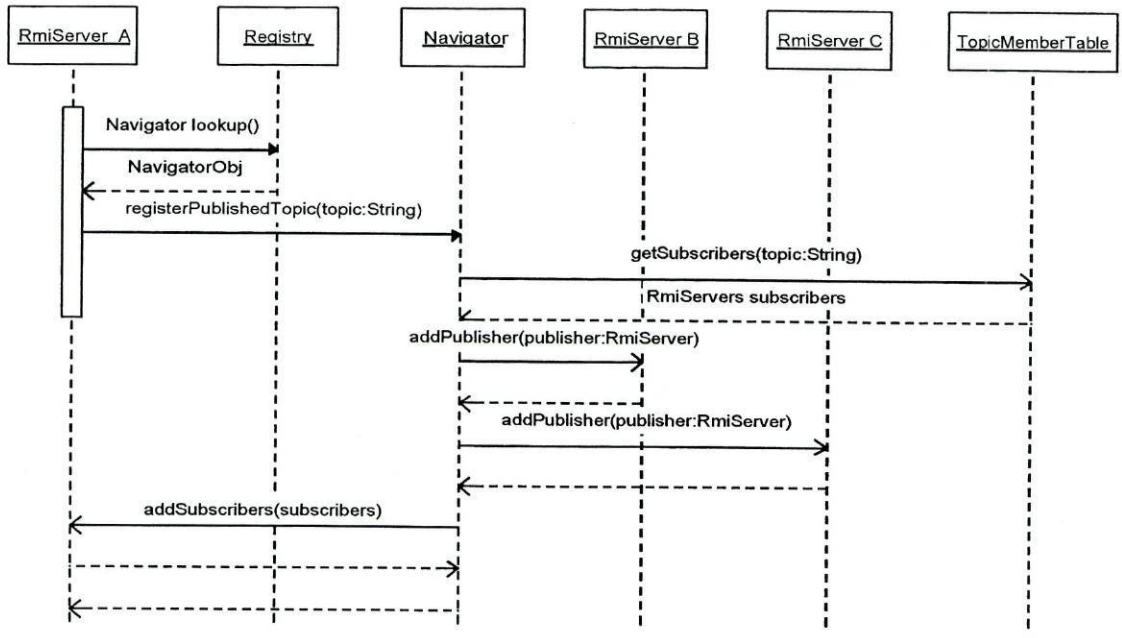

Figura 5.2: Entrada de um novo publicador

Estes passos podem ver-se no diagrama de sequência da figura 5.2.

5.2.3 Desenho do Pacote `comm.rmi`

A figura 5.3, representa o diagrama de classes relativo à implementação RMI do canal.

É de notar, que se integram neste diagrama também as classes relativas à implementação do navegador, apesar de ser uma aplicação independente e separada, procede-se desta forma, por existirem, do ponto de vista RMI, associações com os objectos do canal RMI.

Finalmente, a figura 5.4, explica a ligação que faltava ao *framework* descrito no capítulo anterior, ou seja, a colocação dos eventos na fila principal de processamento da aplicação.

5.3 Implementação Baseada em Serviços *Web*

Da tecnologia RMI para a *Web*, ou, mais especificamente, *usando serviços Web*, há várias diferenças que se poderiam apontar e que já fomos abordando, quer no capítulo 3, a respeito das tecnologias *Web* disponíveis, quer na secção 5.2.1 deste capítulo, a propósito da tecnologia RMI.

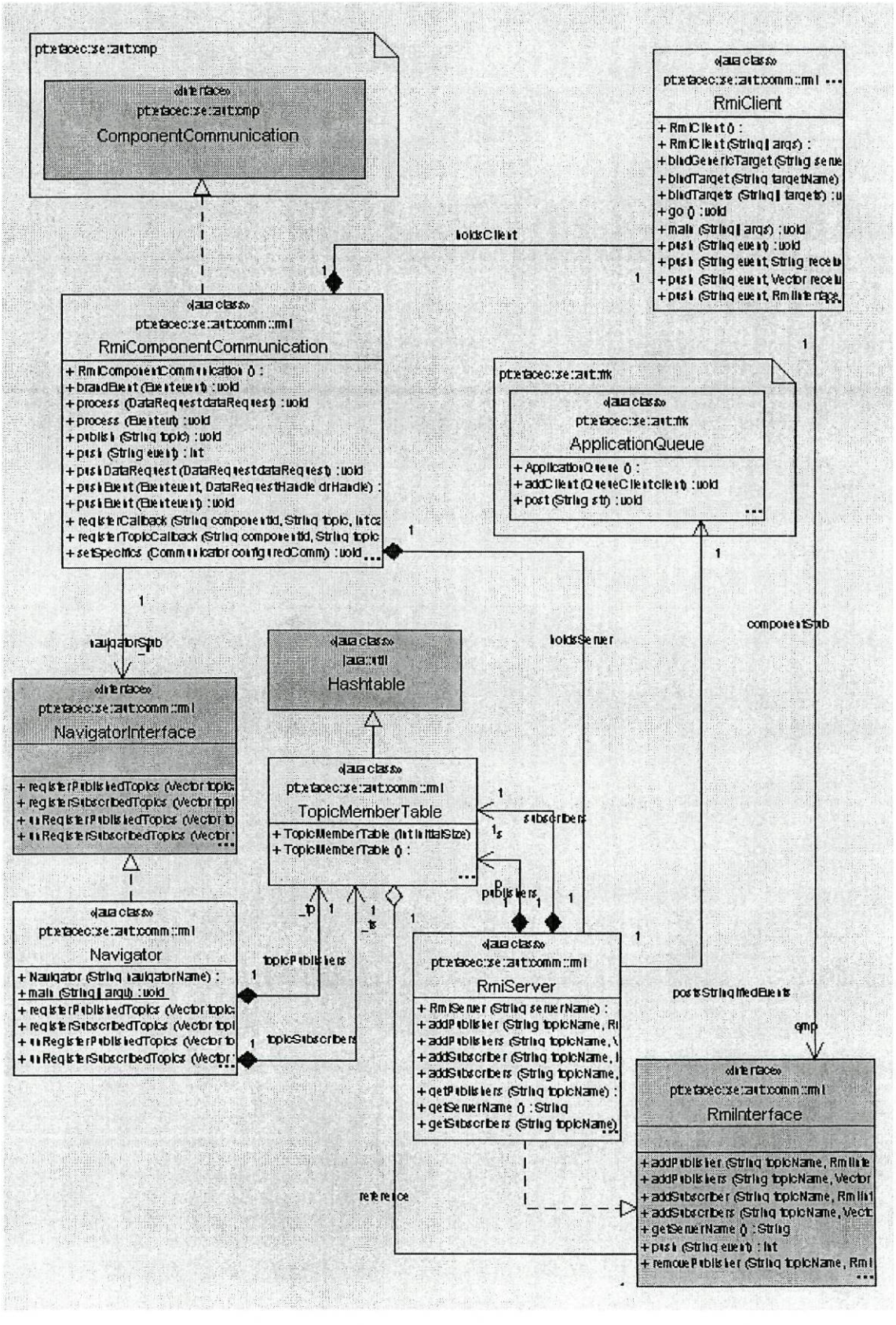

Figura 5.3: Implementação RMI do canal

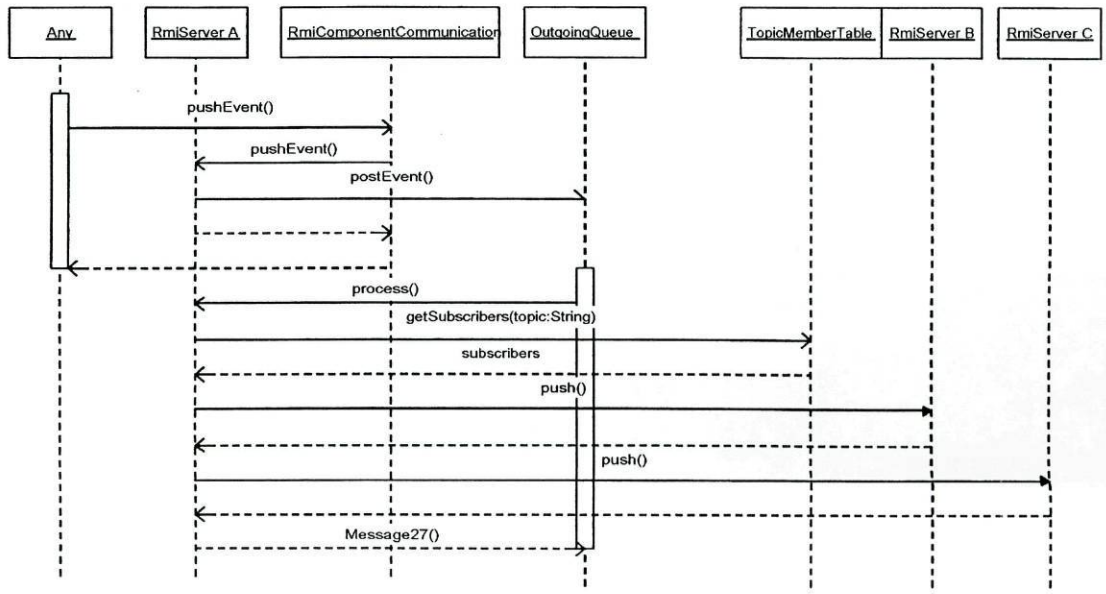

Figura 5.4: Transporte de um evento pelo canal RMI

Algumas dessas diferenças têm um impacto especialmente complicado, na transposição da lógica de componentes descrita até agora, para esta tecnologia. Como sejam, a adaptação de um mundo de aplicações com circuitos "solidamente" estabelecidos, com base na lógica de publicação e subscrição de tópicos, a uma tecnologia tão diferente como é a dos serviços *Web*.

Concretizando essas diferenças:

Não há conexões fixas.

O tempo de vida da instância do serviço *Web* que atende um pedido por parte de um cliente, é o tempo de vida da própria conexão. Ora, imaginando que tenham que circular eventos entre aplicações (passando pela *Web*), é de esperar que a invocação dos serviços *Web* seja frequente, ou seja, que a conexão física entre aplicações seja constantemente interrompida e retomada com uma nova instância, de um serviço, pelo meio.

Não há estado.

Vimos, na secção 3.1.2.1, formas para seguir uma sessão e, consequentemente, contornar esta característica do próprio protocolo HTTP. Usando SOAP (ou no nosso caso XML-RPC), nem sequer esses métodos estão á nossa disposição, uma vez que o HTTP é encapsulado e escondido das aplicações. Isto é um problema de facto porque, como vimos na descrição da implementação em RMI, parece mais ou menos evidente a necessidade de manter estado sobre a publicação e subscrição de tópicos, por parte das outras aplicações.

O ciclo de comunicação tem que começar, necessariamente, pelo cliente.

O modelo que pretendemos implementar é claramente bidireccional, mas usando serviços *Web*, estamos obrigados a que tenha necessariamente que ser o invocador a tomar a iniciativa.

5.3.1 Transporte de Eventos

O problema do transporte de eventos, não é especialmente complicado, desde que nos abstraiamos das outras questões levantadas (conexões, estado, bidireccionalidade).

Temos uma tecnologia que permite invocar um procedimento remotamente, numa classe instalada num servidor *Web*.

Essa tecnologia (XML-RPC) pode usar como protocolo de transporte o HTTP, como tal, ter facilidades na compatibilidade com configurações típicas de segurança para *firewalls* ([CB94]).

É fácil imaginar que, pelo menos no sentido do invocador para o serviço *Web*, "empurrar eventos" seja semelhante do ponto de vista do transporte: é preciso definir um serviço *Web*, que aceite um pedaço de texto como argumento, para que a passagem de um evento serializado se faça.

No sentido contrário há duas soluções possíveis e que passamos a explicar:

- Ter implementações de "servidores" SOAP em todas as aplicações e instalar o serviço definido (para transportar eventos) em todas as aplicações.

  Vantagens: o problema da bidireccionalidade fica resolvido e o da identificação das aplicações também, uma vez que teríamos ligações ponto-a-ponto entre todas as aplicações, de um lado ou do outro da *firewall*.

  Desvantagens: apesar de haver inúmeras implementações do SOAP e de estarem disponíveis para várias linguagens, é de esperar que tenham impacto, quer no desempenho das aplicações, quer na dimensão das mesmas. Ora, este é um problema razoavelmente importante, quando se pretende que as aplicações possam correr quer separadamente, quer num navegador em *applets*.

- A alternativa, é o uso de uma receita típica para problemas deste tipo chamada "Actualizações Sucessivas"[7] e cuja descrição exaustiva pode ser encontrada em [SM02, pp. 248-257].

  De acordo com este padrão, o cliente deverá ciclicamente invocar um serviço no servidor, para recolher potenciais mensagens, aí acumuladas para si. Ou seja, como só o cliente é que pode tomar a iniciativa de invocar um serviço *Web*, a ideia será que ele ciclicamente invoque um serviço, por forma a que também as mensagens do servidor para o cliente sejam escoadas.

[7] Esta receita também é conhecida como *"Client Pull"* ou *"Server Push"*.

Esta receita é especialmente concebida para a transmissão de pequenas mensagens relativas a dados que estão sempre a mudar. Para períodos entre actualizações relativamente pequenos, esta descrição encaixa bem num sistema SCADA, em que existe muita volatilidade nos valores telemetrados, mas cada actualização traz pouca informação (as mensagens acumuladas desde a última actualização).

A opção pela segunda hipótese parece, à partida, claramente preferível.

Surgem no entanto duas questões, importantes, relacionadas com esta solução:

- A primeira e mais importante, está relacionada com o facto de este mecanismo introduzir um atraso forçado na actualização dos clientes.

  Relativamente a esta questão, é evidente que a gestão do período mínimo para as actualizações servidor-cliente é fulcral e tem que ser adaptada, não só ao problema específico desta tese (aplicações SCADA), mas também à intensidade momentânea de tráfego, que pode estar a ocorrer num determinado momento. Este problema parece, aliás, a questão fundamental que poderia negar a tese que se defende aqui e voltaremos, naturalmente, a ele na análise dos resultados obtidos e discussão das conclusões.

