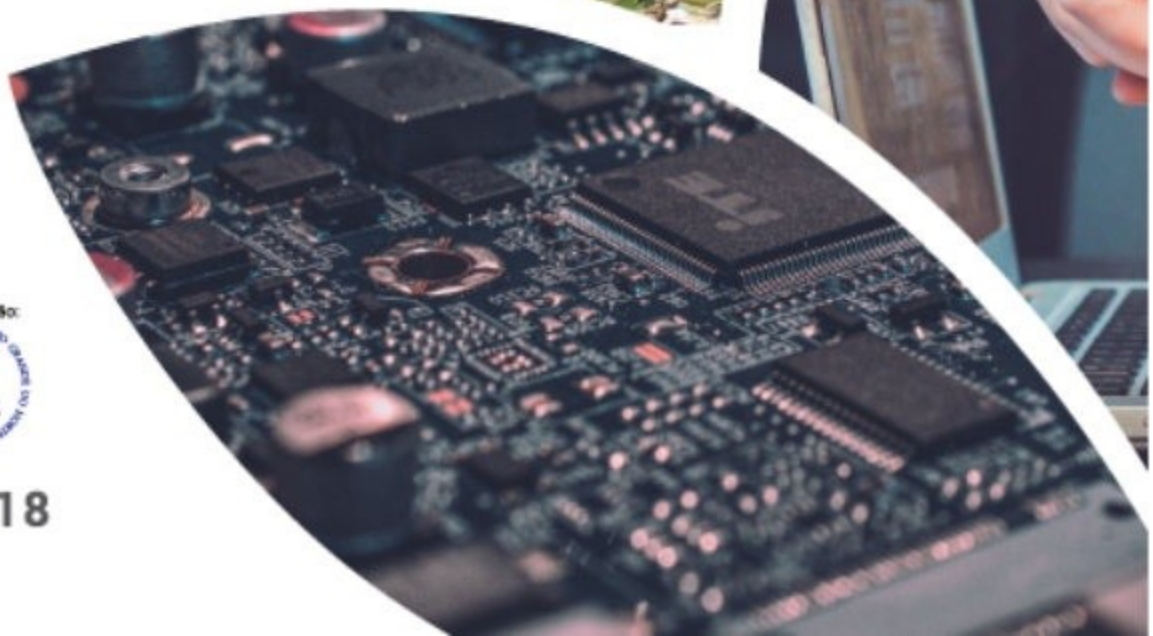
JAI 2018

Jornadas de Atualização em Informática



Realização: SBC

Organização:

NATAL, 2018

JAI 2018

Jornadas de Atualização em Informática

Organização

Eduardo Santana de Almeida

Paulo Gabriel Gadelha Queiroz

Sociedade Brasileira de Computação – SBC

Porto Alegre

2018

Dados Internacionais de Catalogação na Publicação (CIP)

J82 Jornada de Atualização em Informática (37. : 2018 : Natal, RN)
Anais da 37º Jornada de Atualização em Informática [recurso eletrônico] / Organizadores: Eduardo Almeida, Paulo Gabriel Gadelha Queiroz – Porto Alegre : SBC, 2020.

ISBN 978-65-87003-08-5

1. Computação - Congresso. 2. Computação e sustentabilidade. I. Sociedade Brasileira de Computação. II. Universidade do Estado do Rio Grande do Norte.

CDU 004

Catalogação elaborada por Francine Conde Cabral
CRB-10/2606

Coordenação Geral

Eduardo Santana de Almeida (UFBA)

Possui graduação em Ciência da Computação pela Universidade Salvador (2000), mestrado em Ciência da Computação pela Universidade Federal de São Carlos (2003), doutorado pela Universidade Federal de Pernambuco (2007) com período sanduiche na University of Mannheim (2006) e Pós Doutorado no Virginia Tech (2008). Atualmente é Professor Associado da Universidade Federal da Bahia (UFBA). Durante o período de 2012-2014 foi Coordenador do Mestrado em Ciência da Computação (MCC) UFBA-UEFS. Em 2014 foi também Chefe do Departamento de Ciência da Computação (DCC). Atuou como Líder de Pesquisa em Engenharia de Software no Fraunhofer Project Center (FPC/UFBA) de 2012-2015 onde auxiliou na sua concepção inicial. Foi membro e Vice-Coordenador da Câmara de Assessoramento e Avaliação da Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB) (2011-2014) nas áreas de Computação e Engenharias e Presidente no período de 2015-2017. Foi Presidente da Comissão Especial de Engenharia de Software (CEES) da SBC (2016-2017). Na universidade, tem participado e coordenado diversos projetos com financiamentos da FAPESB, FACEPE, FINEP, CNPq, CAPES e iniciativa privada. No cenário internacional, coordenou projetos de cooperação formal com a Suécia (Malardalen University) e Espanha (Universitat Politècnica de València) com financiamentos do Governo Sueco e CAPES (DGU) e coordena o projeto de cooperação com a Bélgica (University of Namur) com financiamento da CAPES (WBI). Tem experiência na área de Ciência da Computação, com ênfase em Engenharia de Software, atuando principalmente nos seguintes temas: métodos, processos, ferramentas e métricas para o desenvolvimento de software reutilizável. Nesta área, é autor de três livros e mais de 250 artigos publicados nos principais congressos e periódicos. Formou mais de quarenta alunos de mestrado, doutorado e pós doutorado. É membro da Sociedade Brasileira de Computação (SBC), Association for Computing Machinery (ACM) Senior Member, IEEE Senior Member, membro do comitê gestor do Instituto Nacional de Ciência e Tecnologia (INCT) para Engenharia de Software (INES), IEEE Computer Society Certified Software Development Professional (CSDP), Membro Afiliado da Academia Brasileira de Ciências (ABC) e Membro Titular da Academia de Ciências da Bahia (ACB). Na sua comunidade de pesquisa tem participado ativamente como General Chair (SPLC, SBCARS, SBQS, VaMoS, ICSR), Program Chair (SPLC, ACM SIGSOFT CBSE, ICSR, SBCARS, SBQS, SBES), Workshop Chair (WICSA/ECSA, ICSR), Publicity Chair (ICSR, ICGSE), Steering Committee (SPLC, ICSR, VaMoS, SBCARS, CBSOFT, SBES) e PC member. Tem também atuado como Guest Editor em relevantes periódicos (IST, JSS, JBCS, JUCS) e é membro do Corpo Editorial do Journal of Systems and Software (JSS) e Journal of the Brazilian Computer Society (JBCS).

Coordenação Local

Paulo Gabriel Gadelha Queiroz (UFERSA)

Possui graduação em Computação pela Universidade Federal do Ceará (2007), mestrado (2009) e doutorado (2015) pela Universidade de São Paulo (ICMC-USP). Atualmente, é professor Adjunto II do curso de graduação em Ciência da Computação da Universidade Federal Rural do Semi-Árido (UFERSA). Tem experiência na área de Ciência da Computação, com ênfase em Engenharia de Software. Os maiores interesses de pesquisa são: reúso, linha de

produtos, sistemas Web, Web Services, geradores de aplicações e sistemas embarcados críticos. (Texto informado pelo autor)

SUMÁRIO

Capítulo 1

Análise Dinâmica de Programas Binários

Hugo Sousa e Mateus Tymburibá

Abstract

Dynamic analyses are techniques used to observe the behavior of programs during their execution. Tools that implement such techniques include not only the well known profilers, such as gprof and Shark, but also dynamic binary instrumentation frameworks, the focus of this course. Dynamic Binary Instrumentation (DBI) adds code to a program's execution flow in order to study the events that occur during its execution. Analysis code is executed as if it were part of the program's normal execution flow, without affecting it, and performing extra tasks - like measuring performance or identifying errors - in a transparent manner. Since they work with the actual input data of programs, dynamic analyses are capable of assessing, in an exact way, conditions that cannot be defined statically. Due to this advantage and the maturity achieved by DBI tools, this technique has been widely used in multiple scenarios, notably the systems security area. This mini course presents the fundamentals about DBI and discusses real examples in security of applications. Hence, it enables professionals and researchers to benefit from the many possibilities offered by DBI, besides allowing the attendants to get acquainted with some concepts related to software security.

Resumo

Análises dinâmicas são técnicas utilizadas para observar o comportamento de programas durante sua execução. Ferramentas que implementam tais técnicas incluem não somente os bem conhecidos perfiladores (profilers), como gprof e Shark, mas também instrumentadores binários, foco deste curso. A Instrumentação Dinâmica de Binários (IDB) adiciona código ao fluxo de execução de um programa a fim de estudar eventos que ocorrem durante sua execução. O código de análise é executado como se fizesse parte da execução normal do programa, sem perturbá-la, e fazendo seu trabalho extra - como medir o desempenho ou identificar erros - de forma transparente. Por trabalharem com valores de entrada reais, análises dinâmicas são capazes de avaliar de forma exata condições que não podem ser definidas estaticamente. Em função dessa vantagem e da maturidade

alcançada pelas ferramentas de IDB, ela tem sido amplamente utilizada em diversos cenários, com destaque para a área de segurança de sistemas. Este minicurso apresenta os fundamentos sobre IDB e discute exemplos reais voltados à segurança de aplicações. Dessa forma, habilita profissionais e pesquisadores a usufruírem das diversas possibilidades oferecidas pela IDB, além de familiarizar seus participantes com alguns conceitos relacionados à segurança de software.

1.1. Introdução

À medida que os sistemas computacionais se tornam mais complexos, com novos paradigmas e modelos de programação, com diferentes tipos de hardware executando concomitantemente em um mesmo computador e com aplicações cada vez maiores, torna-se também mais difícil a compreensão de todas as nuances de uma aplicação. Muitas vezes, para obter-se dados mais precisos, é necessário analisar como um software se comporta quando executado, ao contrário de simplesmente analisar seu código fonte [Reddi et al. 2004].

Foi pensando em atender a essas necessidades que o processo de instrumentação binária foi proposto. Ele consiste na adição de código ao arquivo executável de uma aplicação, de modo a observar ou alterar o comportamento do programa [Laurenzano et al. 2010]. Esse código adicional permite, então, que o programador obtenha informações importantes sobre a aplicação executada, que não estariam disponíveis de outra maneira. Dessa forma, o processo de instrumentação binária é extremamente útil para a análise do comportamento de uma aplicação, a avaliação de desempenho e a detecção de defeitos (*bugs*). Contudo, os usos da técnica de instrumentação binária há muito não se limitam a essas possibilidades. A instrumentação binária já foi utilizada na academia com finalidades diversas, como no auxílio ao projeto de sistemas, na verificação da corretude de programas, na otimização de softwares e na verificação da segurança de sistemas, entre outros [Uh et al. 2006]. Neste trabalho, estudaremos uma modalidade dessa técnica, conhecida como instrumentação dinâmica de binários, que aplica a instrumentação durante a execução de uma aplicação.

1.1.1. Contextualização

Como já mencionado, a maior motivação para a criação da técnica de instrumentação binária foi o processo de compreensão de software [Cornelissen et al. 2009]. Esse processo envolve toda a compreensão do funcionamento de um sistema e era inicialmente feito de forma manual pelos programadores. Como tal, consistia basicamente no estudo do código fonte de um programa e sua documentação (quando existente). Esses fatores faziam com que o procedimento se tornasse demorado e insuficiente para cobrir todos os aspectos de um programa.

O uso de análise dinâmica de software mostrou-se eficaz na abordagem desses problemas. Com esse tipo de ferramenta, dados coletados durante a execução de um programa podem ser analisados. Essa técnica traz maior precisão à compreensão de software, uma vez que permite observar o comportamento de um programa quando executado sobre dados reais. Entretanto, por mais que a análise dinâmica de um programa represente um avanço em relação à compreensão de software essencialmente manual, ela também

apresenta limitações. Dentre elas, talvez a mais importante seja o fato de que uma única execução de um programa provavelmente não exercerá todos os seus possíveis fluxos de execução. Na maioria dos programas escritos hoje em dia, o conjunto de dados utilizados como entrada da aplicação afetam fortemente seu fluxo de execução. Hoje em dia, porém, já existem técnicas para lidar com essas limitações, como o uso de heurísticas e abstrações para agrupar execuções de um programa que compartilham propriedades [Cornelissen et al. 2009].

Desde que a análise dinâmica começou a ser utilizada no contexto de compreensão de software, os métodos de funcionamento e os focos de análise das ferramentas que a utilizam se tornaram os mais variados. Até meados dos anos 2000, o principal foco dessas ferramentas foi a visualização de propriedades estruturais dos programas analisados, como, por exemplo, fluxos de execução e estruturas de dados. Após os anos 2000, além das já citadas ferramentas de visualização, nota-se também que as ferramentas de análise dinâmica passam a ter maior foco em planejamento de sistemas e aspectos comportamentais, como a interação entre *threads* em sistemas distribuídos.

Atualmente, existe uma grande variedade de ferramentas de análise dinâmica disponíveis. Dessa forma, o programador pode ficar em dúvida em relação a qual delas escolher para seus propósitos. Por exemplo, as ferramentas de visualização se mostram mais úteis para o programador que visa estudar uma aplicação e entender suas peculiaridades em um nível mais alto, sem maiores preocupações com os efeitos da execução do programa em termos de arquitetura.

Neste trabalho, o foco de nosso estudo será a análise dinâmica de uma aplicação através da técnica de Instrumentação Dinâmica de Binários (IDB). Essa escolha, em detrimento dos outros tipos de ferramentas de análise dinâmica, se deu principalmente pelo nível de detalhe alcançável com o uso da IDB. Além disso, ela permite ao programador definir quais são os dados específicos de seu interesse, utilizando uma estratégia orientada a objetivos.