- A segunda questão, relaciona-se com a possibilidade evidente de existir mais do que um evento, para ser transmitido do servidor para um determinado cliente, por altura da invocação do serviço. Isso obriga a que, por um lado, seja permitido o transporte de vários eventos serializados no mesmo "pedaço" de texto e por outro, que o mecanismo de despacho de eventos possa tratar uma sequência de vários eventos[8].

5.3.1.1 *Serviço Web*

Definem-se dois procedimentos a incluir no serviço *Web*, que fará parte do canal:

- interface `push`:

  Para que os clientes possam "empurrar" eventos serializados para os servidores. A ser invocada periodicamente, desde que existam eventos na fila de saída de uma aplicação cliente (fora da rede local);

- interface `ping`:

  Para escoar os eventos destinados à aplicação cliente e acumulados no servidor. Deve ser invocado periodicamente, enquanto a "ligação" for válida.

[8] O recurso a um parser SAX para o tratamento do XML (como se pode ver no diagrama de classes da figura 5.3) simplifica esta questão, uma vez que o tratamento é feito "etiqueta" a "etiqueta", por oposição à leitura completa do documento XML, que caracteriza a implementação DOM.

5.3.2 Gestão das Ligações por Eventos

Se observarmos a interface RMI definida para o canal (figura 5.1), verificamos que foram adicionados métodos para a gestão dos publicadores e subscritores, que cada aplicação "conhece". Esses métodos, são única e exclusivamente, para invocação por parte do "*navigator*", enquanto gestor de ligações.

Esta abordagem torna-se impossível na implementação *Web* do canal, pela opção tomada ao nível do transporte de eventos, nomeadamente, pelo facto de pretendermos ter um "invocador" e um "fornecedor" de serviços *Web*, nos extremos do canal.

Explicando um pouco melhor, seria possível adicionar à lista de procedimentos a incluir no nosso serviço *Web*, novas "interfaces", por exemplo os métodos que existem na interface RMI para a gestão das ligações, mas essa solução só poderia ser usada num dos sentidos da comunicação. Ou seja, não seria possível ligar uma aplicação do "lado servidor" a uma aplicação do "lado cliente" (a correr numa *applet* por exemplo).

Assim, a única solução para actualizações dinâmicas do estado do próprio canal nas aplicações clientes, será a utilização das interfaces existentes, aquelas, que permitem passar "pedaços de texto", em qualquer dos sentidos.

Se formos um pouco mais longe e usarmos precisamente eventos comuns para essa mesma configuração dinâmica, então, estamos à partida a garantir menos problemas relativos à implementação de situações especiais, ou excepções na recepção e despacho de eventos.

A este respeito, resta acrescentar dois pontos essenciais para que se compreenda o desenho e consequente implementação efectuada:

- A ideia de "gerir" o próprio canal com base em eventos, não se confina apenas à implementação das aplicações cliente (relativamente ao serviço *Web*). Pode naturalmente ser usada no lado servidor também, com duas vantagens imediatas: a interface do serviço *Web* mantém-se simples (os métodos `push ()` e `ping()` adiantados na secção anterior são suficientes); será possível reutilizar parte do desenho e implementação relativas à gestão das ligações entre as implementações servidora e clientes.

- Como foi possível observar na arquitectura de componentes definida no capítulo 4, para receber um evento, uma aplicação tem que ter uma *callback* e registá-la num componente.

  Isto sugere, fortemente, que o próprio canal (na sua versão *Web*) possua um objecto `Component` para a sua gestão, sugere também, que o comportamento desse componente, seja equivalente nas duas implementações que se adivinham para as versões servidora e cliente do canal na *Web*.

5.3.3 Identificação de Aplicações Remotas

A identificação das aplicações do ponto de vista das comunicações feitas pela *Web*, traduz-se numa chave de sessão. Esta solução é muito comum nas aplicações *Web* em geral e encontra-se descrita detalhadamente em [SM02, pp. 220-230].

Na solução adoptada, o estado relativo às lógicas de publicação e subscrição dos clientes é guardado quer no cliente, quer no servidor. Isto porque, na prática, como veremos o par funcionará como espelho (ou *proxy*) um do outro.

5.3.3.1 Conceito de sessão

Ainda de acordo com a receita referida para a identificação das aplicações remotas, será usada uma chave de sessão para identificar cada cliente perante o servidor e por outro lado, para encaminhar as mensagens do servidor para o cliente certo (supondo por exemplo mais do que um componente na aplicação cliente).

Resta acrescentar, que é necessário que essa chave circule com cada mensagem, para que a comunicação seja bem sucedida (validada pelos dois extremos).

5.3.4 Separação — Servidor e Cliente

Ao longo dos pontos já descritos para a solução *Web*, fomos naturalmente falando do conceito de servidor e de cliente, ao nível da implementação do canal. Esta questão decorre naturalmente das tecnologias envolvidas, mais especificamente do uso de serviços *Web* para a implementação do canal.

Convém, no entanto, esclarecer que esta diferenciação se dá apenas ao nível do canal e relativamente à "localização" geográfica da aplicação; nada distingue, do ponto de vista da lógica funcional uma aplicação que corra como "cliente *Web*" de uma outra que corra como "RMI", de resto, foi definido como objectivo desejável que a mesma aplicação pudesse "transitar" para outro canal, apenas pela configuração ao nível dos seus tópicos.

Recorde-se o esquema da figura 3.4 (capítulo 3, página 25), que apresentava os diferentes actores desta solução.

5.3.4.1 O Serviço *Web*

A explicação dos dois métodos de transporte de eventos a incluir neste serviço, foi dada na secção 5.3.1.1, restando apenas referir a respeito destes métodos que, além dos eventos serializados em texto, transportam também o identificador da sessão "sessionid" para que, no extremo servidor, a fonte das invocações possa ser identificada

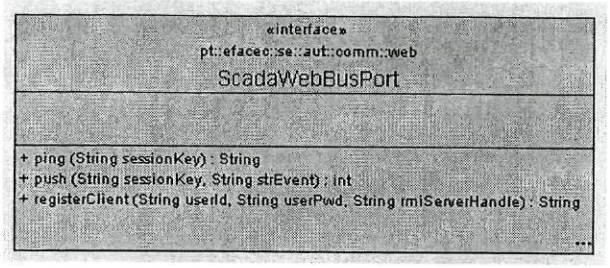

Figura 5.5: Interface do serviço *Web*

(e validada).

Relativamente ao outro método presente nesta interface, *registerClient*, ele será usado para notificação do extremo servidor, da existência de uma nova aplicação cliente, sendo, no retorno deste método, que a chave da sessão é entregue ao cliente.

Um canal *WEB.client* usará o procedimento *registerClient* no arranque, onde, para além dos dados relativos à sua validação (utilizador e chave), passará também um identificador de uma aplicação RMI (veremos à frente a sua utilidade).

Depois disto e sempre que existirem eventos a transportar para o servidor, será invocado o procedimento *push*, incluindo a chave de sessão. O procedimento *ping*, será, como já referimos, chamado periodicamente por cada aplicação cliente na implementação da receita "Actualizações Sucessivas".

5.3.4.2 Canais *WEB.server* — a WebBridge

Para cada canal *WEB.client*, tem sempre que existir um canal *WEB.server*, isto porque, a sua função é precisamente a de servir de intermediário entre este e um mundo legado local (por exemplo aplicações que usam o canal RMI).

Assim, uma aplicação que use um canal *WEB.server* para comunicação nalguns tópicos, usará também outro tipo de comunicação para os mesmos tópicos, irá assim, funcionar como ponte entre o mundo local e a aplicação remota (ver o exemplo de configuração dos tópicos para a aplicação "WebBridge" na secção 5.4).

Descrevendo as diferentes responsabilidades atribuídas a este tipo de canal:

- Disponibilizar um modo de acesso por parte do serviço *Web*.

  Esta questão é resolvida com recurso a uma interface RMI fixa (com um nome bem conhecido), o que permite que o serviço *Web* a descubra no "registry". Essa interface é um espelho da interface do serviço *Web* definida, sendo que o serviço serve apenas de intermediário, estando a lógica de negócio toda implementada pelo canal *WEB.server*.

  Esta solução, leva a que uma chamada remota a qualquer dos procedimentos que

compõem o serviço *Web*, dê origem a uma chamada idêntica por parte do serviço numa aplicação com um canal *WEB.server*.

- Autenticar aplicações clientes.

  A validação é feita aqui com base em pares utilizador/password autorizados.

- Gerar chaves de sessão.

  As chaves de sessão são geradas de forma a serem sempre únicas e identificam univocamente o canal *WEB.client* que está remotamente a registar-se. Por motivos de segurança, as chaves de sessão terão períodos de vida limitados findos os quais terão que ser renovadas.

- Existir simultaneamente no mundo local (no nosso caso como canal RMI).

  Como podemos ver no exemplo de configuração de tópicos para a aplicação "WebBridge" (figura 5.12), esta aplicação terá simultaneamente um canal RMI e um canal *WEB.server*, sendo que o comportamento do primeiro, obedece ao que foi descrito para a implementação RMI comum, ou seja, funciona como um publicador e subscritor comum de eventos e *data requests*.

- Gerir os tópicos publicados e subscritos por cada aplicação remota.

  O papel principal deste canal é o de propagar localmente o comportamento das aplicações *WEB.client* remotas. Neste sentido, esta aplicação terá que reagir a cada acção de publicação e subscrição de tópicos por parte de uma aplicação remota (com uma chave de sessão associada).

  Por exemplo, sempre que uma aplicação remota subscreva um tópico, esta aplicação vai ser notificada com um evento de configuração dinâmica do canal e deve reagir, registando uma *callback* local para todos os eventos desse tópico. Deste modo, os eventos entregues nessa *callback*, poderão ser encaminhados para a aplicação remota.

- Encaminhar os eventos entre mundos.

  Tal como foi descrito no capítulo anterior, os eventos são "marcados" pelo canal de comunicação que os transporta, sendo possível depois identificar o canal que transportou cada evento. Com base nesta característica, implementaram-se vários tipos de publicação: a publicação em todos os canais, apenas no de origem do evento ou em todos com excepção desse.

- Gerir *data requests* pendentes

  Quando uma aplicação remota publica um *data request*, é expectável que as aplicações locais (RMI) possam dirigir respostas para essa aplicação (implementando o conceito descrito na secção 4.1.3.1). Isto implica que o canal *WEB.server* guarde todos os *data requests* oriundos das aplicações remotas e os refaça no mundo local. Por outro lado, quando receber respostas (no canal local), terá que identificar o *data request* original a que se referiam e com base nesse, encaminhar as respostas para o invocador original do pedido, a aplicação remota.

No sentido contrário, passar-se-á exactamente o mesmo; os *data requests* efectuados localmente, terão que ser repetidos no canal *Web*, e as respostas que daí vierem, terão depois que ser dirigidas no mundo local, com base no *data request* original.

5.3.4.3 Canais *WEB.client*

Este canal vai desempenhar o papel de cliente do serviço *Web* descrito. Vai, por isso, ser o responsável pela implementação das invocações periódicas do serviço *Web*.

Na configuração deste tipo de canal (recorde-se o exemplo da figura 4.6), incluem-se dois parâmetros cujo significado apresentamos agora:

- `serviceAddress`:

  Corresponde ao URL onde o serviço *Web* está instalado. O facto de ser configurável deixa em aberto a possibilidade de se fazer algum balanceamento de cargas, por exemplo tendo mais do que uma "WebBridge" em diferentes máquinas.

- `rmiServerHandle`:

  Corresponde ao nome com o qual o canal *WEB.server*, que atenderá este cliente, está registado no "registry". Também aqui fica espaço para a existência de mais do que uma aplicação "WebBridge", desta vez na mesma máquina.

Tal como na implementação *WEB.server*, este canal vai ter de criar um componente especial (ver a classe `WebComponent` no diagrama de classes) para receber os eventos de configuração do próprio canal, sendo com base nesses eventos e na publicação e subscrição dos próprios tópicos, que manterá as associações de publicadores e subscritores.

5.3.5 Desenho do Pacote `comm.web`

Justificada que está a necessidade de ter dois comportamentos complementares, um para as aplicações situadas "do lado de fora" da *firewall* (invocadoras de serviços *Web*) e outro para as aplicações "do lado de dentro", que fazem a ponte entre as primeiras e as aplicações locais do sistema legado, passamos agora ao seu desenho propriamente dito.

Com base nesta separação, organiza-se esta secção, demonstrando inicialmente a implementação comum aos dois sub-canais (*WEB.server* e *WEB.client*), seguida das classes específicas do comportamento *WEB.client* e terminando com as classes apenas *WEB.server*.