1.1.2. Instrumentação Dinâmica de Binários

Nesta seção, a técnica de Instrumentação Dinâmica de Binários será analisada em detalhes. Em um primeiro momento (Seção 1.1.2.1), o conceito geral de instrumentação de binários será apresentado, juntamente com algumas de suas aplicações. Em seguida (Seção 1.1.2.2), será mostrada uma comparação entre instrumentação estática e dinâmica de binários, apresentando-se as vantagens e desvantagens de cada uma. Finalmente (Seção 1.1.2.3), uma perspectiva sobre a grande quantidade de ferramentas de instrumentação dinâmica de binários existentes no mercado será traçada. As ferramentas de maior popularidade atualmente serão destacadas e comparadas com a ferramenta escolhida como foco de estudo deste trabalho.

1.1.2.1. Definição e Aplicações

Como mencionado anteriormente, a técnica de instrumentação binária consiste em adicionar porções de código ao executável de uma aplicação. Primordialmente, quando pensamos nessa ideia, duas perguntas básicas devem ser respondidas. A primeira se refere ao

local do executável onde o código de instrumentação deve ser inserido. A segunda diz respeito a qual código deve ser enxertado. A primeira pergunta define a granularidade da instrumentação, isto é, o nível de abstração do programa ao qual o código de instrumentação será aplicado. Por exemplo, o programador pode estar interessado em analisar os valores dos registradores do processador à medida que as instruções da aplicação são executadas. Nesse caso, a instrumentação será feita com granularidade de instrução. Por outro lado, outro programador pode estar interessado no efeito de uma função do programa analisado na memória da aplicação. Para obter esses dados, ele não precisa adicionar código de instrumentação a cada instrução do programa. Para isso, basta que ele adicione código de instrumentação ao começo e/ou fim de uma função da aplicação. Em relação à segunda pergunta, o programador deve estar atento ao tipo de informação que deseja obter com a instrumentação. Além disso, é importante notar que o nível de detalhes dos dados obtidos com a instrumentação será limitado pela ferramenta utilizada.

Diante dos diversos usos e ferramentas disponíveis para instrumentação binária atualmente, é interessante observar como essas ferramentas são utilizadas. Uma das aplicações mais comuns da técnica de instrumentação binária é seu uso como ferramenta de detecção de erros. Isso se deve à possibilidade de inserir código que coleta informações sobre o estado (memória e registradores) da aplicação em posições estratégicas da execução do programa. Dessa forma, pode-se, por exemplo, monitorar todas as operações de memória realizadas por um programa. Isso inclui a verificação do conteúdo de bytes lidos ou escritos no espaço de endereçamento de um processo. Pode ser de interesse do programador, também, detectar acessos a regiões de memória não endereçáveis, obter informações sobre travas de memória para detectar situações de corrida em aplicações paralelas, ou até mesmo checar os tipos utilizados por um programa durante sua execução. De fato, é possível monitorar todas as operações realizadas com uma variável e impedir que operações inapropriadas (de acordo com a tipagem) sejam executadas. Na literatura, um exemplo do uso da instrumentação para fins de detecção de erros é a implementação do conceito de memória sombra, através do qual dados sobre todos os bytes usados por um programa são coletados [Nethercote and Seward 2007a].

A instrumentação binária também se mostra útil no auxílio ao projeto de sistemas. Métodos para a avaliação de protótipos, com foco no desempenho da memória, já foram utilizados na literatura com o apoio da técnica de instrumentação [Uhlig and Mudge 1997]. A abordagem de avaliação orientada à sequência de acessos à memória, por exemplo, simula o comportamento de um projeto de memória quando efetivamente sob uso. Nela, uma aplicação de interesse é escolhida e executada. Durante sua execução, ela realiza uma série de referências à memória. Essa série de referências é, então, coletada por uma ferramenta de instrumentação binária. Como essa sequência de referências é potencialmente muito grande, algum tipo de redução é realizada. Por fim, a sequência é passada para algum programa que simulará o comportamento do sistema de memória projetado quando submetido à sequência de referências à memória previamente coletada. Note que, nesse contexto, é importante utilizar uma ferramenta que consiga coletar dados em diversos cenários, levando em consideração a existência de aplicações *multithread*, de interferência do sistema operacional, de códigos compilados ou ligados dinamicamente e de gigantescas sequências de referências à memória.

Finalmente, destacamos o uso da técnica de instrumentação para a otimização

de programas. O sistema Etch [Romer et al. 1997], desenvolvido na Universidade de Washington, permite reescrever o arquivo binário executável de uma aplicação para alcançar três objetivos principais: entender como uma aplicação interage com a arquitetura de um computador; avaliar o desempenho de um programa em desenvolvimento; e otimizar o desempenho da aplicação analisada. Os dois primeiros objetivos já foram abordados anteriormente. A ferramenta alcança o terceiro objetivo da seguinte forma: ela executa a aplicação alvo e coleta informações de interesse, como a sequência de acessos à memória e o número de instruções de desvio condicional que falham. A partir dessas e de outras informações, a ferramenta é então capaz de reescrever o binário da aplicação para, por exemplo, melhorar a localidade de referência de memória ou reordenar as instruções para manter o pipeline do sistema cheio, entre outras possíveis otimizações.

1.1.2.2. Instrumentação Estática x Dinâmica

A técnica de instrumentação binária possui duas abordagens, estática e dinâmica, que se diferem pelo momento em que são aplicadas ao arquivo executável do programa alvo. A instrumentação estática de binários ocorre antes que o programa seja executado. Por outro lado, a instrumentação dinâmica de binários insere o código de instrumentação à medida que o programa é executado. [Nethercote 2004].

Existem vantagens e desvantagens no uso de ambas abordagens. Pensando em termos de custo computacional, a abordagem estática se mostra mais atraente, uma vez que a inserção de código de instrumentação é feita somente uma vez, antes da execução do programa, gerando assim um novo arquivo executável [Laurenzano et al. 2010]. Em contrapartida, na Instrumentação Dinâmica de Binários, é inserido um *overhead* adicional para que a ferramenta de instrumentação execute trocas de contexto com a aplicação instrumentada, além de efetuar a desmontagem de códigos e a geração de instruções, tudo durante a execução da aplicação instrumentada. Estudos apontam que, para efetuar uma tarefa simples de contagem do número de blocos de instruções executadas, os frameworks Pin, Strata, DynamoRIO e Valgrind apresentam *overheads* médios que variam entre 2,3 a 7,5 vezes [Luk et al. 2005].

Por outro lado, as soluções de instrumentação estática que exigem uma desmontagem precisa do código binário para que possam modificá-lo só funcionam caso informações de depuração, como a tabela de símbolos, estejam disponíveis. Entretanto, por questões de economia de espaço e proteção de propriedade intelectual, essas informações de depuração normalmente são removidas dos softwares destinados a ambientes de produção. Outra deficiência relacionada à abordagem estática recai sobre a possibilidade de corromper códigos assinados, blocos de instruções que executam verificações sobre si mesmos – como checagens de somas (*checksums*) –, ou códigos que se modificam durante a execução – como trechos ofuscados (*enconded*). Além disso, a inserção estática de instruções também é incapaz de atuar em códigos compilados sob demanda (*JIT-compiled*), comum em aplicações que dão suporte a ambientes com linguagens interpretadas, como Java, JavaScript, Flash, .Net e SilverLight.

A abordagem dinâmica provê muitas vantagens sobre a estática. Entre elas está a possibilidade de instrumentar bibliotecas compartilhadas sem qualquer trabalho extra.

Ou seja, com a instrumentação dinâmica, não é necessário instrumentar separadamente cada uma das bibliotecas compartilhadas que um programa usa. Além disso, ferramentas de instrumentação dinâmica são mais flexíveis, no sentido de que o código de instrumentação pode ser removido ou modificado durante a execução do programa. É importante ressaltar também que a abordagem de IDB não necessita de informações adicionais sobre o código binário, como tabelas de símbolos, e tampouco requer o código-fonte das aplicações. Finalmente, esses sistemas permitem a análise de programas que geram códigos compilados sob demanda (*JIT-compiled*). Por tudo isso, este trabalho foca na Instrumentação Dinâmica de Binários.

1.1.2.3. Ferramentas

Como já discutido na Seção 1.1.1, existe atualmente uma grande variedade de ferramentas de instrumentação dinâmica de binários. Hoje em dia, nota-se uma grande tendência de desenvolvimento de sistemas que permitem a criação e a personalização de ferramentas de instrumentação dinâmica. Esses *frameworks* proveem ao programador a possibilidade de criar suas próprias ferramentas, de acordo com as suas necessidades. Isso permite reduzir o custo da instrumentação, uma vez que a ferramenta só instrumentará partes de interesse no programa. Nessa seção, três dos *frameworks* mais populares para criação de ferramentas de instrumentação dinâmica serão discutidos. São eles: DynamoRIO [Garnett 2003], Valgrind [Nethercote and Seward 2007b] e Pin [Reddi et al. 2004]. As ferramentas de instrumentação criadas com os dois primeiros são escritas em C, ao passo em que o terceiro utiliza C++.

DynamoRIO é um *framework* de instrumentação dinâmica de binários cujo foco é a otimização dinâmica. Por esse motivo, seus criadores se preocuparam com a implementação de um sistema que tivesse um custo de execução baixo. Para isso, ele utiliza um sistema de cache que executa blocos de instruções do programa original à medida que os encontra. Dos três *frameworks* discutidos nesta Seção, DynamoRIO foi mostrado ser o mais eficiente em termos de tempo de execução [Rodríguez et al. 2014]. Atualmente, o *framework* tem suporte para quatro arquiteturas: IA-32, AMD64, ARM e AArch64. Ele pode ser executado em sistemas Windows, Linux e Android.

O segundo *framework* mencionado, Valgrind, tem seu foco na construção de ferramentas de análise "pesada", no sentido de que examinam grandes quantidades de dados, acessados e atualizados em padrões irregulares. Um dos mais populares exemplos dessas ferramentas criadas com Valgrind é a Memcheck. Essa ferramenta mantém um registro das posições de memória utilizadas por uma aplicação que são indefinidas (não inicializadas ou derivadas de outras posições indefinidas), a fim de detectar acessos não seguros à memória. Justamente por seu foco em ferramentas de análise "pesada", o *framework* apresenta os piores índices de eficiência em termos de aumento de tempo de execução reportados na literatura [Rodríguez et al. 2014]. Valgrind apresenta maior flexibilidade do que DynamoRIO em termos de suporte a sistemas e arquiteturas, podendo ser executado nas arquiteturas x86, AMD64, ARM, ARM64, PPC32/64/64LE, S390X e MIPS32/64, e nos sistemas operacionais Linux, Solaris, Android e Darwin.

Por fim, o *framework* Pin da Intel funciona de forma semelhante a seus concorren-

tes. Os usuários escrevem ferramentas de instrumentação dinâmica em C++, utilizando a API[1] do *framework*. Sua API é a mais rica dos três *frameworks* analisados e possui uma grande coleção de funções dedicadas a sistemas x86, o que fornece ao programador maior nível de detalhe nos dados coletados. Em termos de custo de tempo de execução, o Pin apresenta um desempenho próximo do concorrente mais veloz (DynamoRIO) [Rodríguez et al. 2014]. A fim de oferecer instrumentação eficiente, ele utiliza um compilador *just-in-time* para inserir código de instrumentação. Além disso, ao contrário dos seus concorrentes, não precisa da assistência do usuário para otimizar o desempenho de suas ferramentas. Outro fator que o diferencia dos outros dois *frameworks* aqui analisados é a possibilidade de anexá-lo a um processo sendo executado e, em seguida, desvinculá-lo [Luk et al. 2005]. Consideramos que, devido à sua extensa API e ao desempenho competitivo em relação a seus concorrentes, o Pin é um bom ponto de partida para um programador interessado em se iniciar na análise dinâmica de binários. Portanto, o escolhemos como ferramenta a ser utilizada no nosso estudo da técnica de IDB. A próxima seção examina esse *framework* e fornece um guia sobre sua utilização.

1.2. Pin

Nesta seção, o *framework* Pin é apresentado. Em um primeiro momento (Seção 1.2.1), as características da ferramenta serão mostradas para informar o leitor sobre as capacidades e limitações do *framework*. Também serão discutidos aspectos técnicos gerais, como linguagens de programação e suporte a *multithreading*. Em seguida, o curso abordará tópicos específicos sobre funcionamento, instalação, compilação de ferramentas e execução de códigos com o Pin.

1.2.1. Funcionamento

O Pin é um *framework* para criação de ferramentas de IDB. Portanto, o programador precisa utilizar a interface fornecida pelo *framework* para especificar qual será o código aplicado ao programa instrumentado e em quais locais do programa esse código será inserido. Para isso, o programador cria um programa escrito em C++, que será sua ferramenta de IDB. As ferramentas de IDB criadas com o Pin são chamadas de *Pintools*. Atualmente, o *framework* pode ser utilizado nas arquiteturas IA-32, Intel64 e MIC, e nos sistemas operacionais Windows, Linux, OSX e Android.

O Pin pode ser utilizado com diversos tipos de aplicações. O programador só precisa especificar o arquivo executável ao qual o *framework* será anexado. Atualmente, ele tem suporte para aplicações *multithread* e para tratamento de sinais e de exceções. Para conseguir tempo de execução competitivo, o Pin aplica uma série de otimizações de compilação ao código de instrumentação criado pelo programador. Com o Pin, é possível examinar qualquer instrução executada por uma aplicação, mesmo aquelas que pertencem a bibliotecas compartilhadas. Também é possível instrumentar qualquer chamada de função. Além disso, o programador pode escolher instrumentar, ao mesmo tempo, um processo e todos os outros na sua árvore (os processos criados direta e indiretamente por ele) [Devor 2013].

Quando o Pin é executado, são especificados, no mínimo, a ferramenta de IDB a

[1] *Application Programming Interface: rotinas fornecidas para a construção de algum software.*

ser utilizada e o programa a ser instrumentado. Abstraindo alguns detalhes, o *framework* executa como descrito nos passos a seguir:

1. As rotinas de inicialização e outros dados do *framework*, da ferramenta de IDB e do programa alvo são carregados em memória.
2. O Pin começa a execução da ferramenta de IDB, que executa suas próprias rotinas de inicialização.
3. A ferramenta de IDB lê a primeira sequência de instruções da aplicação alvo.
4. A ferramenta de IDB adiciona código de instrumentação à sequência de instruções lida no passo anterior, de acordo com a especificação do programador.
5. O código obtido (aplicação alvo e código de instrumentação) é levado à cache de código do Pin.
6. O código presente na cache é executado e, então, volta-se ao passo 3 para as próximas porções de código da aplicação alvo.

Além do funcionamento regular apresentado acima, o *framework* também trata as chamadas de sistema, os eventos de *threads* e os sinais enviados à aplicação. Caso um desses eventos aconteça, o Pin recorre ao sistema operacional. A Figura 1.1 ilustra o funcionamento do Pin em memória e sua interação com o sistema operacional. Analisando o esquema de execução descrito acima, percebe-se que o custo em tempo de execução causado pelas ferramentas de IDB vem, principalmente, do grande número de trocas de contexto realizadas entre o *framework* utilizado, o sistema operacional e as ferramentas em si.

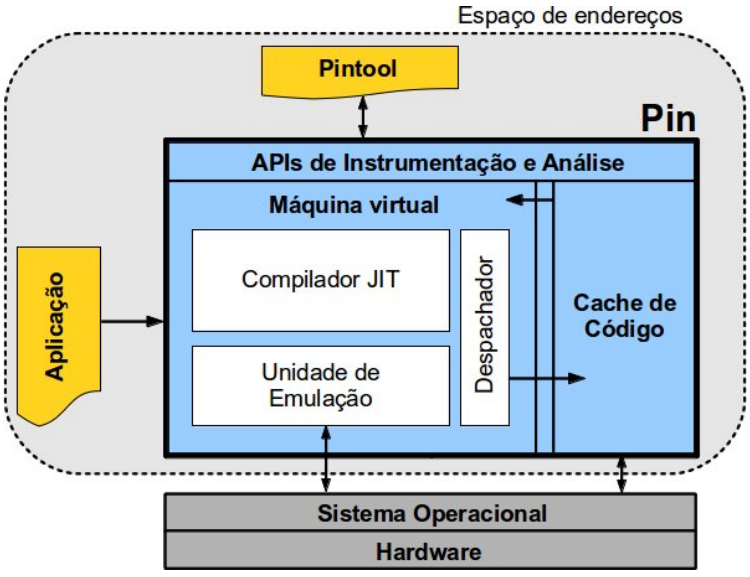


Figura 1.1. Esquema do *framework* Pin em memória. Na figura, a interação entre o Pin, a ferramenta de IDB, a aplicação instrumentada e o sistema operacional. [Ferreira et al. 2014].

1.2.1.1. JIT x Probe

Atualmente, o Pin provê dois métodos de instrumentação. O primeiro, sob demanda (*just-in-time* - JIT), cujo funcionamento foi descrito no passo a passo anterior, é o mais comum. Com ele, o Pin cria, incrementalmente, uma versão modificada da aplicação sendo instrumentada. Isso significa que, à medida que o *framework* encontra novas porções de código da aplicação alvo, ele insere porções de código de instrumentação, de forma que o código original da aplicação nunca é executado, apenas o código gerado a partir da instrumentação. Já o modo de sondagem (*probe*) só pode ser utilizado para instrumentar a aplicação alvo com granularidade de rotinas. Nesse caso, o *framework* modifica as instruções originais da aplicação, inserindo saltos para código de instrumentação, antes ou depois da execução das rotinas da aplicação alvo. Por exemplo, no caso de uma ferramenta que instrumenta as rotinas de uma aplicação antes de sua execução, este é o procedimento:

1. Imediatamente antes do início de uma função *foo*, Pin insere uma instrução de salto para a primeira instrução da função de instrumentação.
2. Ao fim do código da função de instrumentação, *foo* é chamada.
3. Ao fim do código de *foo*, uma instrução de salto redireciona a execução da aplicação para o ponto imediatamente após *foo*, no código do programa.

O modo de instrumentação sob demanda é mais comum e mais flexível, uma vez que algumas funções da API do Pin estão disponíveis somente para esse modo e ele funciona para todos os níveis de granularidade descritos na Seção 1.2.2. Entretanto, o modo de sondagem pode ser uma opção melhor para o programador que precisa de ferramentas com custo de tempo de execução reduzido. Isso se deve ao fato de que o modo de sondagem executa o código original da aplicação e reduz o número de trocas de contexto entre ela e o *framework*.

1.2.1.2. Análise x Instrumentação

Em relação ao processo de construção das ferramentas de IDB, Pin utiliza uma distinção importante entre rotinas de análise e rotinas de instrumentação. Como já mencionado, o programador precisa definir, ao criar sua *Pintool*, em quais regiões do código da aplicação alvo inserir o código de instrumentação e qual código inserir nessas regiões. Isso é feito através das rotinas de instrumentação e análise, respectivamente, e o *framework* fornece uma interface para que o programador defina essas rotinas. Para a criação de rotinas de análise, o programador deve identificar quais dados da execução do programa ele quer coletar e usar as funções apropriadas da API do *framework*. Entretanto, para a criação de rotinas de instrumentação, ele deve conhecer as opções de granularidade que Pin provê ao usuário. Essas opções são apresentadas na seção seguinte.

1.2.2. Granularidade

As opções de granularidade de instrumentação do Pin são apresentadas na Tabela 1.1.

Granularidade	Descrição
Instrução (INS)	Define uma instrução do arquivo executável do programa alvo. É a opção de granularidade que fornece o maior nível de detalhes, uma vez que é possível verificar como cada instrução afeta o estado da máquina. Por outro lado, é a opção que causa maior sobrecarga em tempo de execução.
Bloco básico (BBL)	Um bloco básico é uma sequência de instruções que tem somente um ponto de entrada e somente um ponto de saída. Dessa forma, um bloco básico é sempre atingido e abandonado através de uma instrução de desvio.
Traço (TRACE)	Sequência de instruções que têm somente um ponto de entrada, mas vários pontos de saída. Traços geralmente começam nos alvos de instruções de desvio e terminam em uma instrução de desvio incondicional, incluindo chamadas de função e instruções de retorno. Como o *framework* descobre o fluxo de execução da aplicação alvo à medida que ela é executada, ele constrói traços de forma incremental. Dessa forma, caso uma instrução de desvio seja encontrada no meio de um traço, ele é quebrado em dois blocos básicos.
Rotina (RTN)	Uma rotina se refere a uma função (ou procedimento) como gerada por um compilador para linguagens procedurais. Para que o *framework* consiga identificar essas funções, é necessário que ele tenha acesso às informações de tabela de símbolos criada pelo compilador da aplicação instrumentada. Para tornar essas informações disponíveis para o Pin, o programador deve chamar a função **PIN_InitSymbols()** antes que a aplicação comece a ser executada.
Imagem (IMG)	Uma imagem é o nível de granularidade que define a maior abstração em termos de tamanho da sequência de código instrumentada. Esse nível de granularidade se refere à instrumentação de todo o executável da aplicação alvo, além das bibliotecas que a aplicação utiliza. É importante ressaltar que os objetos do tipo **IMG** só são criados se a biblioteca compartilhada correspondente é carregada.
Seção (SEC)	Seções são as unidades que compõem uma imagem (IMG). Elas podem ser mapeadas ou não mapeadas. No primeiro caso, a seção ocupa um espaço de endereçamento dentro da imagem da qual faz parte. Essas seções foram modeladas no *framework* com base nas seções de arquivos de imagem *elf*.

Tabela 1.1. Opções de granularidade de instrumentação fornecidas pelo *framework* Pin.

O conceito de granularidade define o nível de abstração ou detalhe com que a

ferramenta de IDB instrumenta o programa alvo. Isto é, o programador, ao escrever sua ferramenta, precisa dizer ao *framework* quais são as porções de código que serão lidas do programa alvo, levadas à memória e instrumentadas a cada passo, como descrito na Seção 1.2.1. Esse processo é facilitado pela API do Pin, que define, ao todo, 6 opções de granularidade [Intel 2018b].

De maneira geral, o programador deve, além de determinar os níveis de granularidade que sua ferramenta utilizará, definir o ponto de inserção do código de instrumentação em relação à sequência de instruções instrumentada. No Pin, esse ponto de inserção pode ser antes da sequência de instruções, depois dela, dentro (em qualquer lugar) ou em seus desvios tomados (Seção 1.2.5). É importante notar que algumas dessas opções não estão disponíveis para todos os níveis de granularidade.

1.2.3. Instalação

Para começar o processo de criação de *Pintools*, é necessário instalar e configurar o ambiente de execução e compilação do Pin. Esta seção fornece um guia completo para esse processo, tendo como referência dois sistemas operacionais: Windows e Linux. Serão apresentadas informações sobre o download e a configuração necessária para executar o *framework* sem erros. Ao fim do passo a passo, o leitor poderá iniciar o processo de escrita de sua primeira *Pintool*.

O primeiro passo para instalar o Pin é identificar a versão apropriada para o sistema operacional e a arquitetura da máquina do programador. No *website* do *framework* [Intel 2012], a aba *Downloads* fornece várias opções para fazer o *download* do Pin. Após realizar o *download* e extração dos arquivos do Pin, as instruções são específicas para cada sistema operacional, como descrito nas seções seguintes.

1.2.3.1. Windows

O processo de instalação do Pin nas plataformas Windows é mais extenso do que nas plataformas Linux. Isso se deve ao fato de que muitos dos utilitários usados para execução e compilação das *Pintools* já são instalados por padrão nas distribuições Linux. O passo a passo completo é mostrado a seguir.

1. Download e instalação do ambiente de desenvolvimento Microsoft Visual Studio C++. Esse ambiente é necessário para a compilação das *Pintools*.

2. *Download* e instalação das ferramentas Cygwin. O pacote Cygwin é uma coleção de ferramentas GNU [GNU 2018] e outras de código aberto que fornecem, em plataformas Windows, grande parte da funcionalidade existente em distribuições Linux. Durante a instalação, o usuário será perguntado quais pacotes instalar. Nesse ponto, o único pacote necessário para o Pin é o pacote *Devel*, que deve ser marcado para instalação (opção *Install*). A Figura 1.2 ilustra esse passo.

3. Nas variáveis de ambiente do sistema operacional, adicione o diretório em que o Pin foi extraído na variável $PATH$. Apesar de não serobrigatório, esse passo permite que o programador execute o *framework* na linha de comandos a partir de qualquer

diretório, sem especificar o caminho completo para o diretório em que o Pin foi extraído.

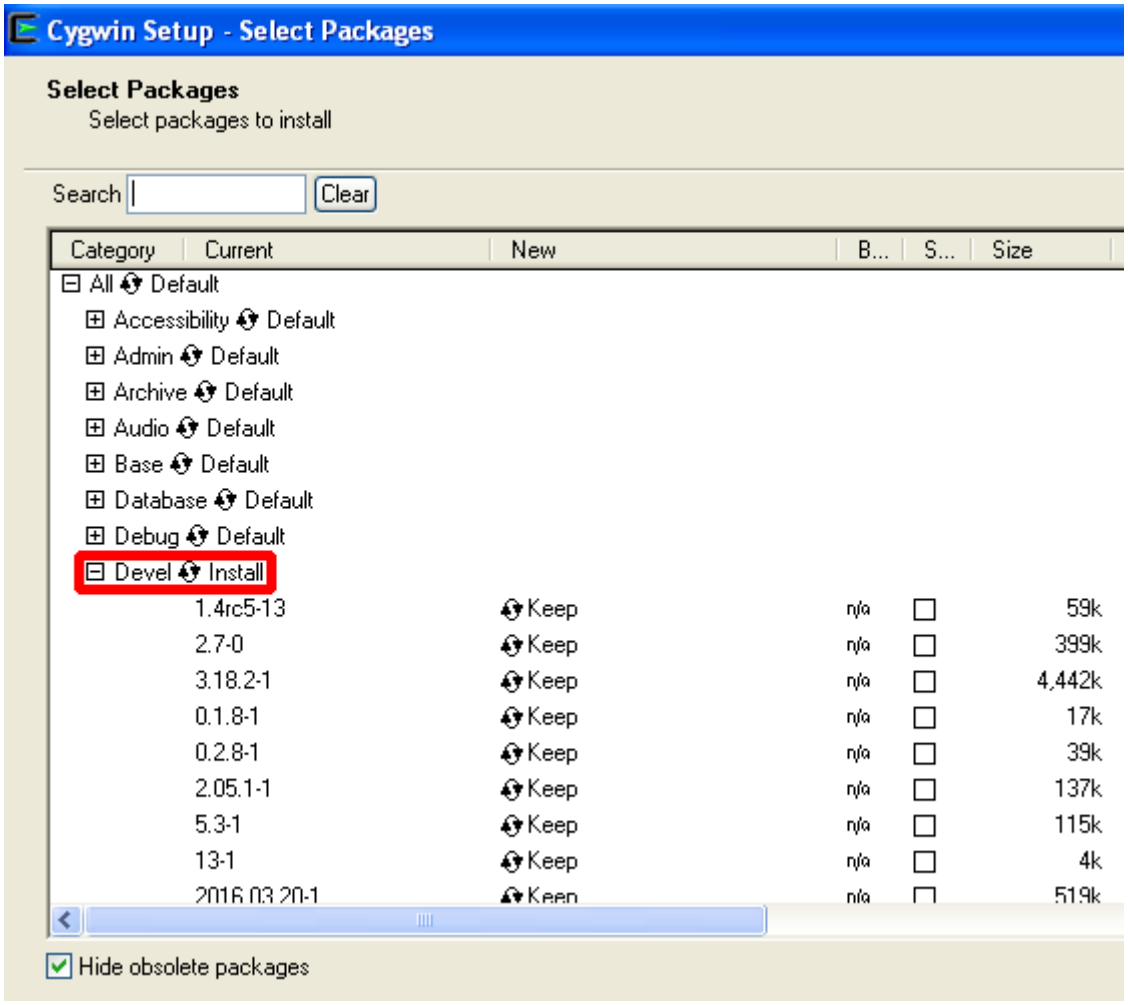


Figura 1.2. Instalação do pacote *Devel* através da coleção Cygwin.