5.3.5.1 Classes comuns

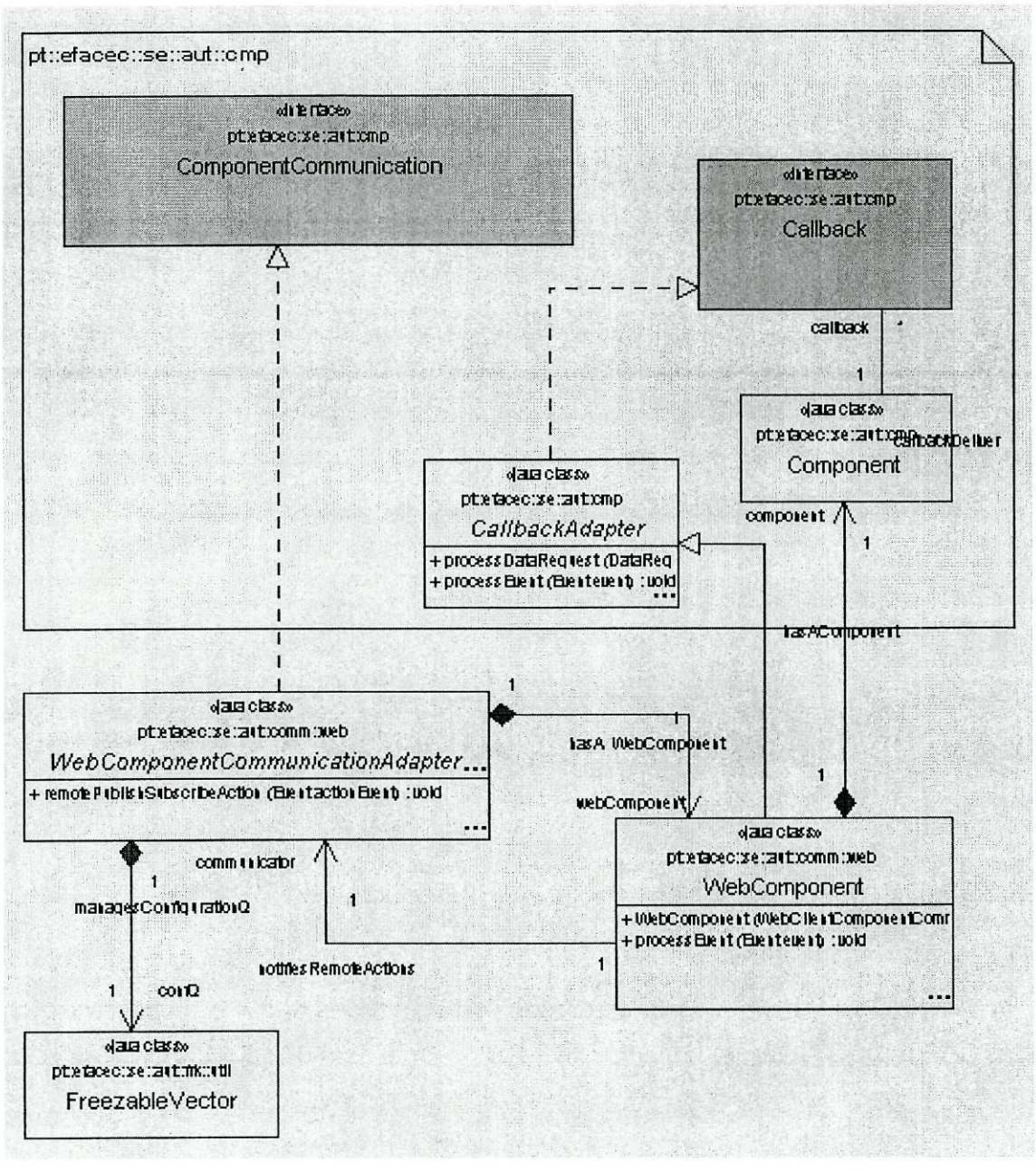

Figura 5.6: Pacote `comm` — Classes comuns *WEB.server* e *WEB.client*

Como já foi referido, para tratamento da gestão dinâmica do canal a implementação é comum aos extremos servidor e cliente da implementação *Web*.

Na implementação deste mecanismo, destacam-se as seguintes classes:

- `WebComponentCommunicationAdapter`

  Implementa a interface `ComponentCommunication`, naquilo que existe de comum

entre as implementações servidor e cliente do canal, nomeadamente a gestão da fila de eventos de configuração, que vai permitir que sempre que se estabeleça uma ligação *Web*, os extremos troquem as suas configurações em termos de publicação e subscrição de tópicos.

- `WebComponent`

  Esta classe é criada pela anterior, acumulando as funções de componente e de *callback*, para recepção dos eventos de configuração dinâmica do canal.

5.3.5.2 Classes *WEB.server*

- `WebBridgeRmiInterface` e `WebBridgeRmiServer`

  Definem e implementam a interface RMI fixa que é disponibilizada para acesso por parte do serviço *Web*. Repare-se, que ambas derivam das definidas para o canal RMI de componentes comuns (interface `RmiInterface` e classe `RmiServer`), mas acrescentam-lhe os novos métodos de `ping` e `registerClient`, decorrentes dos existentes no serviço *Web*.

- `WebBridgeRmiServer`

  Implementa a gestão das ligações a aplicações remotas. Nomeadamente:

  - Cria e destrói filas de eventos para cada nova sessão.
  - Valida pedidos de novas sessões e cria as chaves respectivas.
  - Implementa o encaminhamento de publicações locais para as filas de saída correspondentes (com base na lógica da publicação e subscrição).
  - Derivando de "RmiServer", reutiliza a sua implementação do método `push` para colocação de eventos remotos na fila de processamento da aplicação.

- `WebServerComponentCommunication`

  Implementa o que este canal tem de específico (relativamente ao descrito para a classe pai `WebComponentCommunicationAdapter`), ou seja, a criação do objecto `WebBridgeRmiServer` e o encaminhamento de eventos e *data requests* para a mesma.

5.3.5.3 Classes *WEB.client*

- `WebClientComponentCommunication`

  Esta classe, tal como a `WebServerComponentCommunication`, implementa aquilo que é específico relativamente à classe pai `WebComponentCommunicationAdapter`.

  Por um lado, é responsável por criar o filtro de eventos e o cliente do serviço *Web*. Por outro, irá servir, ela própria de cliente da fila de eventos de saída,

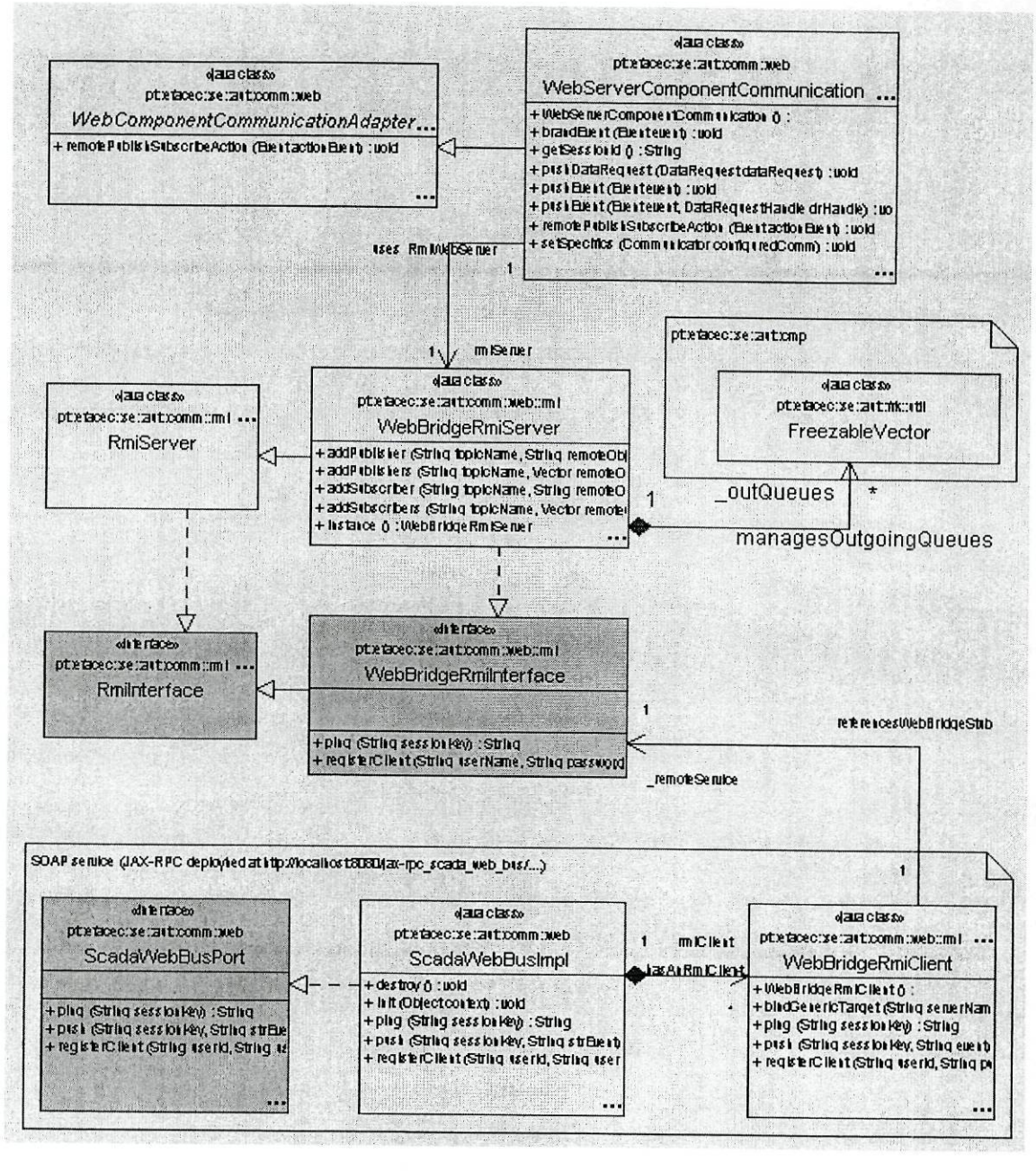

Figura 5.7: Pacote comm — Implementação *WEB.server*

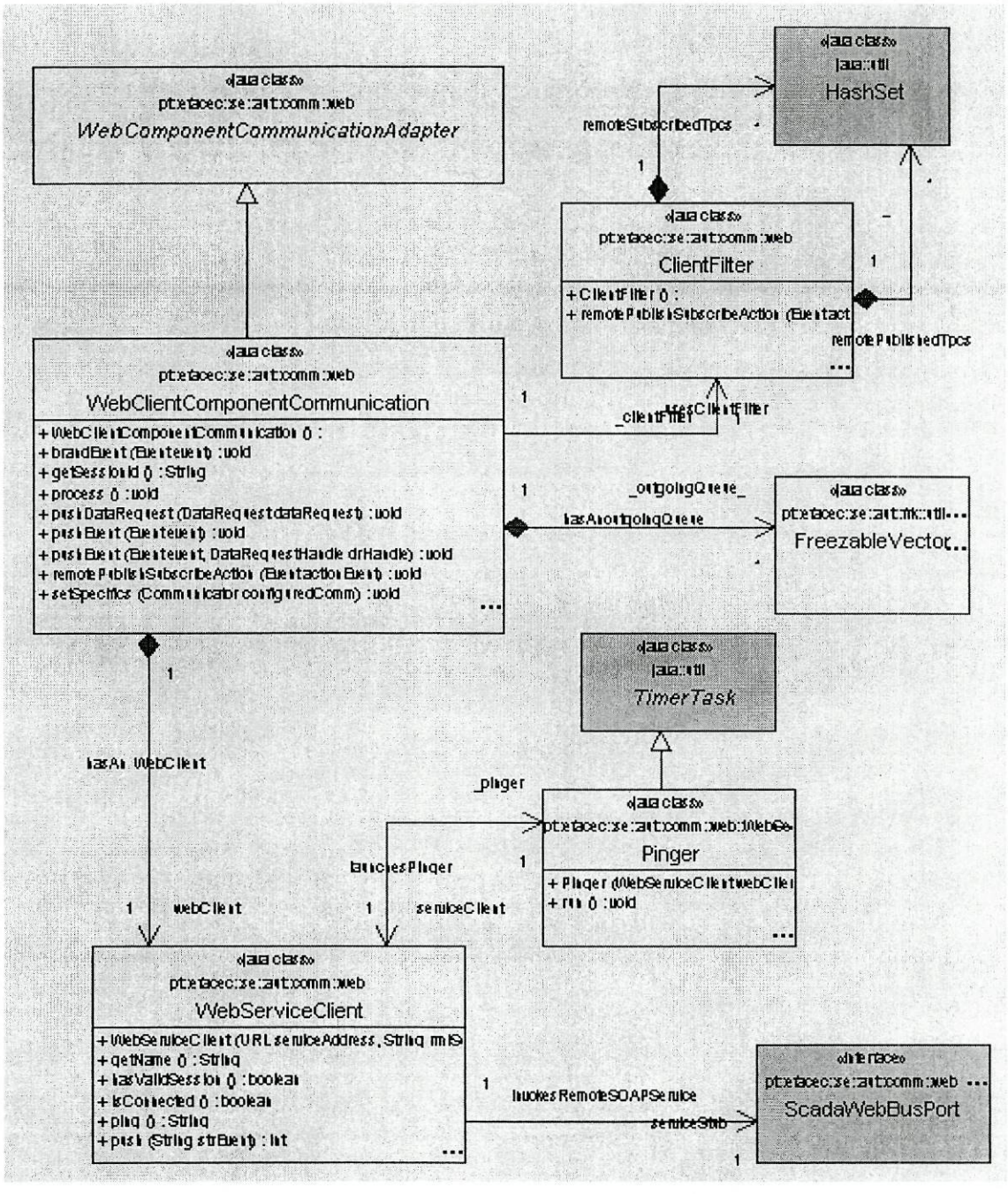

Figura 5.8: Pacote comm — Implementação *WEB.client*

sendo portanto aqui, que o mecanismo periódico de invocação dos serviços *Web*, virá buscar os eventos a transportar para o servidor.

- `ClientFIlter`

  Esta classe trata as alterações de configuração do canal, com base nos eventos de configuração recebidos, mantendo uma lista de tópicos publicados e subscritos remotamente (pela aplicação *WEB.server*).

  Com base nestas listas, implementa métodos de filtragem que o canal usará para decidir se deve ou não transportar eventos ou *data requests*.

- `WebServiceClient`

  Esta é a classe que implementa a componente cliente dos serviços *Web*, mantém um temporizador para que essas invocações sejam periódicas, tal como previsto.

5.3.5.4 Casos de uso

A figura 5.9 descreve, num diagrama de sequência, a entrada de um novo publicador no mundo *Web* (válido também para outro tipo de alteração de configuração).