Para compilar e executar o Pin em plataformas Windows, o usuário deve utilizar o terminal de comandos (*Command Prompt*) do Microsoft Visual Studio C++.

1.2.3.2. Linux

Como já foi dito, as distribuições Linux geralmente já possuem, por padrão, os utilitários que são requisitos para a compilação e a execução das *Pintools*. Usuários de plataformas Linux devem possuir em suas máquinas os seguintes programas instalados:

- Compilador *g++*. Necessário para compilação das *Pintools*, que são programas escritos em C++.
- Utilitário *make*. Necessário para automatizar o processo de compilação utilizado pelo Pin.

Em plataformas Ubuntu, o usuário pode instalar a versão mais recente de ambos com:

```
sudo apt-get update
sudo apt-get install g++
sudo apt-get install make
```

De forma similar ao processo de instalação do Pin em plataformas Windows, o usuário pode desejar executar o *framework* a partir de qualquer diretório, sem especificar o diretório em que ele foi extraído. Para isso, é necessário incluir o diretório de extração à variável PATH do sistema. O que pode ser feito da seguinte forma em sistemas Ubuntu:

1. Edição do arquivo **environment** utilizando qualquer editor de texto simples. Aqui, usaremos *vim*.

   ```
   sudo vim /etc/environment
   ```

2. Adição do diretório do Pin à variável PATH. Para isso, inserir o caminho completo do diretório onde o Pin foi extraído dentro das aspas duplas da definição da variável, após dois pontos. O comando de definição da variável será semelhante ao mostrado abaixo, onde o diretório de extração do *framework* foi "/home/hugo/pin-gcc-linux".

   ```
   PATH="/usr/local/bin:/home/hugo/pin-gcc-linux"
   ```

   Para que a mudança tenha efeito na sessão atual, basta executar o comando:

   ```
   source /etc/environment
   ```

Após os passos anteriores, o usuário já pode compilar e executar *Pintools* em plataformas Linux.

1.2.3.3. Compilação e Execução

Com o *framework* devidamente instalado em sua máquina, o programador pode compilar e executar sua primeira *Pintool*. É importante notar, nesse ponto, que as as *Pintools* são bibliotecas de carga dinâmica e, como tal, possuem implementações diferentes de acordo com o sistema operacional utilizado, o que se reflete nos formatos das ferramentas em cada sistema. Para Windows, elas têm formato **dll**, enquanto para Linux, o formato é **so**. Como primeiro exemplo[2], uma *Pintool* simples será compilada e executada. A *Pintool* de escolha para ilustrar o processo de compilação e execução é a **inscount0.cpp**, que simplesmente conta o número de instruções executadas pela aplicação alvo. Para Windows, a aplicação instrumentada será a calculadora (**calc.exe**) e para Linux, a aplicação instrumentação será o comando **ls**, que lista os arquivos do diretório atual. As Tabelas 1.2 e 1.3 mostram os comandos de compilação para Windows e Linux em uma máquina Intel. Em negrito, os comandos para execução do *framework*.

[2]No momento do desenvolvimento deste trabalho, a versão do Pin utilizada é a 3.6, lançada em Fevereiro de 2018.

	Comandos
32bit	cd source\tools\ManualExamples make dir TARGET=ia32 obj-ia32/inscount0.dll **pin -t obj-ia32\inscount0.dll -- calc.exe**
64bit	cd source\tools\ManualExamples make dir obj-intel64/inscount0.dll **pin -t obj-intel64\inscount0.dll -- calc.exe**

Tabela 1.2. Comandos para compilação da ferramenta "inscount0.cpp"e execução do Pin em máquinas Intel 32 e 64 bit com sistema operacional Windows.

	Comandos
32bit	cd source/tools/ManualExamples make TARGET=ia32 obj-ia32/inscount0.so **./pin -t obj-ia32/inscount0.so -- /bin/ls**
64bit	cd source/tools/ManualExamples make obj-intel64/inscount0.so **./pin -t obj-intel64/inscount0.so -- /bin/ls**

Tabela 1.3. Comandos para compilação da ferramenta "inscount0.cpp"e execução do Pin em máquinas Intel 32 e 64 bit com sistema operacional Linux.

Para que consiga utilizar a estrutura dos arquivos *Makefile* do Pin, ao escrever suas *Pintools*, o programador deve incluir seu código fonte em C++ em um dos seguintes diretórios:

source / tools / ManualExamples
source / tools / SimpleExamples

Ao executar a ferramenta "inscount0", será gerado pelo Pin um arquivo chamado "inscount.out", que contém sua saída. A execução do *framework* através da linha de comandos segue a seguinte estrutura:

pin [Pin Args] [-t <Pintool> [Pintool Args]] – <App> [App Args]

Nessa chamada, **[Pin Args]** refere-se aos argumentos passados ao próprio *framework*. Existe uma grande variedade de argumentos que podem ser passados ao Pin e eles definem diversas funcionalidades. Por exemplo, o argumento **-pid** seguido do número de um processo sendo executado diz ao Pin para instrumentar esse processo. Todos os argumentos do Pin estão descritos em sua API. A opção **-t** especifica qual será a *Pintool* (<Pintool>) utilizada durante a instrumentação. Além disso, as próprias *Pintools* também podem receber argumentos. Nesse caso, eles ([Pintool Args]) são especificados logo após o nome da ferramenta. Finalmente, os caracteres **--** separam o Pin e a *Pintool* utilizada da aplicação instrumentada. O arquivo executável da aplicação é especificado em **<App>**. Ele também pode receber argumentos que, por sua vez, são especificados em **[App Args]**.

1.2.4. Estrutura das Pintools

Esta seção tem o objetivo de fornecer ao leitor o conhecimento necessário para a criação de sua primeira Pintool. Para isso, teremos como referência a mesma *Pintool* da seção anterior, "inscount0.cpp"[3], disponível no pacote do Pin, na pasta "source/tools/ManualExamples". Mesmo que essa seja uma ferramenta simples, ela contém todas as partes fundamentais de uma *Pintool*. Logo, a estrutura dessa ferramenta será analisada com detalhes a seguir.

```
31 #include <iostream>
32 #include <fstream>
33 #include "pin.H"
```

A primeira coisa a ser feita em uma *Pintool* é a inclusão do arquivo de cabeçalho **pin.H**. É através dele que a ferramenta terá acesso a toda a API do Pin. Começando pela função principal da ferramenta:

```
80 int main(int argc, char * argv[])
81 {
82     // Inicia o pin
83     if (PIN_Init(argc, argv)) return Usage();
```

A função **Pin_Init** é responsável por inicializar o *framework*. Ela devolve um valor booleano que indica se houve algum erro na chamada do Pin. No código da ferramenta, caso isso aconteça, ela imprime uma mensagem informativa sobre sua chamada e termina a execução. Em seguida, usa-se o código:

```
88     INS_AddInstrumentFunction(Instruction, 0);
```

O uso dessa função no código revela não só o nível de granularidade usado pelo programador, mas também o nome da rotina de instrumentação da ferramenta (Instruction). Aqui, o programador instrumenta todas as instruções (INS) da aplicação alvo. Como o nome da própria função da API sugere (*AddInstrumentFunction*), o usuário define sua rotina de instrumentação ao passar a função *Instruction* para esse procedimento da API do Pin. Como já discutido, a rotina de instrumentação define onde o código de análise será inserido. Estudando o código da função *Instruction*, é possível entender como isso acontece no Pin.

```
45 VOID Instruction(INS ins, VOID *v)
46 {
47     // Insere uma chamada para "docount" antes de todas as
          instruções
48     INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount,
          IARG_END);
49 }
```

Vemos que essa rotina de instrumentação recebe um objeto do tipo INS. Trata-se da instrução a ser instrumentada. O corpo da rotina faz uso de outra função da API do Pin.

[3]Os códigos completos das *Pintools* analisadas estão disponíveis no repositório: https://github.com/ha2398/jai-2018-pin

A função **INS_InsertCall** define onde, em relação à instrução, o código de análise será inserido. Aqui, a opção do programador foi inserir código de análise antes da execução da instrução, o que é especificado pelo parâmetro **IPOINT_BEFORE**. Em seguida, a função recebe um ponteiro de função (**AFUNPTR)** que indica a rotina de análise propriamente dita. Além disso, a lista de argumentos - vazia, neste caso - vem após a especificação da rotina de análise e sempre termina com o identificador **IARG_END**. A rotina de análise é então definida na *Pintool*:

```
41 // Essa função é executada antes que toda instrução execute
42 VOID docount() { icount++; }
```

Como definido pela chamada de **INS_InsertCall**, o código da função **docount** será inserido antes de toda instrução executada. Essa é uma função simples, que apenas incrementa um contador das instruções encontradas até o momento. Voltando à função **main** da *Pintool*, o código é finalizado com:

```
90     // Registra a função "Fini" para ser chamada quando a aplica
          ção termina
91     PIN_AddFiniFunction(Fini, 0);
92
93     // Começa o programa, nunca retorna.
94     PIN_StartProgram();
95
96     return 0;
97 }
```

A função **PIN_AddFiniFunction** especifica uma função definida pelo programador (**Fini**) para ser executada ao fim da execução da aplicação alvo. No caso da *Pintool* "inscount0.cpp", a função **Fini** simplesmente escreve o valor do contador de instruções em um arquivo. A última chamada da função principal da ferramenta é **PIN_StartProgram**, que nunca devolve nenhum valor. Essa função é responsável por finalmente executar a aplicação instrumentada após todas as definições de rotinas de inicialização, finalização, análise e instrumentação.

1.2.5. Tópicos e API

O objetivo desta seção é fornecer uma visão detalhada sobre alguns tópicos de maior relevância dentro da API do *framework*. Sendo assim, a seção não tem como objetivo ser um guia completo de todas as funções fornecidas pelo Pin, mas um panorama que introduzirá algumas de suas funcionalidades mais interessantes ao leitor.

O primeiro tópico a ser discutido é a **passagem de parâmetros** para as rotinas de análise. Como discutido na Seção 1.2.4, o Pin define algumas rotinas cujo propósito é especificar quais serão as funções de análise, para qual tipo de granularidade elas serão aplicadas e onde nas sequências de código da aplicação o código de instrumentação será inserido. Dentro da rotina de instrumentação (especificada por alguma função **AddInstrumentFunction**), a rotina de análise é especificada com alguma função **InsertCall**. Como exemplo, temos a invocação de função a seguir.

```
1 INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) count, IARG_END);
```

As funções **InsertCall** têm uma estrutura bem definida. Nelas, o primeiro argumento é um objeto do tipo que define a granularidade de instrumentação (nesse exemplo, o objeto **ins**, de tipo **INS**, que define a granularidade de instrumentação de instruções). Em seguida, o ponto de inserção do código de análise é especificado, em relação à unidade de granularidade escolhida. A Tabela 1.4 mostra as opções disponíveis.

Ponto de Inserção	Descrição
IPOINT_BEFORE	Insere o código de análise antes da sequência de instruções definida pela unidade de granularidade. Disponível para INS e RTN.
IPOINT_AFTER	Insere o código de análise depois da sequência de instruções definida pela unidade de granularidade. Disponível para INS e RTN.
IPOINT_ANYWHERE	Insere o código de análise dentro da sequência de instruções definida pela unidade de granularidade. Disponível para BBL e TRACE.
IPOINT_TAKEN_BRANCH	Insere o código de análise quando o fluxo de execução encontra uma instrução de desvio que é tomado. Disponível para INS.

Tabela 1.4. Pontos de inserção de código de análise em relação à unidade de granularidade utilizada.

Após especificar o ponto de inserção do código de análise, o programador diz ao *framework* qual deve ser a função que contém o código de análise a ser executado e quais são os argumentos que essa função recebe. Como o número de argumentos não tem um tamanho fixo, o Pin define uma sintaxe específica para definir a lista de argumentos da função de análise, como mostrado a seguir.

```
1 ... (AFUNPTR) analysis, [IARGLIST], IARG_END);
```

No exemplo acima, a função de análise recebe o nome **analysis**. Além disso, **IARGLIST** define toda a lista de parâmetros que ela recebe. Essa lista pode receber tanto atributos do Pin (**IARG**) quanto variáveis definidas pelo usuário. No segundo caso, o programador deve passar como parâmetro, antes da variável em si, o seu tipo. É importante notar que alguns dos **IARGs** requerem argumentos adicionais. A chamada de função seguinte exemplifica esse processo:

```
1 BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)docount,
2   IARG_UINT32, BBL_NumIns(bbl), IARG_END);
```

Nesse caso, o código da função de análise **docount** será inserido antes dos blocos básicos do programa. Também está especificado que a função **docount** recebe um valor de tipo **UINT32** (resultado da chamada **BBL_NumIns(bbl)**).

Um dos tópicos citados anteriormente como exemplos de aplicação da técnica de IDB é a detecção de erros e análise de memória. Sendo assim, é importante estudar como

fazer **leituras do espaço de endereçamento** e obter informações sobre a memória através do Pin. Para isso, usaremos a *Pintool* "pinatrace.cpp", disponível no diretório “source/-tools/ManualExamples”. Essa ferramenta instrumenta cada instrução da aplicação alvo, porém faz uso da API do Pin para instrumentar somente as instruções que fazem leituras e escritas na memória. Sua função de instrumentação é mostrada a seguir.

```
53 /**
54  * Chamada para toda instrução e somente adiciona código de
55  * análise para instruções de leitura e escrita em memória.
56  */
57 VOID Instruction(INS ins, VOID *v)
58 {
59     /**
60      * O uso da função INS_InsertPredicatedCALL faz com
61      * que a instrumentação seja chamada se, e somente se,
62      * a instrução da aplicação alvo for de fato executada.
63      */
64     UINT32 memOperands = INS_MemoryOperandCount(ins);
65 
66     // Itera sobre cada operando de memória da instrução.
67     for (UINT32 memOp = 0; memOp < memOperands; memOp++)
68     {
69         if (INS_MemoryOperandIsRead(ins, memOp))
70         {
71             INS_InsertPredicatedCall(
72                 ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
73                 IARG_INST_PTR,
74                 IARG_MEMORYOP_EA, memOp,
75                 IARG_END);
76         }
77 
78         /**
79          * Importante notar que, em algumas arquiteturas,
80          * um único operando de memória pode ser tanto lido
81          * quanto escrito em uma mesma instrução. Nesse caso,
82          * a ferramenta realiza a instrumentação duas vezes,
83          * uma para a leitura e outra para a escrita.
84          */
85         if (INS_MemoryOperandIsWritten(ins, memOp))
86         {
87             INS_InsertPredicatedCall(
88                 ins, IPOINT_BEFORE, (AFUNPTR)RecordMemWrite,
89                 IARG_INST_PTR,
90                 IARG_MEMORYOP_EA, memOp,
91                 IARG_END);
92         }
93     }
94 }
```

Na ferramenta, a identificação das instruções que referenciam a memória é feita através da chamada de **INS_MemoryOperandCount**, que devolve o número de operandos de memória que a instrução contém. Caso esse valor seja 0, o laço **for**, onde a instrumentação é aplicada, não é executado. Dentro desse laço, a ferramenta checa se o operando de memória da instrução é lido ou escrito, e adiciona a função de análise adequada a cada caso. Para essa ferramenta, em particular, as funções de análise simplesmente escrevem em um arquivo os endereços de memória referenciados.

A ferramenta "pinatrace.cpp"monitora o espaço de endereçamento através do monitoramento de cada instrução, verificando aquelas que alteram a memória diretamente. Entretanto, esse não é o único modo de inspecionar a memória usando Pin. O *framework* também fornece funções para leitura direta dos *bytes* em um endereço de memória, como mostrado a seguir.

```
1 PIN_SafeCopy(VOID* dst, const VOID* src, size_t size);
2 PIN_SafeCopyEx(VOID* dst, const VOID* src, size_t size,
3   EXCEPTION_INFO * pExceptInfo);
```

A função **PIN_SafeCopy** copia para o destino **dst** o número de *bytes* especificados em **size** da região de memória endereçada por **src**. O *framework* garante que a região de memória acessada tenha, de fato, o conteúdo original da memória da aplicação instrumentada. A função **PIN_SafeCopyEx** tem a mesma funcionalidade, além de fornecer informação detalhada caso algum erro tenha ocorrido. Nesse caso, **pExceptInfo** armazena as informações de erro.

Em conjunto com o tópico de inspeção de memória de aplicação, para obter o estado completo da máquina é necessário saber o valor que seus registradores assumem em algum momento. Pin possui estruturas de dados que fornecem uma abstração para esse processo de **recuperação de contexto**. Os tipos **CONTEXT** e **PHYSICAL_CONTEXT** armazenam todos os valores dos registradores presentes na máquina. Com eles é possível tanto fazer a leitura desses dados quanto modificá-los. A diferença entre os dois tipos se refere ao fato de que o primeiro fornece o contexto da máquina como a aplicação veria caso estivesse executando sem ser instrumentada. Já o segundo fornece o verdadeiro contexto da máquina, que é afetado pela interferência do *framework*. Para propósitos gerais, o programador deve fazer uso do tipo **CONTEXT**, que pode ser obtido nas rotinas de instrumentação. O tipo **PHYSICAL_CONTEXT** é utilizado para obter informações de depuração em caso de erros e exceções.

Para recuperar o contexto da máquina durante as rotinas de instrumentação, o programador pode optar pelos argumentos **IARG_CONTEXT** ou **IARG_CONST_CONTEXT**. A primeira opção permite leitura e escrita em registradores, enquanto a segunda permite somente leitura. Um exemplo de instrumentação com o uso dessas estruturas é mostrado a seguir.

```
1 VOID analysis(CONTEXT *ctxt) {
2   ...
3 }
4
5 ...
6
```

```
7  VOID instrumentation(INS ins, VOID *v) {
8      INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) analysis,
9              IARG_CONST_CONTEXT, IARG_END);
10 }
```

Uma vez obtido o objeto do tipo **CONTEXT**, o programador pode obter o valor de qualquer registrador da máquina (para esse processo, o Pin só fornece suporte para arquiteturas Intel). A relação completa dos registradores das arquiteturas Intel, juntamente com seus nomes dentro do *framework* podem ser encontrados no Guia do Usuário do Pin [Intel 2018b]. A fim de ilustração, alguns exemplos são mostrados abaixo.

```
1  VOID analysis(CONTEXT *ctxt) {
2      ADDRINT registerEAX;
3      ADDRINT registerESI;
4
5      registerEAX = PIN_GetContextReg(ctxt, REG_EAX);
6      registerESI = PIN_GetContextReg(ctxt, REG_ESI);
7  }
```

A fim de controlar a interferência de fatores externos ao processo da aplicação, o programador pode estar interessado em monitorar **eventos assíncronos** como os sinais das plataformas Linux ou as chamadas de procedimento assíncronas (APC - *Asynchronous Procedure Calls*) das plataformas Windows. Atualmente, existem duas formas de tratar esses eventos utilizando Pin, como descrito a seguir.

- **PIN_InterceptSignal** Essa função permite à *Pintool* interceptar os sinais recebidos pela aplicação e tratá-los como desejar. Dessa forma, uma função é definida para ser executada sempre que um sinal é recebido pela aplicação. Esse sinal pode ou não ser encaminhado para o programa instrumentado. Para chamar essa função, o programador deve especificar qual sinal deverá ser interceptado e qual função deverá ser chamada caso o sinal seja enviado à aplicação. O exemplo abaixo ilustra a função. É importante notar que essa função só pode ser utilizada em sistemas Linux.

```
1      int main(int argc, char * argv[])
2      {
3        ...
4          PIN_InterceptSignal(SIGSEGV, Intercept, 0);
5          ...
6      }
7
8      static BOOL Intercept(THREADID, INT32, CONTEXT *,
9        BOOL, const EXCEPTION_INFO *, void *)
10     {
11         /**
12          * Tratamento do sinal.
13          */
14     }
```

Na *Pintool* acima, a função **Intercept** será executada sempre que a aplicação instrumentada receber um sinal **SIGSEGV**. As funções de tratamento de sinal passadas como argumento para a função **Pin_InterceptSignal** devem possuir o mesmo cabeçalho e receber, portanto, 6 valores. Esses valores representam, na ordem da lista de argumentos: o ID da *thread* que recebeu o sinal; o número do sinal; o contexto da máquina quando a aplicação recebeu o sinal; um booleano que indica se há uma função já registrada para tratar o sinal em questão; um objeto com uma descrição de exceção, caso o sinal represente uma; o valor passado para a função na chamada de **PIN_InterceptSignal** (no exemplo acima, 0). A função devolve *true* caso tenha sido bem sucedida ao interceptar o sinal.

- **PIN_AddContextChangeFunction** Essa função registra uma função que é executada sempre que a aplicação troca de contexto devido ao recebimento de algum sinal das plataformas Linux ou um APC do Windows. Geralmente, essa função é mais utilizada quando há somente a necessidade de notificar a *Pintool* sobre o recebimento de sinais e trocas de contexto. O código abaixo ilustra seu uso.

```
1   int main(int argc, char * argv[])
2   {
3     ...
4       PIN_AddContextChangeFunction(OnAPC, 0);
5       ...
6   }
7
8   static void OnAPC(THREADID, CONTEXT_CHANGE_REASON,
9     const CONTEXT *, CONTEXT *, INT32, VOID *)
10  {
11      /**
12       * Tratamento da troca de contexto.
13       */
14  }
```

Assim como na função anterior, as funções registradas através da chamada de **PIN_AddContextChangeFunction** devem ter seu cabeçalho definido com uma estrutura definida pelo Pin. Aqui, a lista de argumentos deve receber os tipos mostrados no código, que representam, na ordem da lista: o ID da *thread* que trocou de contexto; a causa da troca de contexto (APC, exceção, sinal, etc); o contexto da máquina antes da troca de contexto; o contexto da máquina após a mudança de contexto; uma informação adicional que depende da causa da troca de contexto; o valor passado para a função na chamada de **PIN_AddContextChangeFunction**.

Como visto na Seção 1.2.3.3, o Pin admite argumentos passados pela linha de comando no momento de sua invocação. Além disso, as próprias *Pintools* podem receber argumentos. Para isso, Pin fornece uma interface para adição de parâmetros obrigatórios e opcionais chamados de **KNOBs**. Cada KNOB define um argumento que a *Pintool* receberá pela linha de comando. O código abaixo ilustra o uso de KNOBs. Nele, o KNOB declarado especifica que o nome (*string*) do arquivo de saída da *Pintool* será definido pela

linha de comando através da *flag* **-o**. Além disso, também especifica um valor padrão caso a *flag* não esteja presente na invocação do *framework*.

```
1  KNOB<string> knobArquivoSaida(KNOB_MODE_WRITEONCE, "pintool",
2    "o", "output.log", "Nome do arquivo de saida");
3
4  static ofstream arquivoSaida;
5
6  int main(int argc, char *argv[])
7  {
8    ...
9    arquivoSaida.open(knobArquivoSaida.Value().c_str());
10   ...
11 }
```

1.2.6. Documentação e Suporte

Mesmo que existam guias e tutoriais do Pin no *website* do *framework*, muitas das funções de sua API não são muito intuitivas de serem usadas ou apresentam documentação insuficiente. Dessa forma, o programador deve procurar suporte em outros sítios. Abaixo encontra-se uma relação com as principais mídias de suporte ao Pin.

- **Guia do usuário**: O guia do usuário [Intel 2018b] do Pin é a documentação mais completa do *framework*. No guia, o programador encontra uma descrição de todas as funções da API do Pin, além de exemplos de *Pintools* e algumas discussões sobre o funcionamento do *framework*.

- **Tutorial do Pin**: Apresentado no simpósio CGO[4] de 2013, o tutorial do Pin [Devor 2013] fornece um compilado das principais funcionalidades do *framework*, com exemplos de código e discussões sobre sobrecarga de tempo de execução, desafios do *framework* tanto em plataformas Windows quanto Linux, etc.

- **Grupo *Pinheads***: Fórum de discussão [Intel 2004] sobre o *framework*. Nesse grupo, o programador tem acesso a um acervo de questões e respostas feitas por outros programadores que utilizam o Pin. Além disso, também pode se inscrever e postar suas próprias perguntas (fórum em inglês).

- ***Pintools* de exemplo**: Um bom método para estudar o funcionamento da interface do Pin e a criação de *Pintools* é através do próprio código fonte de outras *Pintools*. Por isso, as ferramentas de exemplo disponíveis no próprio *kit* do *framework* são referência de documentação. Essas ferramentas estão disponíveis a partir do diretório raiz onde o Pin é extraído, nas pastas "source/tools/SimpleExamples"e "source/tools/ManualExamples".

[4]Simpósio Internacional de Geração e Otimização de Código (*International Symposium on Code Generation and Optimization*).

1.3. Estudos de casos

Os casos estudados nesta seção compreendem a análise de situações em que os autores deste curso utilizaram a IDB para realizar trabalhos de pesquisa com foco na arquitetura x86. Trata-se, portanto, de exemplos de códigos de instrumentação criados pelos autores com finalidades diversas, todas na área de segurança de software. Esses códigos são apresentados e seus detalhes são minuciosamente explicados. O objetivo é ilustrar os conceitos apresentados na Seção 1.2 do texto, consolidando o aprendizado para que os leitores consigam desenvolver seus próprios códigos de instrumentação. Além disso, esta seção serve também como uma referência rápida para consulta de como se deve utilizar várias APIs importantes do Pin, já que muitas delas possuem uma documentação limitada.

1.3.1. Conceitos de segurança

Conforme mencionado, os estudos de casos envolvem a análise de programas e o desenvolvimento de protótipos de proteções elaborados no contexto de segurança de software. Para facilitar o pleno entendimento dos códigos de instrumentação a serem discutidos, antes apresentamos alguns conceitos de segurança de software que serão exercitados nesses códigos. O objetivo desta subseção é somente garantir que o leitor seja capaz de entender o contexto em que se encaixa cada um dos exemplos estudados. Não se pretende efetuar uma análise profunda desses assuntos, já que isso fugiria do escopo deste curso.

1.3.1.1. Estouro de Memória na Pilha

O conceito de estouro de memória na pilha, comumente referenciado pela expressão em inglês *Stack Buffer Overflow*, é importante para o entendimento dos três exemplos de código de IDB a serem apresentados nas Seções 1.3.2 a 1.3.4. Para compreendê-lo, é necessário assimilar como funciona a divisão do espaço de endereçamento virtual de um processo em segmentos. A Figura 1.3 ilustra a disposição geral dos segmentos de memória alocados para um processo executado por um sistema Linux em uma arquitetura x86 de 32 bits. Apesar de alguns detalhes serem ligeiramente diferentes em outros ambientes, a disposição e a função dos segmentos é bastante similar. Por questões didáticas, omitimos algumas informações, como aquelas referentes a espaços de deslocamento utilizados para a randomização do endereço inicial dos segmentos. Fazemos isso pois o intuito é focar no entendimento da função exercida pelo segmento de pilha.

Cada processo possui um espaço de endereços virtuais exclusivo, conforme representado na Figura 1.3. No caso de arquiteturas de 32 bits, esse espaço equivale a um bloco de 4GB. Os endereços virtuais, usados pelas instruções do programa, são mapeados para endereços reais da memória através de tabelas mantidas pelo *kernel* do Sistema Operacional (SO). Como o próprio kernel do SO é um processo, ele tem uma porção do espaço de endereços virtuais reservada para si dentro da faixa de endereços de todo processo (normalmente, do endereço 0xC0000000 ao endereço 0xFFFFFFFF). O restante (do endereço 0x00000000 ao endereço 0xBFFFFFFF), que equivale a 3GB, é destinado ao uso de cada processo de usuário. Os 3GB de memória dedicados ao processo são divididos em segmentos, de acordo com o tipo de dados que armazenam.