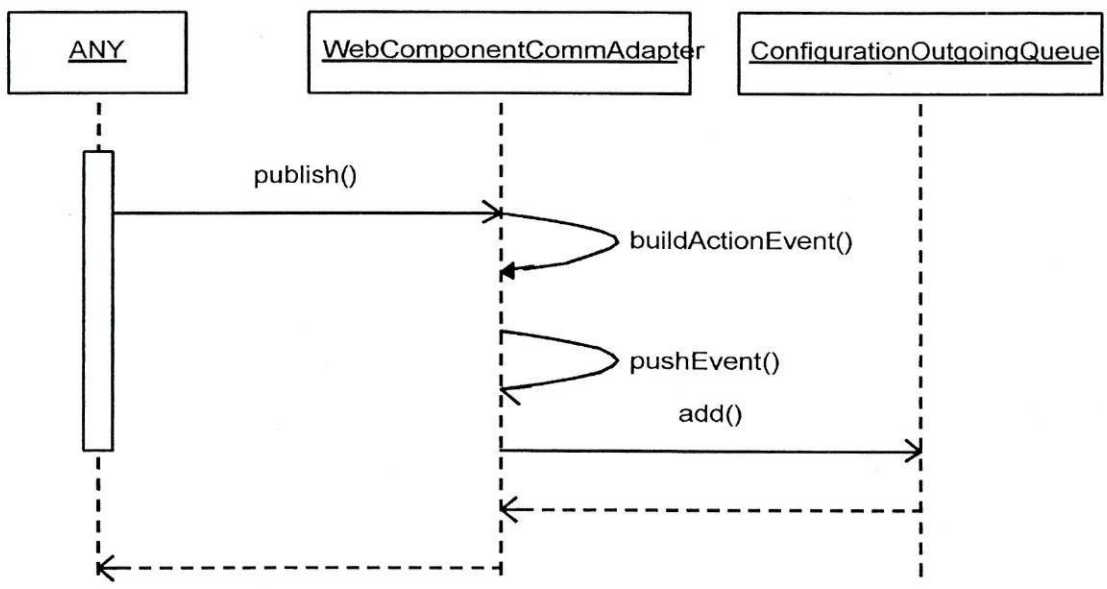

Figura 5.9: Novo publicador *Web*

Relativamente à publicação de um evento pelo cliente, o cenário pode ser visto na

figura 5.10[9].

Finalmente, para a publicação de um evento pelo servidor, temos o diagrama da figura 5.11.

5.4 Configuração de Canais de Comunicação por Tópicos

Como se pode ver na figura 5.12, é ao nível do tópico que se define em que tipo de canal é que a aplicação vai falar. Podem ter-se configurações de, mais do que um canal por tópico, ou de apenas um, conforme as necessidades das diferentes aplicações.

[9] A linha ao alto separa a computação efectuada na máquina do cliente *Web*, da do servidor.

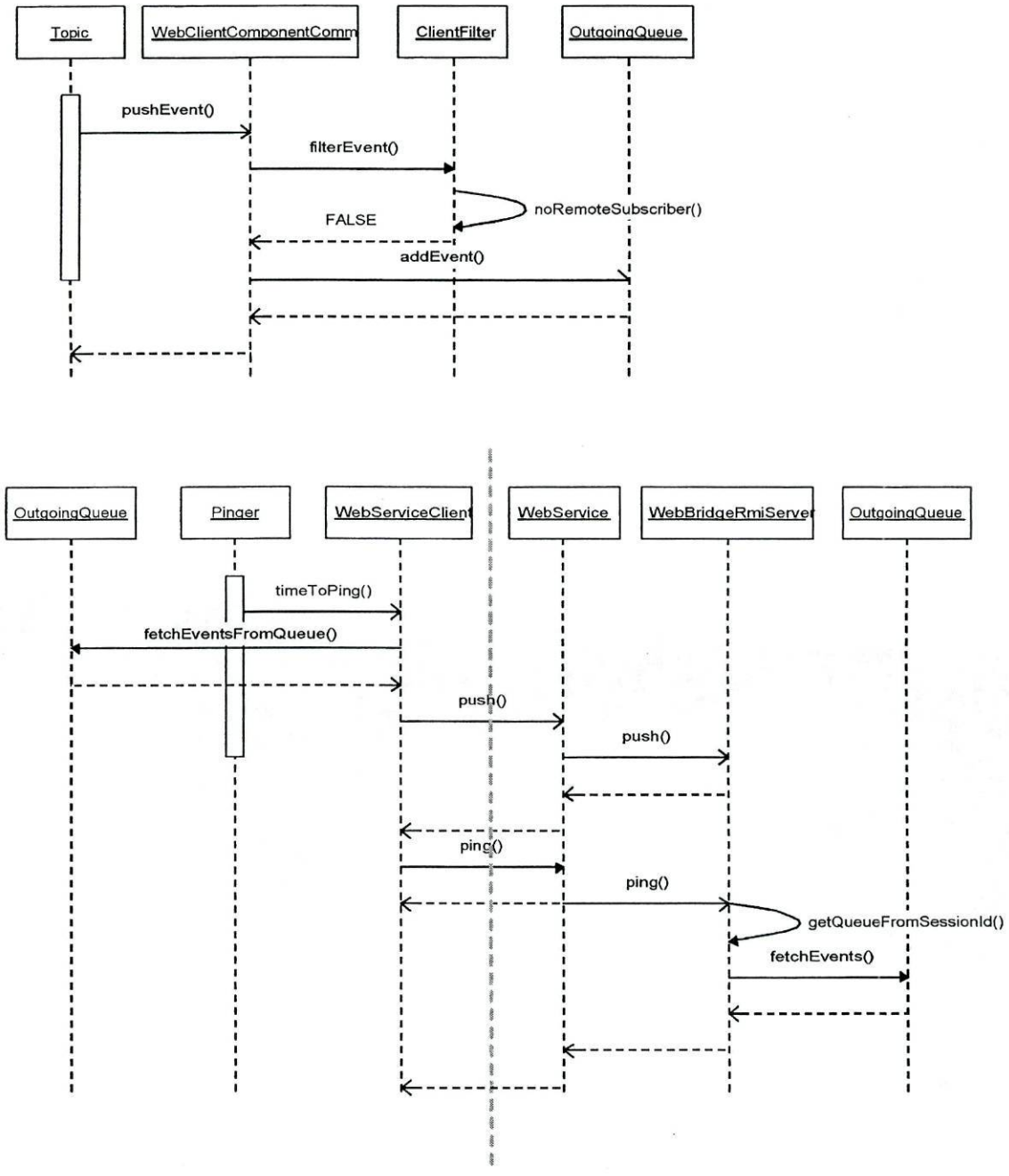

Figura 5.10: Publicação de evento por parte do *WEB.client*

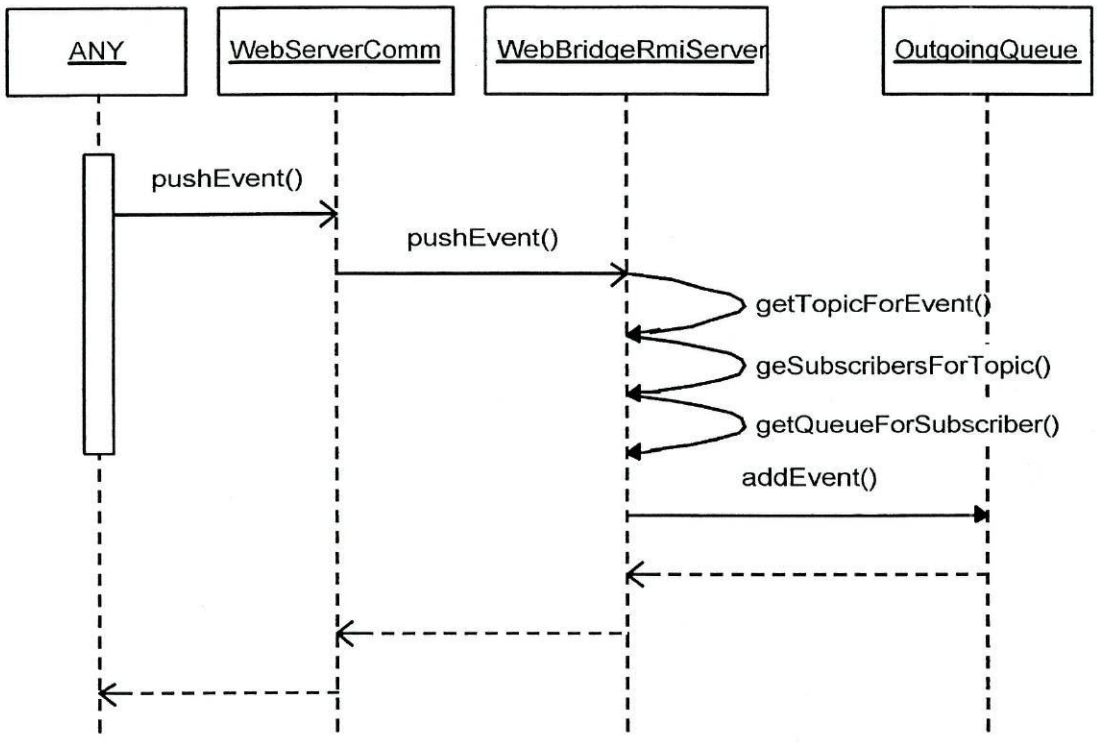

Figura 5.11: Publicação de evento por parte do *WEB.server*

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
        <topic name="Scada">
                <communicator commType="WEB.server"/>
                <communicator commType="RMI"/>
        </topic>
        <topic name="Areas">
                <communicator commType="WEB.server"/>
                <communicator commType="RMI"/>
        </topic>
        <topic name="Construction">
                <communicator commType="WEB.server"/>
                <communicator commType="RMI"/>
        </topic>
        <topic name="CommManagement">
                <communicator commType="WEB.server"/>
                <communicator commType="RMI"/>
        </topic>
</configuration>
```

Figura 5.12: Configuração dos tópicos da aplicação WebBridge

5.5 Resumo

Com este capítulo fica terminada, nas suas duas versões (RMI e *Web*), a implementação do *Framework* de comunicações a que nos propusemos para demonstração da tese. Far-se-á agora a explicação do ambiente de testes usado na captura de resultados, para serem analisados no final.

Capítulo 6

Exploração e Avaliação da Arquitectura

Construímos até aqui um *framework* de computação distribuída com dois canais de comunicação, RMI e *Web*. Definiremos agora um ambiente de teste que faça uso do *framework* definido e que permita a avaliação do cumprimento, ou não, dos objectivos a que nos propusemos. Com base nesse ambiente, far-se-ão testes e a análise dos respectivos resultados.

6.1 Definição do Ambiente de simulação

Até este momento conseguiu montar-se uma arquitectura que permite construir sistemas distribuídos. Utilizando o *framework* que definimos, podem implementar-se aplicações com componentes que comunicam entre si, segundo uma lógica de publicação e subscrição baseada em tópicos. Além disso, o transporte de eventos entre componentes pode ser feito numa tecnologia típica de rede local (RMI) ou "através" da *Internet* "usando tecnologias típicas da *Web*".

Mas este facto, por si só, ainda não serve de prova para a tese que defendemos neste texto. Não está provado ainda, que é possível com a implementação *Web* do canal, cumprir todos os requisitos de comunicação apresentados para as aplicações SCADA.

Alguns desses requisitos, que apresentámos no capítulo 2, foram sendo cumpridos pelas opções tecnológicas e de implementação que foram tomadas. É o que se passa com a:

- interoperabilidade;
- portabilidade;
- garantia de entrega de todos os eventos;

- sequencialidade dos mesmos;
- possibilidade de implementação de interfaces gráficas ricas;
- bidireccionalidade;

Já relativamente às restrições quantificadas relacionadas com as latência e largura de banda típicas ou máximas, ainda nada está provado (requisitos definidos na secção 2.2.5).

Definimos, por isso, um ambiente de simulação baseado no próprio *framework* com o objectivo de testar especificamente essas duas "grandezas" de forma controlada. Nas próximas secções, descreve-se precisamente esse ambiente e os cenários de teste efectuados.

6.1.1 Cálculo das Grandezas

Para fazer as medições relacionadas com a latência e a largura de banda suportadas pelo canal de comunicação, poderiam ter-se construído aplicações especiais, apenas para esse efeito ou incluir no próprio canal essa capacidade auto-avaliativa.

Também neste aspecto, optámos por incluir a maior parte da implementação no próprio *framework* por dois motivos essenciais:

- Vai permitir que todos os componentes "ligados" possam fazer uso dessa informação.
- Poderá vir a ser útil na implementação de mecanismos de auto-regulação e adaptabilidade do próprio canal. Recorde-se que estes mecanismos foram adiantados como um requisito desejável dada a instabilidade da *Internet* enquanto "meio" de comunicação (ver secção 2.2.5).

Por outro lado, como o cálculo dessas características implica, ele próprio, algum tráfego adicional no próprio canal, optámos por incluir no *framework* os meios para o fazer, mas deixando a cargo das aplicações a decisão de os usar ou não.

Resta esclarecer que, sendo o "evento" a mais pequena mensagem que dois componentes podem trocar entre si, é precisamente com base nos eventos que mediremos quaisquer grandezas relacionadas com o canal. Ou seja, se falamos de latência, falamos do tempo que um evento demora a percorrer o "caminho" entre dois componentes, se falamos em largura de banda, estamos a falar na quantidade (e tamanho) de eventos que podem ser transaccionados por unidade de tempo, entre dois componentes.

6.1.1.1 Latência

A forma mais simples de implementar uma medição sistemática da latência entre dois componentes, será obviamente a medição do tempo que decorre desde a invocação de

um *data request* por parte de um deles, até à recepção da resposta enviada pelo outro. Assumindo que não é efectuada nenhuma operação especial com o pedido e que a construção da resposta é quase imediata, essa medição de tempo será, aproximadamente, o dobro da latência que pretendíamos medir.

A melhor forma de garantir homogeneidade nas medidas, será precisamente, a implementação de um serviço especial, suportado pelo framework, com as seguintes características:

- invocação do pedido (publicação de um *data request* para um URN especial — *Ping*);
- implementação da *callback* para a recepção do pedido, com a construção automática da resposta e sua publicação dirigida;
- implementação da callback para a recepção da resposta, com a contabilização do tempo total e entrega do mesmo aos interessados;

Para suportar os objectivos enumerados em cima, foi criada uma classe no pacote `cmp` — `PingCallback`, que se apresenta esquematicamente na figura 6.1.

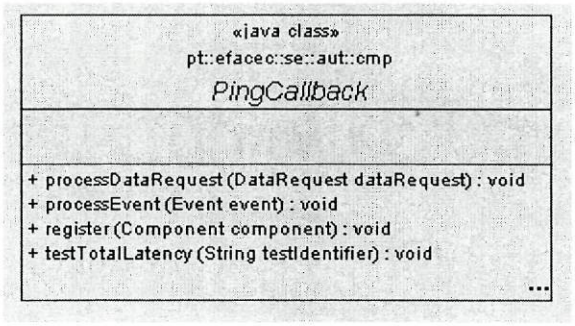

Figura 6.1: Classe PingCallback

Acrescente-se ainda que qualquer componente interessado nos valores de latência (ou na largura de banda como veremos na próxima secção), deve derivar desta classe e implementar um método puramente abstracto *totalLatencyAvailable* onde serão entregues os valores assim que disponíveis, depois da invocação de *multiPing*.

Finalmente, é possível controlar o tamanho dos eventos a usar no pedido e na resposta e repetir o teste múltiplas vezes se se pretender (sendo que neste caso a latência será entregue várias vezes depois do pedido efectuado).

6.1.1.2 Largura de Banda

Com base na mesma ferramenta usada para o cálculo da latência, implementou-se uma variante em que os eventos a usar na invocação do serviço "Ping" são progressivamente maiores (entre limites estabelecidos pelo utilizador).

Desta forma, com o registo da latência para cada situação, obter-se-á um gráfico com a latência em função do tamanho dos eventos usados. O pico desse gráfico será o ponto da maior largura de banda na comunicação entre dois componentes, usando o canal.

A divisão do"tamanho dos eventos" pela latência, dará a largura de banda útil fornecida pelo canal em cada situação. É de esperar que este valor cresça com o aumento dos eventos a usar, até ao limite permitido pela rede. A partir desse momento, é expectável que estabilize até o desempenho do próprio canal se degradar (pela dimensão dos eventos a processar), diminuindo progressivamente a partir daí.

Como se pode ver na classe `PingCallback`, para se efectuar o pedido de medição de largura de banda é necessário invocar o método *checkBandWidth* e esperar pelos diferentes valores em *bandWidthReading*.

Deve ter-se em conta o peso das diferentes camadas de comunicação e transporte na própria largura de banda. Ou seja, o cálculo da largura de banda, efectivamente necessária para transportar os eventos nas diferentes simulações, terá sempre que ser feito com base no inflaccionamento da medida, dum factor relacionado com o tipo de transporte utilizado.

6.1.1.3 Componente global de testes — *Simulator*

Não só como aplicação de teste e validação do próprio framework ao longo da implementação do mesmo, mas também como aplicação que mede as grandezas já referidas e captura esses resultados, o *Simulator* é uma aplicação com uma interface muito simples que traduz exactamente as características do modelo de publicação e subscrição.

Veja-se na figura 6.2, a interface da aplicação de simulação e teste. Como se pode verificar na imagem, é possível publicar ou subscrever tópicos, anular essas operações, é possível também registar *callbacks* para a recepção de determinados eventos (altura a partir da qual se passam a ver, em formato de texto, esses eventos quando entregues à aplicação).

Além das opções descritas, esta aplicação tem dois botões que permitem iniciar as medições respectivamente da latência e da largura de banda, sendo que os parâmetros para estas operações são introduzidos nos campos de edição para o efeito. A saber:

- *Test Id*: para catalogar o teste a ser efectuado (terá importância para efeitos de captura de resultados em ficheiro);
- *Ev. Size / 1st Size*: para a latência, define o tamanho dos eventos a usar; no caso da medição ser de largura de banda, estipula o tamanho do primeiro evento a ser publicado.
- *Max Size*: interessa apenas para a medição da largura de banda e impõe o tamanho máximo da série de eventos a trocar.

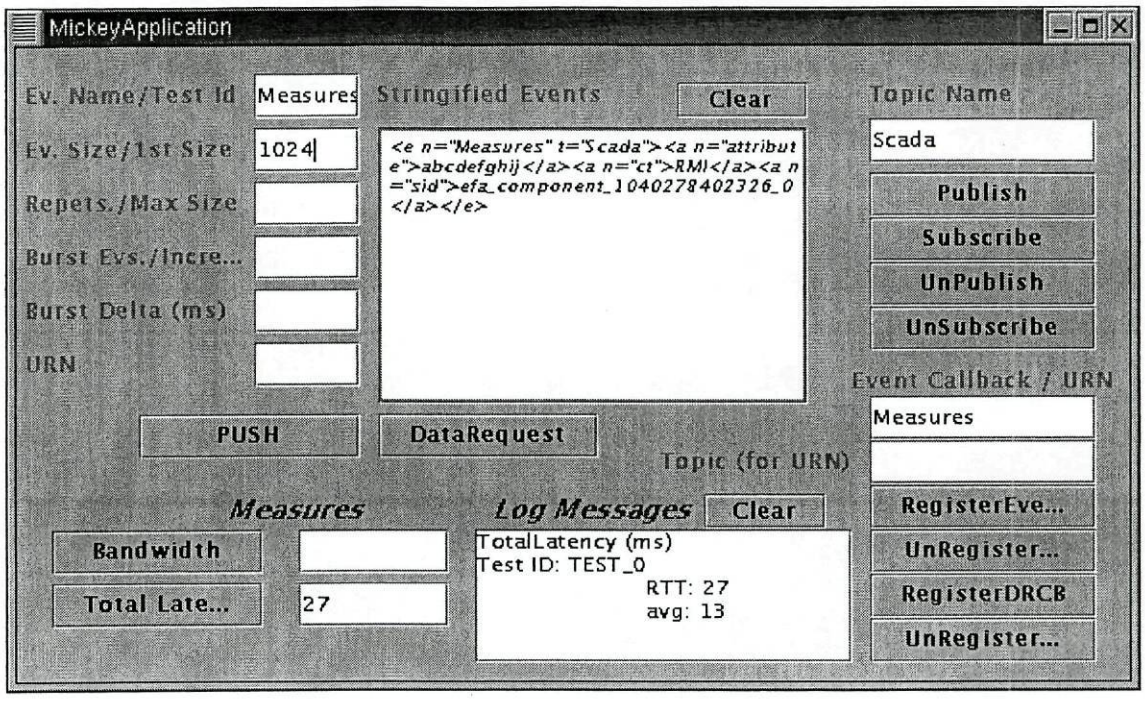

Figura 6.2: Interface gráfica do Simulator

- *Increment*: contém (para a largura de banda) o incremento no tamanho dos eventos a serem trocados.

Esta aplicação guarda em ficheiro todos os resultados das medições efectuadas, por forma a ser possível depois avaliar esses mesmos valores.

Finalmente, como é feita usando o framework, é possível colocar a mesma aplicação a correr em ambiente RMI ou em ambiente puramente *Web*. Bastando, para isso, configurar os seus tópicos para os canais desejados (ver secção 5.4).

6.2 Configurações de Teste

Por simplicidade, associa-se o nome "Tom" à aplicação *Simulator* que irá correr nos nossos testes com todos os tópicos a comunicarem pelo canal RMI. E "Jerry" à aplicação idêntica, mas que comunicará apenas como um cliente *Web*.

Assim, para avaliar a comunicação entre duas aplicações RMI, bastará colocar duas instâncias do "Tom" a correr, enquanto que para avaliar a comunicação entre uma aplicação remota (cliente *Web*) e uma local, poremos o "Tom" e o "Jerry" a correr. Estas configurações encontram-se esquematizadas na figura 6.3.

Chamaremos "RMI" ao conjunto de teste Tom—Tom e "WEB" ao conjunto de teste Jerry—WebBridge—Tom.

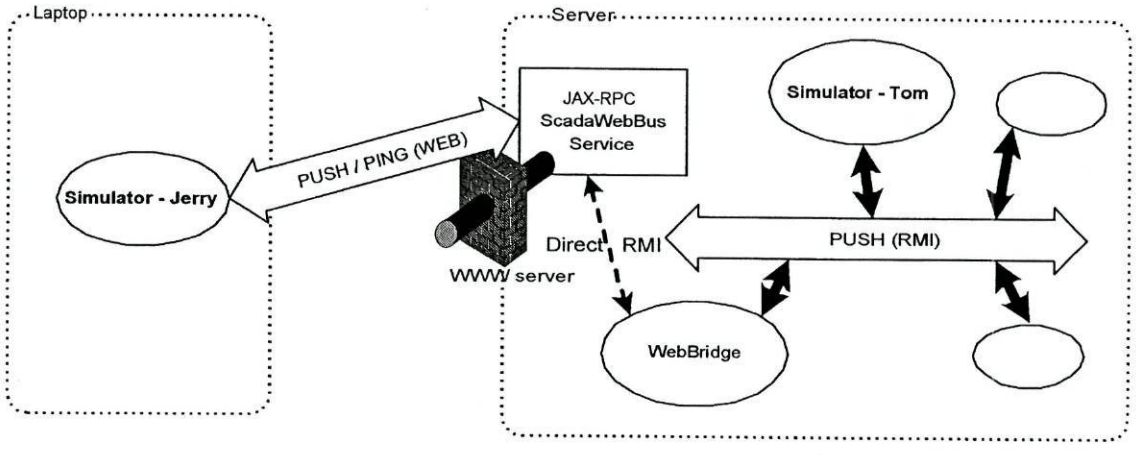

Figura 6.3: Tom & Jerry — Teste WEB

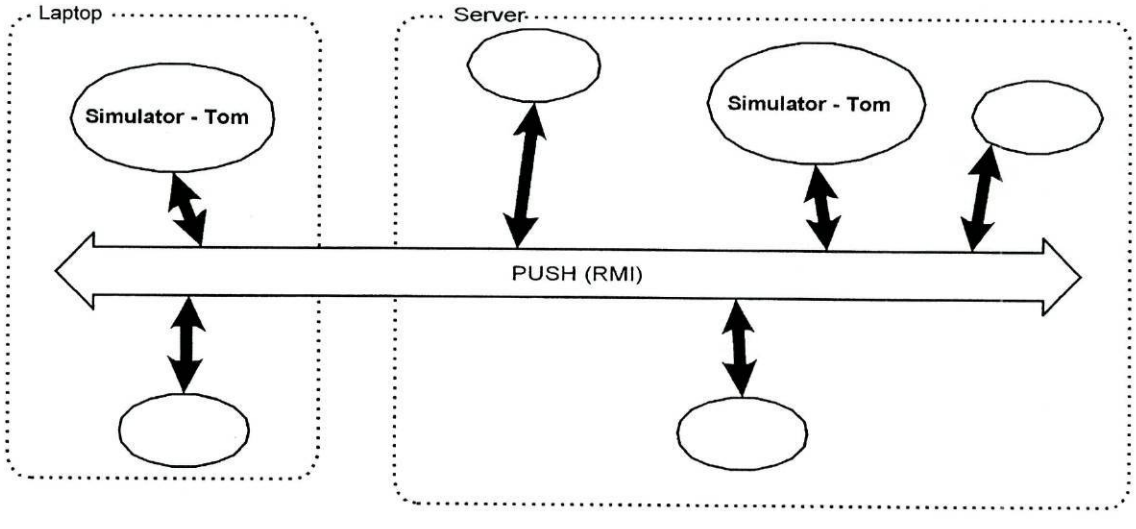

Figura 6.4: Tom & Tom — Teste RMI

6.2.1 Cenários de Teste

Os testes foram efectuados de forma controlada e sistemática, definiu-se um "guião" de cenários para capturar um conjunto vasto de "amostras" das grandezas tratadas, latência e largura de banda.