A *Pilha* guarda variáveis locais (pertencentes ao escopo de uma única função) e

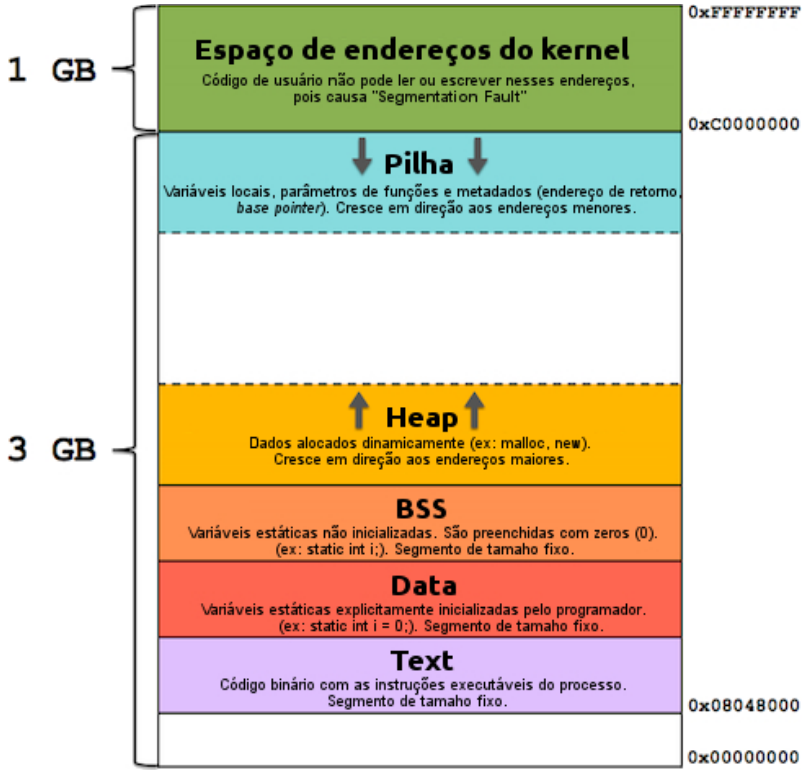


Figura 1.3. Distribuição dos segmentos de memória de um processo [Tolomei 2015].

metadados usados pelo processador para controlar chamadas de funções. O conjunto de valores armazenados para cada função executada por um programa é chamado de *Frame* da função. Cada *Frame* acomoda os valores atribuídos às variavés locais, os parâmetros de chamada da função e seu endereço de retorno. Dependendo do nível de otimização utilizado ao compilar o programa, a pilha pode armazenar também o endereço base do *Frame* da função para onde o fluxo de execução deve retornar. Nesse caso, o endereço base de cada função é mantido em um registrador denominado *Base Pointer*. O segmento de Pilha obedece às características de funcionamento da estrutura de dados de mesmo nome. Ou seja, dados novos são empilhados no topo da estrutura e são os primeiros a serem removidos. Para isso, o processador possui um registrador que indica a posição de topo da pilha, denominado *Stack Pointer*. A Pilha normalmente é posicionada nos endereços de memória mais altos, logo abaixo do espaço reservado para o kernel do SO, e cresce no sentido dos endereços de memória menores.

Durante a chamada de uma função qualquer, a pilha é alimentada com valores seguindo uma sequência de passos. Após a execução desses passos, a pilha ilustrada na parte esquerda da Figura 1.4 se transforma na pilha apresentada na parte direita da mesma figura. Posteriormente, quando a função chamada retorna, a pilha volta ao estado indicado no lado esquerdo da Figura 1.4. O processo de retorno de uma função consiste em realizar as seguintes operações, inversas ao procedimento de chamada: 1) depois de executar as instruções previstas em seu código e antes de retornar, a função chamada faz o registrador *Stack Pointer* apontar novamente para o endereço onde foi armazenado o endereço base

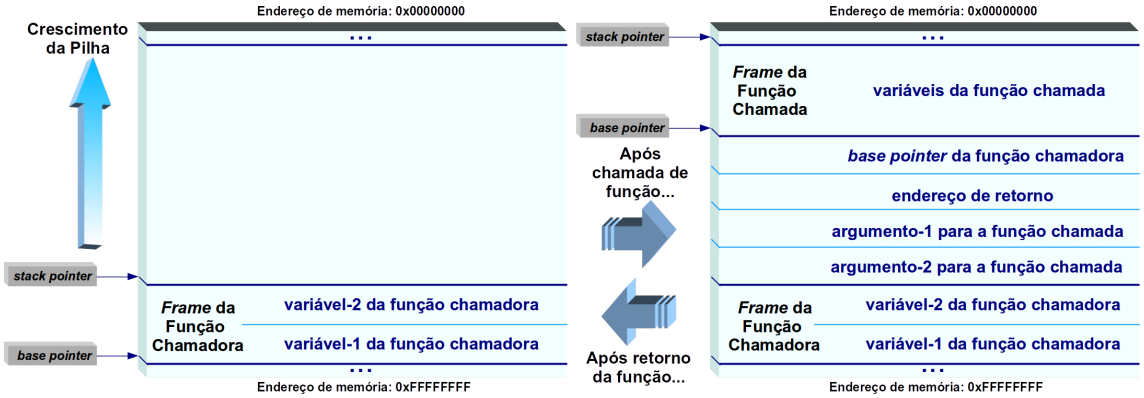


Figura 1.4. Estados da pilha em chamada e retorno de função [Tymburibá et al. 2012].

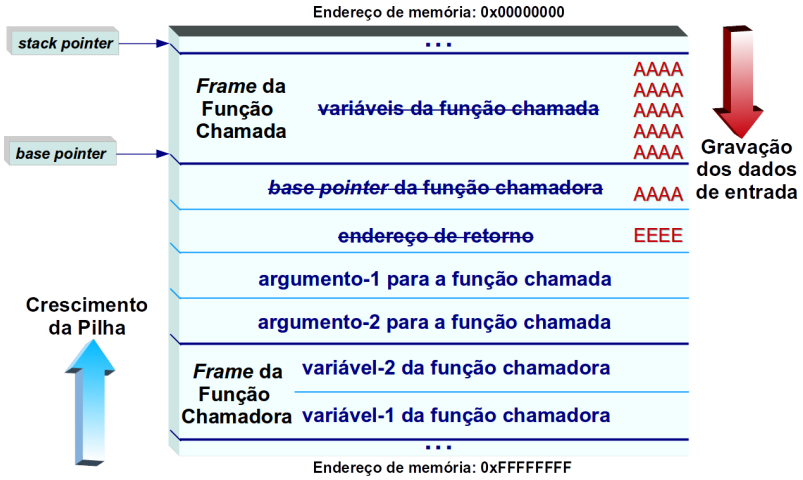


Figura 1.5. Sobrescrita de endereço de retorno ocasionada por um estouro de memória na pilha [Tymburibá et al. 2012].

do *Frame* pertencente à função chamadora; 2) o endereço base do *Frame* pertencente à função chamadora é desempilhado e restabelecido no registrador *Base Pointer*; 3) o endereço de retorno é desempilhado e o fluxo de execução é desviado para esse endereço.

O estouro de memória na pilha consiste em enviar como entrada para a aplicação uma quantidade de dados maior do que o espaço alocado na pilha para as variáveis locais da função. Para isso, basta que um usuário envie como entrada para uma aplicação que não checa o tamanho de suas entradas, uma sequência de dados maior do que a área reservada para essa entrada de usuário. Nesse cenário, uma entrada de tamanho devidamente calculado pode, portanto, extravasar os limites do *Frame* da função e sobrescrever seu endereço de retorno. Assim, quando a instrução de desvio para o endereço de retorno é executada, o fluxo de execução é transferido para um endereço escrito pelo atacante na pilha, conforme representado na Figura 1.5.

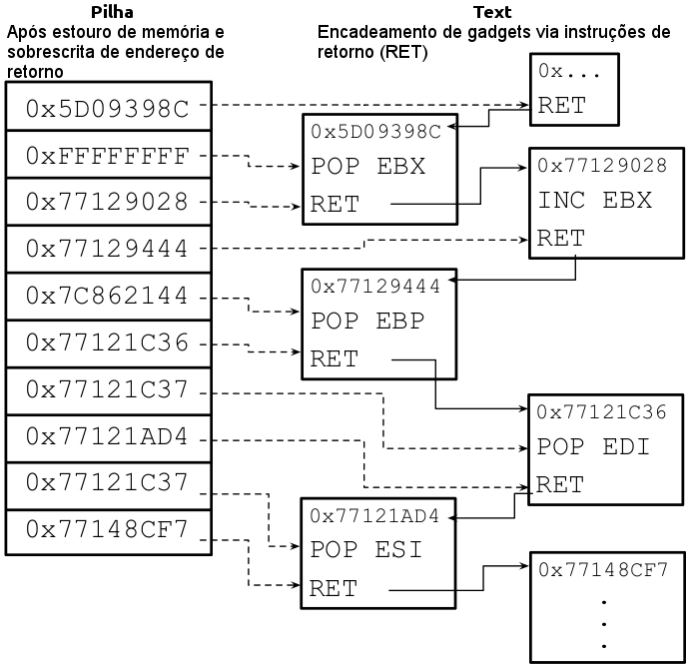


Figura 1.6. Exemplo de encadeamento de *gadgets* em um ataque do tipo ROP.

1.3.1.2. Return-Oriented Programming

Return-Oriented Programming, ou simplesmente ROP, é a principal técnica empregada atualmente por atacantes para executar códigos maliciosos em aplicações vulneráveis. Os exemplos de código de IDB apresentados nas Seções 1.3.2 a 1.3.4 fazem parte de trabalhos de pesquisa, encabeçados pelos autores deste curso, que visam propor proteções contra ataques ROP. Por conta disso, esta subseção é dedicada a introduzir os conceitos relacionados a ROP. Com isso, espera-se que os leitores compreendam a motivação e o contexto em que se enquadram os exemplos de IDB a serem apresentados.

Quando os primeiros ataques de corrupção da memória se tornaram públicos, era possível executar códigos maliciosos simplesmente inserindo-os na pilha junto com os dados que causavam o estouro da memória, exatamente como ilustrado na Figura 1.5 [One 1996]. Assim, ao induzir a sobrescrita de um endereço de retorno, bastava forçar a transferência do fluxo de execução para o endereço onde o código malicioso era armazenado na pilha. Com o intuito de impedir esse tipo de ataque, foi criado um marcador (bit de execução) que impede a execução de instruções localizadas fora do segmento executável do processo (*Text*) [Kanellos 2004]. Dessa forma, atualmente atacantes estão impedidos de transferir o fluxo de execução diretamente para códigos maliciosos inseridos na Pilha, já que esse segmento não possui permissão de execução, por ser unicamente destinado a dados. Diante desse obstáculo, o artifício encontrado por atacantes foi reutilizar instruções do próprio programa, já que os bytes correspondentes a essas instruções obrigatoriamente precisam ter permissão de execução, para que o processo funcione.

ROP baseia-se justamente no reuso de código para superar a proteção oferecida pelo bit de execução. Para isso, encadeia pequenos trechos de código da própria aplicação alvo (ou de bibliotecas utilizadas por ela), denominados *gadgets*. Para conseguir

esse encadeamento, a última instrução de cada trecho de código escolhido deve executar um desvio, conforme ilustrado na Figura 1.6. A ideia original do ROP utiliza *gadgets* finalizados com instruções de retorno (RET) para interligar as frações de código escolhidas [Shacham 2007]. Daí surgiu o nome da técnica. Posteriormente, foi demonstrado que também é possível encadear *gadgets* através de instruções de desvio indireto do tipo jump incondicional (JMP) [Checkoway et al. 2009, Chen et al. 2011] ou do tipo chamada de função (CALL) [Carlini and Wagner 2014, Göktas et al. 2014]. Através desse encadeamento de sequências de código, atacantes são capazes de superar a proteção do bit de execução para executar códigos maliciosos arbitrários.

A Figura 1.6 representa parte da estrutura de um *exploit* ROP disponível publicamente [Blake 2011]. Os dados inseridos pelo usuário na pilha, que ocasionam o estouro de memória e a consequente sobrescrita de um endereço de retorno, estão indicados à esquerda na figura, em formato hexadecimal. As linhas tracejadas que partem desses dados apontam qual instrução os utiliza. No lado direito da Figura 1.6, as caixinhas representam os *gadgets* que, encadeados, compõem o ataque. O encadeamento dessas pequenas sequências de instruções é representado na Figura 1.6 por linhas contínuas. Note que, nesse exemplo, a transição de um *gadget* para outro sempre ocorre ao executar uma instrução de retorno (RET). Essas instruções desviam o fluxo de execução (retornam) para o endereço posicionado no topo da pilha (indicados pelas linhas tracejadas). Como o conteúdo da pilha é inicialmente sobrescrito por dados introduzidos pelo usuário, dessa forma um atacante consegue controlar o encadeamento de *gadgets*, garantindo que computações arbitrárias possam ser executadas a seu critério.

1.3.2. Simulando o LBR

Last Branch Record (LBR) é um conjunto de registradores, presentes nos processadores modernos da Intel, que registram os últimos desvios de fluxo de execução tomados por um programa [Kleen b]. Esses dados de desvios recentes são importantes para diversas aplicações, como análise de desempenho e implementação de operações de memória transacional [Kleen a]. No contexto de segurança de software, pode ser usado para identificar situações onde uma determinada aplicação pode estar sendo alvo de um ataque ROP. Essa identificação se baseia no fato de que durante um ataque ROP, a correspondência ideal entre chamadas de procedimentos (um tipo particular de desvios) e instruções de retorno é geralmente quebrada. Em função dessas características de comportamento dinâmico inerentes a um ataque ROP, a utilização de uma ferramenta de IDB pode ser efetiva na identificação dessas tentativas de ataque. Esta seção detalha a implementação de uma *Pintool* que simula a estrutura do LBR de forma a utilizar a informação registrada para identificar possíveis ataques ROP, além de conseguir determinar informações úteis sobre a correspondência entre instruções de chamada de procedimento (CALLs) e instruções de retorno (RETs).