Os diferentes cenários, representam um leque amplo de situações, deduzidas a partir dos requisitos de comunicação assumidos, indo desde o funcionamento típico, até ao funcionamento extremo. Para esses cenários, estudam-se diferentes configurações *Web* e a implementação RMI de referência.

6.2.1.1 Latência

Neste cenário pretendemos medir, para diferentes tamanhos de eventos, o tempo que eles demoram a ser transportados entre componentes.

A dimensão dos eventos será variada, de forma a incluir todos os regimes expectáveis de funcionamento descritos na secção 2.2. Começando com o pequeno evento com 200 bytes, relativo à informação de *login* ou de um alarme, até aos picos de utilização, por exemplo, para uma avalanche de 2000 alarmes quase em simultâneo (2000*200=400000 bytes).

Com este teste pretende-se avaliar, para a implementação RMI e para a implementação *Web* do canal, quais as diferenças essenciais e se as limitações da *Web* impossibilitam o cumprimento dos requisitos definidos.

Resta apenas explicar que na configuração "WEB", serão repetidos os testes para diferentes temporizações entre as "actualizações sucessivas" do cliente *Web* ou, mais especificamente, entre invocações do serviço *Web* que sustenta o canal. Teremos assim medidas:

- WEB_1: para um segundo entre actualizações;
- WEB_2: para 2 segundos entre actualizações;
- WEB_5: para 5 segundos entre actualizações;
- WEB_10: para 10 segundos entre actualizações;

Pretende-se assim, avaliar o efeito deste parâmetro no desempenho do próprio canal.

6.2.1.2 Largura de Banda

Para os testes de largura de banda, pretendemos aferir a partir de que momento é que o processamento do próprio canal estrangula a sua capacidade de transmissão.

Para condições físicas de largura de banda superiores às necessárias em todos os testes (LAN a 10Mbit/s), pretende-se descobrir, a partir de quando é que as actividades de transmissão e despacho de eventos, começam a não ser suficientemente rápidas para "atender" a todos os pedidos de transmissão.

Ao contrário do cenário anterior, em que se pretendeu avaliar a latência para algumas situações típicas de quantidade de informação a transferir, aqui pretendemos fazer um acompanhamento fino da evolução da largura de banda efectiva, com o tamanho dos eventos a transferir.

Os tamanhos de eventos limite serão, mais uma vez, idênticos aos usados para os testes da latência, mas desta vez com um incremento mais fino de 1000 bytes.

Pretende-se no final poder observar graficamente essa mesma evolução, descobrindo o pico da largura de banda, em cada situação.

Faremos também aqui, diferentes testes para diferentes configurações do período entre "invocações *Web*", por parte do cliente.

6.2.2 Condições Físicas para os Ensaios

- Um portátil (PIII a 1GHz com 128Mb de RAM);
- Um PC (AMD-thunderbird a 1GHz com 256Mb de RAM);
- Ligados por uma rede *Ethernet* de 10Mbit/s;

6.3 Resultados

Relativamente às medições de latência, apresentaremos os resultados tabularmente, uma vez que é reduzido o número de amostras. Já para os resultados relativos aos ensaios de largura de banda, serão apresentados graficamente dado o elevado número de amostras. Em ambos os casos, faremos uma análise crítica dos mesmos.

6.3.1 Latência

Da análise dos valores desta tabela, podem tirar-se várias conclusões:

- Nas configurações *WEB*, para o transporte de eventos de dimensão útil menor ou igual a 100k, a latência média é muito próxima do período entre actualizações,

	Configurações *(latência em ms)*				
Dimensão (bytes)	WEB_1	WEB_2	WEB_5	WEB_10	RMI
200	783	1942	4990	8414	20
1000	958	1607	4808	8877	22
10000	969	1954	4963	5939	52
100000	1751	2228	4958	7057	1027
400000	12932	13722	15053	26707	13753

Tabela 6.1: Resultados dos testes de latência

sendo menor que este na maior parte das vezes[1]. Isto sugere que para eventos de dimensão nesta gama a influência dos mecanismos de serialização/desserialização dos eventos baseados em XML, bem como os mecanismos de despacho dos eventos fornecidos pelo framework, são quase irrelevantes no processo.

- Para eventos grandes (100k e 400k úteis[2]), esta regra deixa de ser válida o que mostra que nestas condições de teste, o processamento relativo ao canal começa a ter um impacto não desprezável. Como pode ser visto na coluna de resultados relativa aos cenários "RMI", este fenómeno também aí pode ser observado. Este facto era esperado uma vez que as implementações são idênticas, mudando apenas a parte relativa ao transporte dos eventos propriamente dita.

- Relativamente ao cumprimento ou não dos requisitos no que diz respeito à latência, pode concluir-se que eles são cumpridos para todos os ensaios com um período entre actualizações que não seja, já de si, superior à latência aceitável. Ou seja, só para os ensaios com actualizações de 10 em 10 segundos, é que os requisitos impostos para latência não foram cumpridos.

  Para os ensaios de picos (correspondentes aos eventos com maior dimensão), pode concluir-se que o comportamento é o esperado, com um atraso no tratamento geral do evento, mas com recuperação da normalidade depois de passada a situação de "crise".

- No que diz respeito à comparação entre o desempenho das configurações *WEB* com o desempenho da configuração RMI, a diferença principal situa-se no comportamento para transacções pequenas, nesta situação a implementação RMI mostra-se muito mais reactiva, transferindo quase de imediato os eventos publicados.

  Já para situações de "crise", a implementação *Web* tem um comportamento semelhante à RMI, uma vez que os períodos entre actualizações correspondem a processamento efectivo nos extremos do canal.

[1] Os valores obtidos são uma média de várias leituras seguidas; como existe uma invocação de *push* e uma invocação de *ping* por "período", pode haver simultaneidade no envio de pedidos novos e recepção de respostas anteriores.

[2] Note-se que os testes foram efectuados por forma a que a dimensão considerada para os eventos fosse apenas a parte com interesse útil para as aplicações, não contando, por isso, com atributos adicionais colocados pelo próprio framework, ou com a informação relativa ao transporte do evento.

O facto de, nos cenários de eventos maiores, as configurações *WEB_1*, *WEB_2* e *RMI* terem tempos semelhantes (à volta dos 13 segundos), mostra que o incremento grande nos atrasos, se situa necessariamente no código comum às duas implementações, nomeadamente ou no *parsing*, ou no despacho dos eventos. O primeiro, é, como pudemos observar, uma implementação SAX (da *Sun*), e o funcionamento do segundo (despacho de eventos) é idêntico, quer se tratem de eventos grandes ou pequenos, isto indicia que a implementação do parser, reduz dramaticamente o seu desempenho, para *streams* XML com mais de 100kbytes.

Outra conclusão que se pode tirar, do facto de estes cenários terem tido tempos equivalentes (*WEB_1*, *WEB_2* e *RMI* para eventos de 400kb), é a de que a invocação do serviço *Web* propriamente dita, não é muito relevante no tempo total. O que aliás já era esperado, uma vez que o serviço Web tem uma interface muito simples, o que faz com que o *parsing*, nesta fase, seja irrelevante (só está definido um argumento do serviço e é um pedaço de texto).

Concluindo, do ponto de vista da latência, as diferenças entre as implementações só serão relevantes se se pretender uma garantia de transmissão "imediata", de resto, para os requisitos que impusemos ao canal e dependendo do período entre actualizações, a versão *Web* cumpre-os.

- Finalmente, observa-se alguma inconstância nas medidas feitas com as configurações *WEB*; esta inconstância facilmente se compreende, se recordarmos que o início do cenário apesar de despoletado manualmente (carregando no botão relativo à latência), só corresponderá a uma primeira transmissão na invocação do serviço *Web* seguinte o que, no limite, pode acontecer quase imediatamente a seguir ou um período inteiro depois.

6.3.2 Largura de Banda

A análise dos gráficos, para as diferentes configurações, leva-nos a conclusões semelhantes às que pudemos observar para os testes de latência, embora aqui se possa verificar, muito mais detalhadamente, a transição para um regime em que o processamento do canal de transmissão começa a estrangular a própria transmissão.

Facilmente se compreende que, depois desse pico de largura de banda atingido, haja um declínio progressivo da mesma pela utilização cada vez maior do CPU, por parte do canal. Isto explica-se, se observarmos que o tempo de processamento gasto em actividades do canal (nomeadamente o *parsing* do XML dos eventos), não cresce linearmente com o tamanho dos eventos, acelera o seu crescimento a partir de certa altura, vindo a assumir um papel cada vez mais importante no tempo total da transmissão.

Podemos ver que esse ponto de "estrangulamento", acontece para eventos cada vez maiores, à medida que o período entre invocações *Web* aumenta, sendo que nas figuras 6.5 e 6.6 (*WEB_1* e *WEB_2*) esse estrangulamento é bem visível e nas outras configurações *WEB* quase não se faz sentir (figuras 6.7 e 6.8).

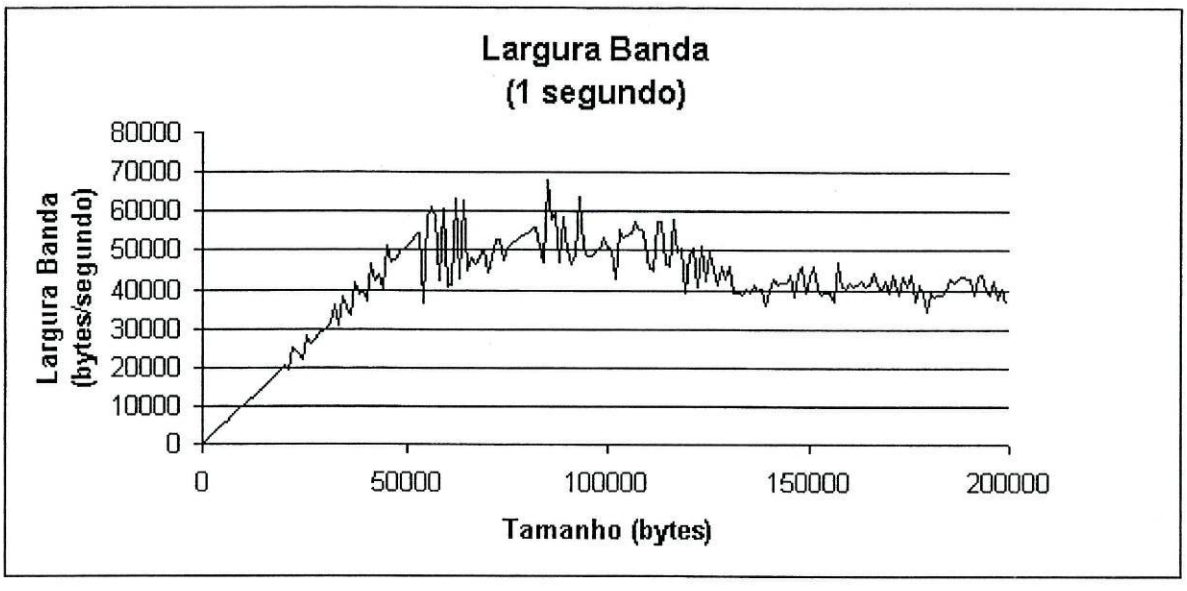

Figura 6.5: Largura de banda — Configuração WEB_1

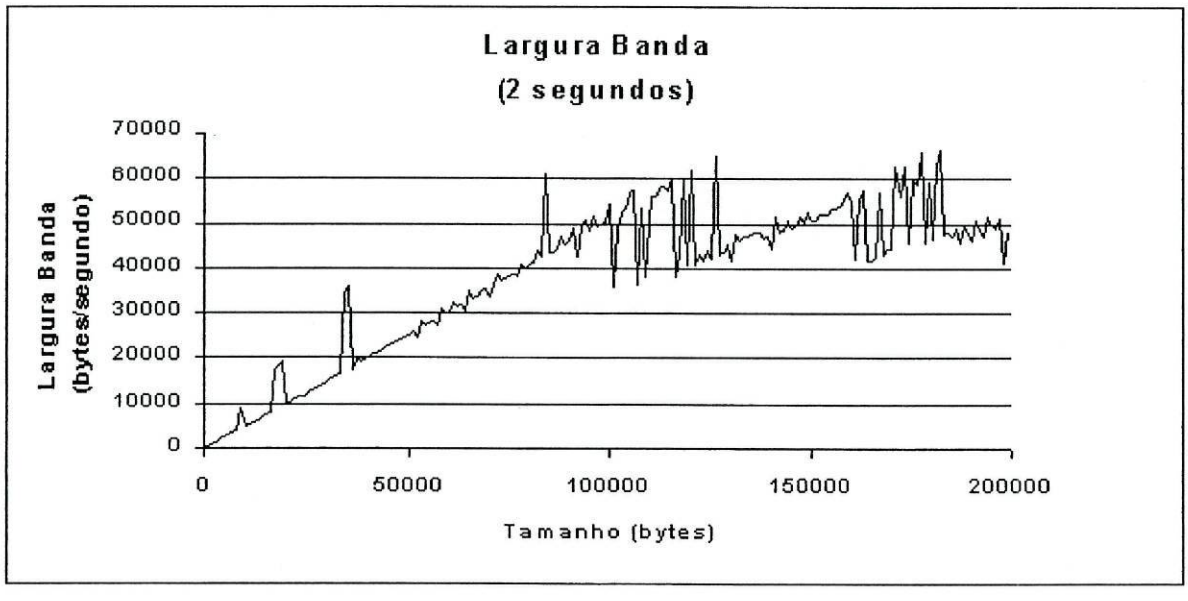

Figura 6.6: Largura de banda — Configuração WEB_2

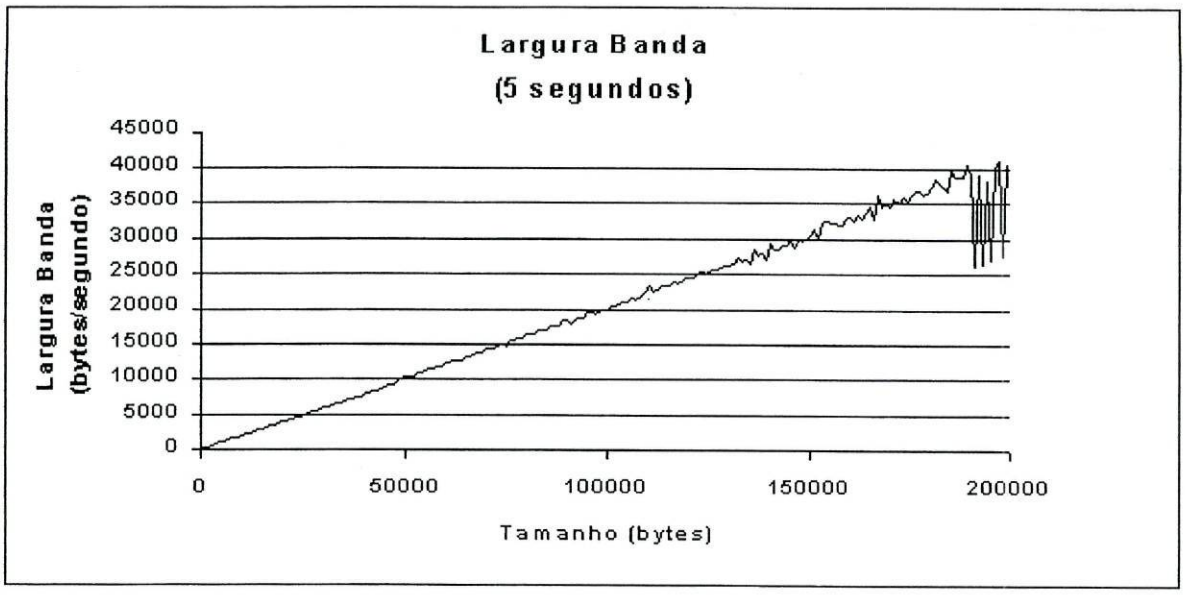

Figura 6.7: Largura de banda — Configuração WEB_5

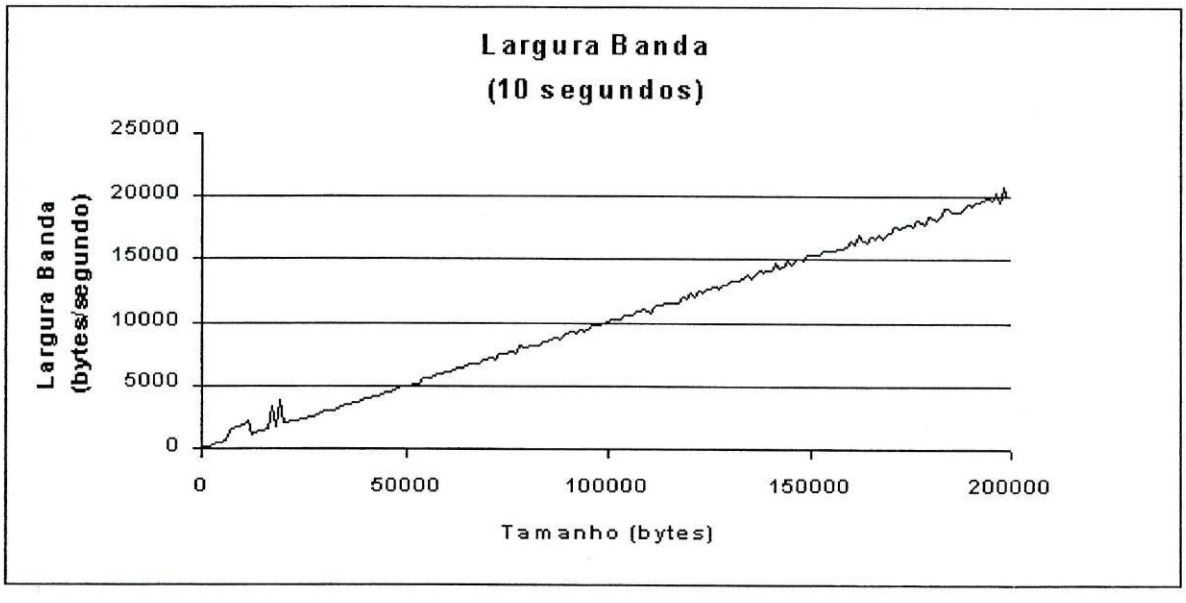

Figura 6.8: Largura de banda — Configuração WEB_10

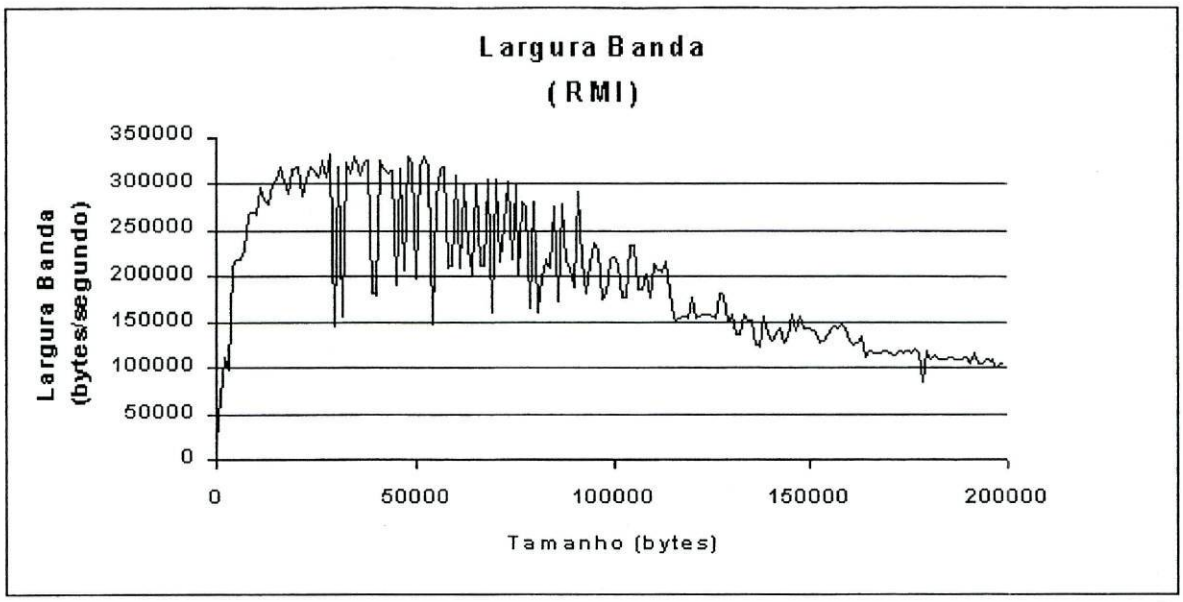

Figura 6.9: Largura de banda — Configuração RMI

Outra observação importante, corresponde ao declive do gráfico para os cenários *WEB*, nos períodos antes do "estrangulamento" do canal. Nesses períodos, o processamento efectuado pelo canal é quase irrelevante relativamente ao período de inactividade (o que corresponde à espera entre invocações dos serviços *Web*). Este facto, é comprovável pelo facto de o declive para o teste WEB_1 ser aproximadamente 1, 1/2 para o WEB_2, 1/5 para o WEB_5 e, finalmente, aproximadamente 1/10 para o teste WEB_10, ou seja, como se o tempo total relativo à transmissão das mensagens, fosse sempre idêntico apenas ao período entre invocações *Web*.

A observação do gráfico relativo às medições para a configuração RMI (figura 6.9), mostra que o tempo de transporte é muito menos relevante do que nos cenários anteriores e, consequentemente, o tempo de processamento gasto nas restantes actividades do canal (nomeadamente o *parsing* do XML), rapidamente se torna fulcral no desempenho global do canal. Apesar de não ser relevante para a tese que se pretende demonstrar, esta é uma medida importante, do peso das diferentes funções do canal, no desempenho global do sistema.

Em ambos os casos, é válida a afirmação de que, para um melhor rendimento, o funcionamento do canal deve situar-se na zona de crescimento linear do gráfico de largura de banda. Por exemplo, para o cenário *WEB_1*, o comportamento do canal é óptimo para tráfegos (úteis) na casa dos 50kbytes/s (ou menos) e com invocações *Web* de eventos com um tamanho máximo de 50kbytes (para que o *parser* se comporte como esperado).

Com base nestas medidas, o canal pode escolher o regime mais eficaz nas diferentes situações de rede e de tráfego necessário. Analisando um exemplo para verificar a importância desta escolha, imaginemos que é necessário ao canal, debitar eventos a

uma taxa útil de 50kbytes/s, com um regime de invocações *Web*, a cada 5 segundos; facilmente se conclui que, em média, estarão 250kb de eventos acumulados para transmitir, no instante de cada invocação. Como pudemos ver, quer nos testes de latência, quer nos testes de largura de banda, o *parsing* de uma *stream* XML com 250kb não é, de todo, eficiente (na implementação usada), o que indicia que o canal iria funcionar num regime "estrangulado" e provavelmente atrasar-se-ia progressivamente no despacho dos eventos (pode ver-se na figura 6.7, *WEB_5*, que o estrangulamento começa a ocorrer um pouco antes dos 200kb). No entanto, nessas mesmas condições mas com um período entre invocações de 1 segundo, o comportamento do canal é bom, como facilmente se conclui do gráfico para esse regime[3].

6.4 Conclusões

Neste capítulo definiu-se o ambiente de simulação para a obtenção de resultados. Efectuaram-se os testes previstos e capturaram-se os resultados correspondentes. Fez-se também uma análise desses resultados à luz dos requisitos impostos para o canal.

No próximo capítulo, tiram-se conclusões e apresentam-se possibilidades de continuação futura deste trabalho.

[3]Ver sugestões para trabalhos futuros, secção 7.2.

Capítulo 7

Conclusões

Com base nos resultados obtidos e resumidos no capítulo anterior, faz-se uma análise aos objectivos iniciais e aos requisitos típicos do mundo SCADA, concluindo do interesse, ou não, deste canal para o efeito previsto.

Apresentam-se também algumas propostas de melhoria e extensão do trabalho efectuado no âmbito desta tese.

7.1 Satisfação dos Objectivos

Recuperando o enunciado da tese avançado na secção 3.3.1:

> "É possível implementar um canal de comunicações que permita ter aplicações em pontos diferentes da Internet (por exemplo as interfaces gráficas separadas geograficamente do centro de comando com uma *firewall* no meio), mas cumprindo os requisitos descritos para aplicações SCADA."

Estamos agora em condições de avaliar se foi ou não cumprido esse objectivo e provada a tese.

Antes de se fazer a avaliação objectiva, com base nos resultados obtidos, revê-se o percurso efectuado e descrito nos capítulos até aqui.

7.1.1 Opções Tomadas

As grandes questões com que esta tese se debateu foram as seguintes:

Qual a vantagem de ter aplicações SCADA na *Web*? Esta foi a questão inicial e está na génese da própria tese. Procurámos responder a esta questão

no capítulo 1 abordando as dificuldades com que os fabricantes de *Software* SCADA se debatem. Dificuldades essas, que seriam de alguma forma minoradas pelo uso de tecnologias *standard*, sobretudo protocolos, preferencialmente em texto (XML) por permitirem uma muito mais imediata interoperabilidade com módulos de outros fabricantes.

Avançámos mais tarde outras vantagens, com a abordagem que usámos de construção de um *framework* comum, permitindo a implementação de uma só versão das aplicações, independentemente de elas virem a correr na *Web* ou na rede local, o que é uma clara vantagem tecnológica.

Porquê ter clientes pesados? Como vimos, uma das formas mais comuns de abordar a computação distribuída na Internet, é tendo clientes "leves" implementados com tecnologias orientadas apenas para a interacção, estilo formulário, com refrescamento induzido pelo próprio operador.

Na definição dos requisitos típicos para aplicações SCADA (secção 2.2), encarámos a riqueza gráfica tradicional nas aplicações de interface com o operador do tipo SCADA, como um requisito importante. Nomeadamente, com possibilidade de interagir com as aplicações como acontece nas aplicações tradicionais.

Neste contexto, tornou-se necessário assumir que existiriam aplicações computacionalmente "competentes" nos dois extremos do canal.

Porquê a opção por JAVA, SOAP e XML? As opções tecnológicas são explicadas detalhadamente no capitulo 3, sendo que os motivos essenciais estão relacionados com os requisitos de portabilidade e interoperabilidade impostos anteriormente.

Eram necessárias tecnologias *Web* interoperáveis, o que só acontece verdadeiramente usando XML na codificação dos eventos a transmitir e um transporte igualmente *standard* na *Internet* (o que está em vias de acontecer com o SOAP).

Por outro lado, dentro dos panoramas de ambiente de desenvolvimento à disposição, com programação de alto nível (orientada a objectos) e fácil manipulação das tecnologias *Web* escolhidas, a opção reduziu-se às tecnologias ".NET" e "J2EE". O requisito de portabilidade condicionou esta escolha, uma vez que só o JAVA garante, nesta altura, uma portabilidade efectiva entre a generalidade dos sistemas operativos existentes.

Para quê a construção de um framework? Nos capítulos 3 e 4 procurou justificar-se o uso de receitas (*patterns*) de computação de um modo geral, como sendo a solução mais eficiente na resolução de problemas sistematizáveis.

Demonstrámos que o *framework* é o paradigma dessa mesma abordagem em que se aposta na reutilização e encapsulamento de diversos problemas, que afectam todas as aplicações. Esta opção permitiu que se construísse um ambiente de simulação e testes de forma simples e que serviu para a recolha de resultados, quer na implementação RMI do canal quer na implementação *Web*, sem esforço adicional.

Porquê o modelo de publicação e subscrição? Esta não era, claramente, a única opção, mas é uma opção cada vez mais utilizada, por oposição às tradicionais ligações cliente–servidor, sendo que as vantagens no tempo de desenvolvimento das aplicações, na clarificação das responsabilidades, mas sobretudo no desacoplamento final de cada componente relativamente aos outros, justificam claramente a opção.

Procurámos demonstrar estas vantagens no capítulo 4, exemplificando com soluções deste tipo por parte de importantes fabricantes mundiais. Seguimos a abordagem seguida pela EFACEC - Sistemas de Electrónica, SA, justificando as diferenças conceptuais para outras abordagens.

Para que serviu a implementação em RMI do canal? Interessava ter um termo de comparação, para validar a implementação *Web* do canal. Podia ter-se optado por validar essa implementação apenas contra os requisitos estabelecidos no capítulo 2, mas por um lado a análise desses resultados não seria tão interessante na discussão da abordagem *"middleware"* tradicional, comparativamente com a abordagem *Web*. Por outro lado, a implementação RMI, sendo mais simples tecnologicamente de implementar, permitiu validar o *framework* construído do ponto de vista da lógica de publicação e subscrição, por forma a que a implementação *Web* se concentrasse apenas no problema de transporte de eventos.

Porquê a opção pelos "pedidos sucessivos" por parte dos clientes? Esta questão foi discutida no capítulo 5, sendo que, mais uma vez, se tratou da opção por uma receita "prescrita" para problemas como o que tínhamos que resolver. A abordagem veio a revelar ter resultados interessantes embora, naturalmente, condicionados pelo período entre pedidos sucessivos.

7.1.2 Resultados Satisfatórios

Como pudemos ver, vários dos requisitos impostos pelo enunciado da tese e pelo contexto das próprias aplicações SCADA, foram cumpridos pelas opções tomadas em termos de arquitectura e tecnologias a usar.

Ainda assim, só com base nos resultados capturados e analisados no capítulo 6, é que foi possível validar totalmente a tese. Pudemos observar na análise efectuada, que dependendo essencialmente da escolha do período entre actualizações invocadas pelo cliente e do tamanho máximo de eventos a transportar, se conseguiam resultados satisfatórios para os requisitos relativos às aplicações SCADA.

Verificou-se também que a comparação com os resultados, para os mesmos cenários de teste com o canal RMI, não revelou diferenças de desempenho demasiado grandes com excepção, evidentemente, da latência para tráfegos residuais.

Com excepção do requisito de "adaptabilidade" às condições instantâneas por parte do canal, conclui-se que a tese foi validada, ficando não só provada a possibilidade de

colocação de aplicações SCADA na *Web*, como adiantada uma implementação suporte para cumprir esse objectivo. Relativamente ao requisito desejável de adaptabilidade às condições instantâneas da rede, muniu-se o *framework* de um mecanismo de medição da largura de banda e da latência e estudou-se a importância da escolha de um bom regime para diferentes situações.

7.2 Trabalhos Futuros

Sugerem-se agora, alguns aspectos que poderão ser melhorados no trabalho efectuado e novas funcionalidades para extensão do *framework* construído.

7.2.1 Oportunidades de Melhoria

- Como já referimos, o problema da adaptabilidade do canal, não foi totalmente tratado. Inseriu-se no *framework* um mecanismo de leitura das condições da rede, entre dois pontos e em qualquer instante, mas faltou implementar duas funcionalidades essenciais para a utilização industrial do canal na *Web*:
  - com base em medidas periódicas (com os mecanismos que já existem) o canal deve ser capaz de se adaptar, mantendo um regime de funcionamento óptimo tal como foi explicado na análise dos resultados (secção 6.3.2);
  - o canal deve "informar" os componentes que o usam, da qualidade do mesmo em cada instante, por exemplo, fornecendo uma estimativa do tempo de entrega dos eventos ou da largura de banda;
- Seria também interessante testar o comportamento do canal, na *Internet*, para diferentes tipos de acesso no que diz respeito à largura de banda e fiabilidade (Modem, RDIS, ADSL). Avaliar, para esses regimes, se as conclusões retiradas nesta dissertação se mantêm válidas.
- Finalmente, poderia avaliar-se o efeito da introdução da encriptação no desempenho global do canal. Quer por SSL, quer pela aplicação de XML Encription.

7.2.2 Novas Funcionalidades

- Implementação de mais um canal de comunicação (a incluir no *framework*) IIOP, para integração directa com componentes CORBA desenvolvidos na filosofia de publicação e subscrição.
- Implementação de um canal nativo (JNI), baseado em sockets para permitir a conexão a qualquer tipo de processo apenas por TCP/IP. Esta implementação

implica, naturalmente, que se faça uma tradução da lógica de publicação e subscrição, para uma interface nativa de comunicações simples.

- Integração do *framework* com aplicações SCADA HMI, feitas em JAVA e usando este framework.
- Introdução de algoritmos de compactação em tempo-real na transmissão do canal. Avaliação dos ganhos.

7.3 Conclusões

Terminado o trabalho, é notório que o percurso que este acabou por levar, se tenha pautado mais pela discussão de questões relacionadas com a computação distribuída, nomeadamente na sua adaptação à *Web*, do que pelo contexto SCADA propriamente dito.

O contexto SCADA serviu, ainda assim, para dar uma dimensão concreta a este trabalho, impondo referências, métricas, requisitos, sem os quais a discussão seria eventualmente demasiado abstracta e talvez menos interessante.