A ferramenta analisada será a **lbrmatch.cpp**. Primeiramente, o seu cabeçalho é definido como mostrado a seguir.

```
1 /**
2  * Autor: Hugo Sousa (hugosousa@dcc.ufmg.br)
3  *
4  * lbrmatch.cpp: Pintool usada para simular a estrutura de
```

```
5   * um LBR para instruções de chamada de função (CALL) e
6   * checar por correspondências entre elas e instruções de
7   * retorno.
8   */
9
10  #include "pin.H"
11
12  #include <iostream>
13  #include <fstream>
14
15  using namespace std;
16
17  KNOB<string> outFileKnob(KNOB_MODE_WRITEONCE, "pintool",
18    "o", "lbr_out.log", "Nome do arquivo de saída.");
19
20  KNOB<unsigned int> lbrSizeKnob(KNOB_MODE_WRITEONCE,
21    "pintool", "s", "16", "Número de entradas do LBR.");
```

Nas linhas 17 e 20 temos a definição de 2 parâmetros de linha de comando aceitos pela *Pintool*. O primeiro (*flag* **-o**) define o nome do arquivo de saída e o segundo (*flag* **-s**) define o número de entradas do LBR a ser simulado. De acordo com o manual da Intel, esse número varia nos valores de 4, 8 e 16 entradas [Guide 2011]. Quanto maior for esse número, maior será o número de chamadas de função aninhadas que o LBR será capaz de armazenar, como mostrado na parte seguinte do código.

```
23  /**
24   * Estrutura de dados LBR (Last Branch Record).
25   */
26
27  /**
28   * Uma entrada do LBR nessa Pintool será composta pelo endereço
29   * da instrução CALL e um booleano que indica se esse é um CALL
30   * direto (true) ou indireto (false).
31   */
32  typedef pair<ADDRINT, bool> LBREntry;
33
34  class LBR {
35  private:
36    LBREntry *buffer;
37    unsigned int head, tail, size;
38  public:
39    LBR(unsigned int size) {
40      this->size = size;
41      head = tail = 0;
42      buffer = (LBREntry*) malloc(sizeof(LBREntry) * (size + 1));
43    }
44
45    bool empty() {
46      return (head == tail);
```

```
47    }
48
49    void put(LBREntry item) {
50      buffer[head] = item;
51      head = (unsigned int) (head + 1) % size;
52
53      if (head == tail)
54        tail = (unsigned int) (tail + 1) % size;
55    }
56
57    void pop() {
58      if (empty())
59        return;
60
61      head = (unsigned int) (head - 1) % size;
62    }
63
64    LBREntry getLastEntry() {
65      if (empty())
66        return make_pair(0, false);
67
68      unsigned int index = (unsigned int) (head - 1) % size;
69
70      return buffer[index];
71    }
72  };
```

Cada entrada do LBR simulado nessa *Pintool* guarda dois valores: o endereço da instrução de chamada (CALL) de função e um valor que indica se essa instrução é um CALL direto ou indireto. Além disso, como o LBR tem um número limitado de entradas, caso ele esteja cheio no momento de uma nova adição (método **put**), a entrada mais antiga será sobrescrita. Prosseguindo, a *Pintool* define sua que a instrumentação será aplicada a cada traço da aplicação alvo e itera sobre os blocos básicos de cada um.

```
168     TRACE_AddInstrumentFunction(InstrumentCode, 0);
```

```
122 VOID InstrumentCode(TRACE trace, VOID *v) {
123   /**
124    * Função de instrumentação da Pintool.
125    *
126    * Cada bloco básico tem um único ponto de entrada
127    * e um único ponto de saída. Assim, CALLs e RETs
128    * somente podem ser encontradas ao fim de blocos
129    * básicos.
130    */
131
132   for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl =
        BBL_Next(bbl)) {
133
134     INS tail = BBL_InsTail(bbl);
```

```
135
136     if (INS_IsRet(tail)) {
137       INS_InsertCall(tail, IPOINT_BEFORE, (AFUNPTR) doRET,
138         IARG_BRANCH_TARGET_ADDR, IARG_END);
139     } else if (INS_IsCall(tail)) {
140       INS_InsertCall(tail, IPOINT_BEFORE, (AFUNPTR) doCALL,
141         IARG_INST_PTR, IARG_END);
142     }
143   }
144 }
```

Aqui, o código de análise é diferente para CALLs e RETs. Para os primeiros, a função de análise é **doCALL**, que recebe o endereço da instrução como parâmetro. Para os segundos, a função de análise é **doRET**, que recebe o endereço de retorno da instrução como parâmetro. Como mostrado no código seguir, a função **doCALL** simplesmente adiciona entradas ao LBR, enquanto **doRET** verifica se o endereço de retorno da instrução corresponde ao endereço imediatamente posterior ao endereço da instrução CALL no topo do LBR. Caso essa correspondência exista, a aplicação alvo está sob funcionamento regular e, caso contrário, pode estar sob um ataque ROP. É importante notar que nesse código de exemplo, a distinção entre CALLs diretos e indiretos não é explorada, entretanto, isso pode ser feito caso seja de interesse do programador, através da estrutura das entradas do LBR.

```
84  VOID doRET(ADDRINT returnAddr) {
85    /**
86     * Função de análise para instruções de retorno.
87     *
88     * @returnAddr: Endereço de retorno.
89     */
90
91    LBREntry lastEntry;
92
93    /**
94     * Instrução CALL anterior ao endereço de retorno
95     * pode estar de 2 a 7 bytes antes dele.
96     *
97     * callLBR é um objeto da classe LBR.
98     */
99    lastEntry = callLBR.getLastEntry();
100   for (int i = 2; i <= 7; i++) {
101     ADDRINT candidate = returnAddr - i;
102
103     if (candidate == lastEntry.first) {
104       calllLBRMatches++;
105       break;
106     }
107   }
108
109   callLBR.pop();
```

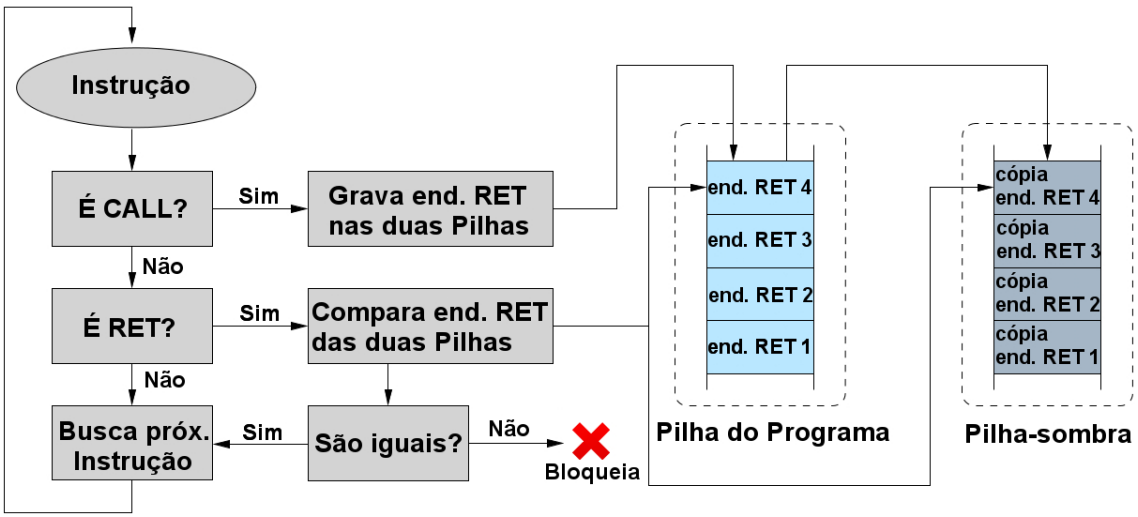


Figura 1.7. Proteção baseada em pilha-sombra [Davi et al. 2011].

```
110 }
111
112 VOID doCALL(ADDRINT addr) {
113   /**
114    * Função de análise para instruções de chamada de função.
115    *
116    * @addr: O endereço da instrução.
117    */
118
119   callLBR.put(make_pair(addr, true));
120 }
```

1.3.3. Pilha-Sombra

Pilha-Sombra é a tradução da expressão *Shadow-Stack*, do inglês, e refere-se a um conceito amplamente utilizado na área de segurança de software. Esse conceito estabelece que é possível impedir diversos ataques de corrupção da memória através da criação de uma segunda pilha de execução, protegida, onde são armazenadas cópias dos endereços de retorno de procedimentos. Dessa forma, pode-se comparar tais endereços com aqueles que a aplicação efetivamente utiliza para desviar o fluxo de execução, evitando que corrupções maliciosas do conteúdo da pilha acarretem no sequestro do fluxo de execução de um programa [Vendicator 2000]. Conforme ilustrado na Figura 1.7, sempre que uma instrução de chamada de uma função (CALL) é executada, além de ser anotado na tradicional pilha do processo, o endereço de retorno é também escrito em uma estrutura de pilha adicional (a pilha-sombra), protegida contra a escrita por instruções do processo. Essa proteção é necessária para evitar que um atacante altere os endereços de retorno nas duas pilhas, fazendo-os coincidir e, consequentemente, superando a proteção. No momento em que instruções de retorno são executadas (RET), checa-se a coincidência dos endereços entra as duas pilhas. Assim, se o endereço de retorno armazenado na pilha do processo for alterado, o ataque será identificado.

Diversas implementações desse conceito, inclusive utilizando IDB, já foram propostas na literatura, com o objetivo de evitar ataques como estouro de memória na pilha (Seção 1.3.1.1) e ROP (Seção 1.3.1.2). A seguir, apresentamos o código-fonte de uma implementação do conceito de Pilha-Sombra elaborada pelos autores deste curso no *framework* de instrumentação dinâmica Pin.

```
1  #include "pin.H"           // para usar APIs do Pin
2  #include <stack>           // para usar estrutura de dados Pilha
3  #include <stdio.h>         // para usar "fprintf" e "snprintf"
4  #include <stdlib.h>        // para usar "calloc"
5  #include <string.h>        // para usar "memset" e converter nú
       meros para string
6  #include <sstream>         // para converter números para string
7  #include <fstream>         // para imprimir no arquivo de saída
8  #include <sys/time.h>      // para registro do tempo de
       processador usado pelo algoritmo
9  #include <sys/resource.h> // para registro do tempo de
       processador usado pelo algoritmo
10
11 static TLS_KEY chave_tls;           // chave para acesso ao
       armazenamento local (TLS) das threads
12 static std::ofstream arquivo_saida; // arquivo onde a saída é
       escrita
13
14 void Uso(){
15    fprintf(stderr, "\nUso: pin -t <Pintool> [-o <
          NomeArquivoSaida>] [-logfile <NomeLogDepuracao>] -- <
          Programa alvo>\n\n"
16                    "Opções:\n"
17                    "  -o       <NomeArquivoSaida>\t"
18                    "Indica o nome do arquivo de saida (padrão:
                         $PASTA_CORRENTE/pintool.out)\n"
19                    "  -logfile <NomeLogDepuracao>\t"
20                    "Indica o nome do arquivo de log de depuracao
                          (padrão: $PASTA_CORRENTE/pintool.log)\n\n
                         ");
21 }
22
23 static string converte_double_string(double valor){
24    ostringstream oss;
25    oss << valor;
26    return(oss.str());
27 }
28
29 void IniciaThread(THREADID tid, CONTEXT * contexto, int flags,
       void * v){
30    stack<ADDRINT> *pilhaSombra = new stack<ADDRINT>();
31    PIN_SetThreadData(chave_tls, pilhaSombra, tid);
32 }
```

As linhas 1 a 9 importam as bibliotecas do Pin e de C++ necessárias. Em seguida, as linhas 11 e 12 criam duas variáveis globais. A primeira (chave_tls), é utilizada para diferenciar pilhas-sombras pertencentes a *threads* distintas em aplicações *multi-thread*. A segunda (arquivo_saida), referencia o arquivo onde os resultados são escritos. A função **Uso** simplesmente imprime uma mensagem no *prompt* de comandos indicando as opções de uso da Pintool. Por sua vez, a função **converte_double_string** faz o que seu nome sugere: recebe um valor do tipo *double* como parâmetro, converte-o para o tipo *string*, e devolve o *string* convertido.

Uma vez que cada *thread* deve possuir sua própria pilha-sombra, a função **IniciaThread** é usada para instanciar um objeto do tipo **stack** no TLS (*Thread Local Storage*). O TLS é um espaço de armazenamento criado pela API do Pin que oferece a opção de reservar e gerenciar, de forma eficiente, áreas de memória exclusivas para *threads*. Qualquer *thread* de um processo pode guardar e recuperar dados no seu próprio espaço, referenciando-o através de uma chave única (vide linha 11). Como o Pin cria um identificador (ID) único para cada thread, normalmente utiliza-se o próprio ID de cada thread como sua chave de acesso ao TLS. Como veremos adiante no código, a função **IniciaThread** é posteriormente registrada para ser executada sempre que uma nova *thread* for criada. A função **Fim** (linhas 34 a 47), é chamada quando a aplicação instrumentada termina de executar. Neste exemplo, ela calcula o tempo de CPU (usuário + sistema) consumido pelo processo e imprime os resultados no arquivo de saída.