Por outro lado, o contexto *Web*, está a sofrer uma evolução muito importante, sobretudo com o crescimento exponencial dos serviços *Web*. Neste contexto evolutivo, o W3C tem vários temas em plena fase de discussão, desde protocolos, especificações de segurança, até à arquitectura da própria *Web*. Paralelamente, a indústria do *Software*, acompanha essas evoluções, chegando por vezes a antecipá-las.

Neste contexto frenético, qualquer dissertação nesta área, corre o risco de ser ultrapassada, ou perder a oportunidade, ainda antes de ter terminado. Julgo, sinceramente, que tal não aconteceu com este trabalho, ainda me parece oportuno e inovador na abordagem diferente que faz da computação distribuída na *Web*.

As opções tomadas são, naturalmente, discutíveis. Existirá quem defenda protocolos binários por oposição ao XML, arquitecturas cliente–servidor por oposição ao *publish/subscribe*, ou ainda, clientes leves na *Web* por oposição a aplicações comuns numa arquitectura de N camadas. Ainda assim, julgo que este trabalho traz algum contributo por "desbravar" caminho nesta área e demonstrar a viabilidade de uma alternativa para a computação distribuída na *Web*.

Apêndice A

Acrónimos

ADSL	Asymmetric Digital Subscriber Line
ASCII	American Standard Code For Information Interchange
CCIC	Committee on Computing, Information, and Communications
CERJ	Companhia de Eletricidade do Rio de Janeiro
CERN	Conseil Européen pour la Recherche Nucléaire
CORBA	Common Object Request Broker Architecture
DAML	DARPA Agent Markup Language
DCS	Distributed Control Systems
DMS	Distribution Management System
DOM	Document Object Model
DTD	Document Type Definition
EJB	Enterprise JavaBeans TM
EMS	Energy Management System
FNC	Federal Networking Council
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
IEC	International Engineering Consortium
IP	Internet Protocol
J2EE	Java TM 2 Platform, Enterprise Edition
JDK	Java TM Development Kit
JMS	Java TM Messaging Service
JRE	Java TM Runtime Environment
LSN	Large Scale Networking

OSI	Open Systems Interconnection
PKI	Public Key Information
RDF	Resource Description Framework
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAX	Simple API for XML
SCADA	Supervisory Control and Data Acquisition
SGML	Standard Generalized Markup Language
SOAP	Simple Object Access Protocol
SSL	Secure Socket Layer
TCP	Transmission Control Protocol (RFC 793)
UDDI	Universal Description Discovery and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
W3C	World Wide Web Consortium
WDDX	Web Distributed Data eXchange
WSDL	Web Services Description Language
WWW	World Wide Web
XML	Extended Markup Language
XML-RPC	XML Remote Procedure Call
XSL	Extensible Stylesheet Language
XSL-FO	XSL Formatting Objects
XSLT	Extensible Stylesheet Language Transformations

Referências

[ABC+01] Sharon Adler, Anders Berglund, Jeff Caruso, Stephen Deach, Tony Graham, Paul Grosso, Eduardo Gutentag, Alex Milowski, Scott Parnell, Jeremy Richman, and Steve Zilles. Extensible Stylesheet Language (XSL), Version 1.0. Disponível na Internet em http://www.w3.org/TR/xsl/, October 2001.

[BL89] Tim Berners-Lee. Information Management: A Proposal. Disponível em http://www.w3.org/History/1989/proposal.html, Março 1989. Revisto em Maio de 1990.

[BLBC+02a] Tim Berners-Lee, Tim Bray, Dan Connoly, Paul Cotton, Row Fielding, Chris Lilley, David Orchard, Norman Walsh, and Stuart Williams. Architecture of the world wide web. Documento disponível na Internet em http://www.w3.org/TR/2002/WD-webarch-20021115/, Novembro 2002.

[BLBC+02b] Tim Berners-Lee, Tim Bray, Dan Connoly, Paul Cotton, Row Fielding, Chris Lilley, David Orchard, Norman Walsh, and Stuart Williams. Architecture of the World Wide Web. Documento disponível na Internet em http://lists.w3.org/Archives/Public/xml-dist-app/2000Feb/0006.html, Novembro 2002.

[BLFF96] Tim Berners-Lee, F. Fielding, and H. Frystyq. Hiper Text Transfer Protocol – HTTP/1.0, May 1996.

[BLFM98] Tim Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax, Agosto 1998.

[Boy99] Stuart A. Boyle. *SCADA: Supervisory Control and Data Acquisition.* ISBN: 1556176600. ISA - The Instrumentation, Systems, and Automation Society, 2nd edition, January 1999.

[Boy01] J. Boyer. RFC3076 Canonical XML Version 1.0. Request For Comments, published by the Internet Society, March 2001.

[CB94] William Cheswick and Steven Bellovin. *Firewalls and Internet Security, Repelling the Wily Hacker.* Addison-Wesley Professional, 1994.

[CDD01] Peter Chen, R. Daniel, and S. et al DeRose. XML Linking Language (XLink) Version 1.0. Disponível na Internet em http://www.w3.org/TR/2001/REC-xlink-20010627/, Junho 2001.

[CFNO02] M. Champion, C. Ferris, E. Newcomer, and D. Orchard. Web Services Architecture. Documento disponível na Internet em http://www.w3.org/TR/ws-arch/, Novembro 2002.

[COM02] Internet Access, IEC tutorial. IEC tutorial — Internet Engineering Consortium, 2002. Disponível na Internet em http://www.iec.org/online/tutorials/int/nb_acc/.

[D1000] Regulamento da Rede de Transporte. Despacho 10315 — Diário da República n. 116, Maio 2000.

[D1200] Regulamento da Qualidade de Serviço. Despacho 12917 — Diário da República n. 143, Junho 2000.

[D1399] Regulamento da Rede de Transporte. Despacho 13615 — Diário da República n. 164, Junho 1999.

[DAM] DAML — DARPA Agent Markup Language. Disponível na Internet em http://daml.semanticweb.org/.

[DS99] A. Daneels and W. Salter. What is SCADA? In *ICALEPCS'99*, pages 339–343, Outubro 1999. Trieste — Itália.

[Eck00] Bruce Eckel. *Thinking in Java.* Prentice Hall, 2nd edition, 2000.

[Fer02] David Ferguson. Comparing High Performance Web Servers. Disponível em: http://support.zeus.com/doc/tech/zws_comparison.pdf, Fevereiro 2002.

[FG+99] R. Fielding, , J. Gettys, J. Mogul, H. Frystyq, L. Masinter, P. Leach, and Tim Berners-Lee. Hiper Text Transfer Protocol – HTTP/1.1, June 1999.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL Protocol, Version 3.0. Disponível na Internet em http://wp.netscape.com/eng/ssl3/, Abril 1996.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software.* Addisson Wesley Longman, Inc, 1995.

[GSB+01] Steve Graham, S. Simeonov, T. Boubez, Glen Daniels, Doug Davis, Y. Nakamura, and R. Neyama. *Building Web Services with JAVA: Making Sense of XML, SOAP, WSDL and UDDI.* SAMS, October 2001.

[HM01] Elliotte R. Harold and W. Scott Means. *XML in a Nutshell.* O'Reilly & Associates, Inc., January 2001.

[IDS02] Takeshi Imamura, Blair Dillaway, and Ed Simon. Namespaces in XML. Disponível na Internet em http://www.w3.org/TR/xmlenc-core/, Dezembro 2002.

[IET] IETF. Overview of the IETF. Acessível em http://www.ietf.org/overview.html, em Agosto de 2002.

[JMS02] *JAVA Message Service API Tutorial.* Sun Microsystems, Inc., 2002.

[KM97] D. Kristol and L. Montulli. HTTP State Management Mechanism. Disponível na Internet em http://www.ietf.org/rfc/rfc2109.txt, Fevereiro 1997.

[Lei00] Barry M. Leiner. A Brief History of the Internet, version 3.31. Documento disponível na internet (http://www.isoc.org/internet/history/brief.shtml) em 15/07/2002, Agosto 2000.

[LTE96] Disponibilizada pela EFACEC — Sistemas de Electrónica, SA., Dezembro 1996. Proposta de sistema SCADA/DMS para LTE.

[Mar02] Floyd Marinescu. *EJB Design Patterns.* John Wiley & Sons, Inc., 2002.

[MM97] Thomas J. Mowbray and Raphael C. Malveau. *CORBA Design Patterns.* John Wiley and Sons, Inc., 1997.

[Moo65] Gordon E. Moore. The experts look ahead — Cramming more components onto the integrated circuits. *Electronics*, 38(8), April 1965.

[MV01] Dave Marsh and Paulo Viegas. The BUS Architecture. In J. T. Saraiva and M. A. Matos, editors, *2001 IEEE Porto PowerTech*, number 01EX502C. INESC Porto - Power Engineering Unit, FEUP, PES Portuguese Chapter, IEEE, September 2001.

[Ohl02] Carl Ohlen. E-Commerce, On–line design. *Power Engineering International*, pages 25–27, Abril 2002. ABB Power Technology, Sweden.

[Pem00] S. et al. Pemberton. XHTML 1.0 The Extensible HiperText Markup Language (Second Edition). Disponível na Internet em http://www.w3.org/TR/2002/REC-xhtml1-20020801/, Janeiro 2000. revisto em Agosto de 2002.

[Pos82] Jonathan B. Postel. SIMPLE MAIL TRANSFER PROTOCOL, August 1982. by Information Sciences Institute, Un. of Southern California.

[PR85] J. Postel and J. Reynolds. FILE TRANSFER PROTOCOL (FTP), October 1985.

[RDF] RDF — Resource Description Framework. Disponível na Internet em http://www.w3c.org/rdf/.

[RFC81] INTERNET PROTOCOL DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION, September 1981. by Information Sciences Institute, Un. of Southern California.

[RR02] J. Rosa and A. Rodrigues. Centralizing Control Centers. *Transmission & Distribution World*, pages 52–57, Agosto 2002.

[SCD+01] A. Subrahmnayam, Beust C., J. Davies, Tyler Jewell, Rod Johnson, A. Longshaw, R. Nagappan, D. O'Connor, Dr. P. G. Sarang, A. Toussaint, S. Tyagi, G. Watson, M. Wilcox, and Alan Williamson. *Professional Java Server Programming, J2EE 1.3 Edition.* Wrox Press, Ltd., September 2001.

[Sch96] Bruce Schneier. *Applied Criptography.* John Wiley & Sons, Inc., 2nd edition, 1996.

[Sil02] J. Silva. Qualidade de Serviço em Sistemas de Distribuição de Energia Eléctrica. Acção de formação realizada no INESC Porto, Novembro 2002.

[SM02] Stephen Stelting and Olav Maassen. *Applied Java Patterns.* Sun Microsystems, 2002.

[SR98] Brian Selic and Jim Rumbaugh. Using UML for Modeling Complex Real-Time Systems. WhitePaper da Rational Software - http://www.rational.com/media/whitepapers/umlrt.pdf, Março 1998.

[Ste90] W. Richard Stevens. *UNIX Network Programming.* Software Series. Prentice Hall, 1990.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated: the protocols*, volume 1. Addison Wesley Longman, Inc., 17th edition, April 1994.

[STE02] Disponibilizada pela EFACEC — Sistemas de Electrónica, SA., Dezembro 2002. Proposta de sistema SCADA/DMS para a STEG — Tunísia.

[STK02] James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web Services with SOAP.* O'Reilly & Associates, Inc., January 2002.

[SV02] Douglas Schmidt and Steve Vinoski. Object Interconnections: CORBA and XML — Part 3: SOAP and Web Services. *C/C++ Users Journal*, 2002.

[UNI02] Compuware UNIFACE Products Home. 4GL UNIFACE da Compuware — portal descritivo, 2002. http://www.compuware.com/products/uniface/.

[Vin01] Steve Vinoski. The Chief Architect's View: WebServices. *IONASphere*, Maio 2001. disponível em http://www.iona.com/hyplan/vinoski/.

[W3C99] W3C. Namespaces in XML. Disponível na Internet em http://www.w3.org/TR/1999/REC-xml-names-19990114/, Janeiro 1999.

[W3C02a] W3C. SOAP Version 1.2 Part 1: Messaging Framework. Documento disponível na Internet em http://www.w3.org/TR/soap12-part1/, Junho 2002.

[W3C02b] W3C. SOAP Version 1.2 Part 2: Adjuncts. Documento disponível na Internet em http://www.w3.org/TR/soap12-part2/, Junho 2002.

[WDD] WDDX — Web Distributed data Exchange. Disponível na Internet em http://www.openwddx.org/.