Conforme ilustrado na Figura 1.7, o mecanismo de pilha-sombra exige duas funções de análise: uma para tratar as instruções de chamada de função (CALL) e outra para tratar as instruções de retorno de chamadas de funções (RET). Em nosso código, a função **AnaliseCALL** é registrada junto ao Pin para executar sempre que uma instrução CALL for executada. Essa função grava o endereço de retorno recebido como parâmetro na pilha-sombra da *thread* cujo ID também é recebido como parâmetro.

```
34 void Fim(INT32 codigo, void *v){
35    // salva instante atual para registrar o momento de término
36    time_t data_hora = time(0);
37
38    // calcula consumo total de tempo de CPU pelo processo (usuá
         rio + sistema)
39    struct rusage ru;
40    getrusage(RUSAGE_SELF, &ru);
41    double tempo_fim = static_cast<double>(ru.ru_utime.tv_sec) +
         static_cast<double>(ru.ru_utime.tv_usec * 0.000001) +
42                       static_cast<double>(ru.ru_stime.tv_sec) +
                            static_cast<double>(ru.ru_stime.tv_usec
                             * 0.000001);
43
44    // imprime no arquivo de saída os resultados
45    arquivo_saida << " #### Instrumentação finalizada em " <<
         converte_double_string(tempo_fim) << " segundos" << endl;
46    arquivo_saida << " #### Fim: " << string(ctime(&data_hora))
         << endl;
47 }
```

```
48
49 void PIN_FAST_ANALYSIS_CALL AnaliseCALL(THREADID tid, ADDRINT
       endereco){
50    // obtém ponteiro para a pilha sombra
51    stack<ADDRINT> *pilhaSombra = static_cast<stack<ADDRINT> *>(
          PIN_GetThreadData(chave_tls, tid));
52    // empilha o endereço de retorno na pilha sombra da thread
53    pilhaSombra->push(endereco);
54 }
```

Nas linhas 56 a 102, aparece a função **AnaliseRET**, registrada junto ao Pin para executar sempre que uma instrução RET for executada. É ela que checa se o endereço de retorno corresponde ao endereço anotado no topo da pilha-sombra. Note que nas linhas 80, 85, 95 e 100 são utilizadas travas de sincronização de *threads* do próprio Pin, para evitar que *threads* concorrentes escrevam simultaneamente no arquivo de saída.

```
56 void PIN_FAST_ANALYSIS_CALL AnaliseRET(THREADID tid, CONTEXT *
       contexto){
57    // inicializa endereço de retorno anotado na pilha original
          da thread
58    ADDRINT end_ret_original = 0;
59
60    // inicializa endereço de retorno anotado na pilha sombra
61    ADDRINT end_ret_sombra = 0;
62
63    // recupera endereço do topo da pilha original
64    ADDRINT * ptr_topo_pilha = (ADDRINT *) PIN_GetContextReg(
          contexto, REG_STACK_PTR);
65
66    // copia conteúdo do topo da pilha (endereço de retorno) para
          a variável inicializada
67    PIN_SafeCopy(&end_ret_original, ptr_topo_pilha, sizeof(
          ADDRINT));
68
69    // obtém ponteiro para a pilha sombra
70    stack<ADDRINT> *pilhaSombra = static_cast<stack<ADDRINT> *>(
          PIN_GetThreadData(chave_tls, tid));
71
72    // checa se há algum endereço anotado na pilha sombra
73    if(pilhaSombra->size() != 0){
74       // obtém endereço de retorno anotado no topo da pilha
             sombra
75       end_ret_sombra = pilhaSombra->top();
76       // se os endereços de retorno não coincidirem, sinaliza a
             suspeita de ataque ROP
77       if(end_ret_sombra != end_ret_original){
78
79          // Ativa uma trava interna do Pin para evitar que
              threads concorrentes escrevam simultaneamente no LOG
```

```
80            PIN_LockClient();
81
82            arquivo_saida << " #### Suspeita de ataque ROP! O
                 endereço de retorno " << hexstr(end_ret_original,
                 sizeof(ADDRINT)) << " não coincide com o endereço
                 anotado na pilha sombra (" << hexstr(end_ret_sombra,
                  sizeof(ADDRINT)) << ")" << endl;
83
84            // Libera trava
85            PIN_UnlockClient();
86         }
87         // desempilha o endereço anotado no topo da pilha sombra
88         pilhaSombra->pop();
89      }
90      else{
91         /* se uma instrução RET está sendo executada e não há
               endereço de retorno na pilha sombra,
92            significa que a paridade CALL-RET foi violada */
93
94         // Ativa uma trava interna do Pin para evitar que threads
               concorrentes escrevam simultaneamente no LOG
95         PIN_LockClient();
96
97         arquivo_saida << " #### Suspeita de ataque ROP! Não há
               nenhum endereço de retorno anotado na pilha sombra e o
               programa pretende retornar para o endereço de retorno "
                << hexstr(end_ret_original, sizeof(ADDRINT)) << endl;
98
99         // Libera trava
100        PIN_UnlockClient();
101     }
102  }
```

Em seguida, apresentamos a função **InstrumentaCodigo** (linhas 104 a 127), que na função **main** (linhas 129 a 165) é registrada junto ao Pin para executar a instrumentação do código. Ela registra junto ao Pin as funções **AnaliseCALL** e **AnaliseRET**, já apresentadas. Para isso, usa o conceito de BBLs (*Basic Blocks*), evitando a instrumentação de todas as instruções executadas. Ao invés disso, por questões de eficiência, checa apenas a última instrução de cada BBL, já que todo BBL possui um único ponto de saída. Note que utilizamos a opção **IPOINT_ANYWHERE** ao registrar a função **AnaliseCALL**. Essa opção permite que o Pin agende a chamada da função de análise em qualquer lugar do BBL, para obter um melhor desempenho. No caso do registro da função **AnaliseRET**, a opção **IPOINT_ANYWHERE** não pôde ser usada pois, entre as diversas instruções de um BBL, o topo da pilha pode variar em relação àquele válido no momento em que a instrução de retorno (RET) for executar. Também por questões de desempenho (passagem de argumentos otimizada pelo Pin), a opção **IARG_FAST_ANALYSIS_CALL** é utilizada em ambos os casos.

```
104 void InstrumentaCodigo(TRACE trace, void *v){
```

```
105     // percorre todos os BBLs
106     for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl =
            BBL_Next(bbl)){
107
108         // obtém a última instrução do BBL
109         INS ins = BBL_InsTail(bbl);
110
111         // se a última instrução do BBL for uma instrução CALL
112         if( INS_IsCall(ins) ){
113             // Registra a função "AnaliseCALL" para ser chamada
                    quando o BBL executar,
114             // passando o ID da thread e o endereço de retorno a
                    ser empilhado.
115             BBL_InsertCall(bbl, IPOINT_ANYWHERE, (AFUNPTR)
                    AnaliseCALL, IARG_FAST_ANALYSIS_CALL, IARG_THREAD_ID
                    , IARG_ADDRINT, INS_Address(ins) + INS_Size(ins),
                    IARG_END);
116         }
117         else{
118             // se a última instrução do BBL for uma instrução RET
119             if(INS_IsRet(ins)){
120                 // registra a função "AnaliseRET" para ser chamada
                        imediatamente antes de uma instrução RET executar
                        ,
121                 // passando o ID da thread e o ponteiro para o
                        contexto de execução (Pilha, regs, etc).
122                 INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)
                        AnaliseRET, IARG_FAST_ANALYSIS_CALL,
                        IARG_THREAD_ID, IARG_CONTEXT, IARG_END);
123             }
124         }
125     }
126 }
```

Finalmente, apresentamos a função **main** que, neste exemplo, tem a função de tratar opções da linha de comandos e arquivos de saída, inicializar o Pin e a aplicação a ser monitorada, instanciar o TLS e registrar as funções apresentadas anteriormente (**Fim**, **IniciaThread** e **InstrumentaCodigo**) junto ao Pin.

```
129 int main(int argc, char *argv[]){
130
131     // Usado para receber da linha de comandos (opção -o) o nome
            do arquivo de saída. Se não for especificado, usa-se o
            nome "Pintool.out"
132     KNOB<string> KnobArquivoSaida(KNOB_MODE_WRITEONCE, "pintool",
            "o", "pintool.out", "Nome do arquivo de saida");
133
134     // Inicializa o Pin e checa os parâmetros
135     if(PIN_Init(argc, argv)){
```

```
136         // imprime mensagem indicando o formato correto dos parâ
                metros e encerra
137         Uso();
138         return(1);
139     }
140 
141     // Abre o arquivo de saída no modo apêndice. Se não for
            passado um nome para o arquivo na linha de comandos, usa "
            Pintool.out"
142     arquivo_saida.open(KnobArquivoSaida.Value().c_str(), std::
            ofstream::out | std::ofstream::app);
143 
144     // obtem e imprime no arquivo de saída o momento em que a
            execução está iniciando
145     time_t data_hora = time(0);
146     arquivo_saida << endl << " #### Inicio: " << string(ctime(&
            data_hora));
147 
148     // obtém a chave para acesso à área de armazenamento local
            das threads (TLS)
149     chave_tls = PIN_CreateThreadDataKey(0);
150 
151     // registra a função "Fim" para ser executada quando a aplica
            ção for terminar
152     PIN_AddFiniFunction(Fim, NULL);
153 
154     // registra a função "IniciaThread" para ser executada quando
             uma nova thread for iniciar
155     PIN_AddThreadStartFunction(IniciaThread, NULL);
156 
157     // registra a função "InstrumentaCodigo" para instrumentar os
             "traces"
158     TRACE_AddInstrumentFunction(InstrumentaCodigo, NULL);
159 
160     // inicia a execução do programa a ser instrumentado e só
            retorna quando ele terminar
161     PIN_StartProgram();
162 
163     // encerra a execução do Pin
164     return(0);
165 }
```

1.3.4. Janela Deslizante

Janela deslizante é uma estratégia de proteção contra ataques ROP criada por um dos autores deste curso. Essa estratégia foi implementada em um protótipo denominado RipRop utilizando-se o instrumentador Pin [Tymburibá et al. 2015]. Nesta seção, são detalhados os procedimentos de implementação adotados, incluindo as abordagens de otimização de código empregadas com o intuito de minimizar o *overhead* do protótipo. Antes disso,

porém, explicamos os conceitos relacionados à solução de janela deslizante.

Normalmente, as sequências de instruções que compõem cada *gadget* usado em um ataque ROP são extremamente curtas, dificilmente contendo mais do que cinco instruções. Essa é uma característica inerente aos ataques ROP, porque quanto maior a sequência de instruções, maior a probabilidade de existir entre essas instruções uma operação que altere o estado da memória ou de um registrador de forma a comprometer o ataque. Essa alteração de estado é comumente chamada por atacantes de "efeito colateral" de um *gadget*. A fim de evitar esses efeitos colaterais, quase sempre os *gadgets* escolhidos pelos atacantes são extremamente curtos.

Por evitar os mencionados "efeitos colaterais", ataques ROP acabam apresentando uma elevada concentração de instruções de desvio indireto em um curto espaço de tempo. Diante dessa constatação, diversos autores investiram esforços em uma estratégia de controle da frequência de instruções de desvio indireto como forma de detectar a execução de cadeias de *gadgets* [Chen et al. 2009, Davi et al. 2009, Min et al. 2013, Tymburibá et al. 2014]. Uma dessas estratégias é a janela deslizante, ilustrada na Figura 1.8, que consiste em checar se a contagem do número de instruções de desvio indireto em uma determinada "janela de instruções"é maior do que um determinado limiar [Tymburibá et al. 2014]. Para definir o valor ideal desse limiar, é possível tanto estabelecer um valor padrão, com base na análise de um conjunto de aplicações, quanto efetuar uma etapa de análise estática do código com cada software que se pretende proteger, a fim de estabelecer o limiar máximo atingido por aquela aplicação [Emílio et al. 2015].

A lógica de funcionamento da janela deslizante segue o esquema ilustrado na Figura 1.9. Assumindo-se que a janela possui um tamanho N, pode-se dizer que a função da janela é permitir a contagem do número de desvios indiretos executados nas últimas N instruções. Nessa janela, as posições correspondentes às instruções de desvio indireto são anotadas com um bit 1 e as demais instruções são representadas pelo bit 0. Ao executar qualquer instrução, a janela precisa ser atualizada. Para evitar um *overhead* excessivo decorrente da análise de todas as instruções de um programa, novamente exploramos o conceito de bloco básico (*Basic Block*, ou BBL). Assim, insere-se um código de análise para um BBL, ao invés de avaliar cada instrução do programa, tornando a instrumentação mais eficiente. No esquema proposto, utiliza-se uma API do Pin (**BBL_InsTail**) para buscar a última instrução do BBL que está sendo instrumentado. Como, por definição, um BBL é um bloco de instruções com um único ponto de entrada e um único ponto de saída, sabe-se que a única instrução desse bloco que eventualmente poderá corresponder a um desvio indireto será a última instrução do bloco.

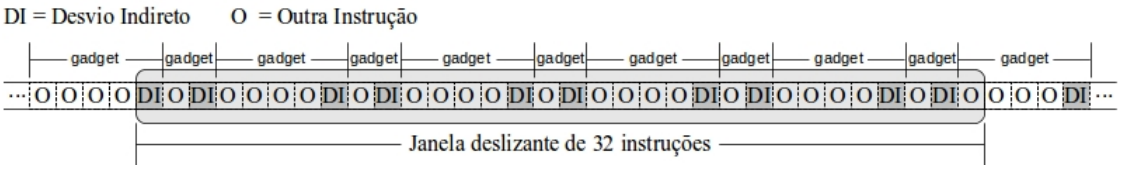


Figura 1.8. Funcionamento de uma janela deslizante durante a execução de *gadgets* ROP [Tymburibá et al. 2016].

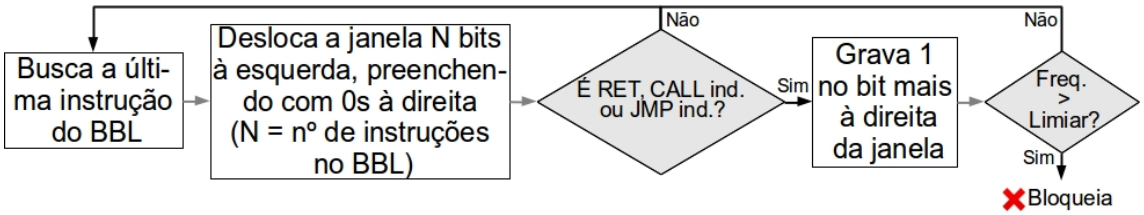


Figura 1.9. Lógica de funcionamento de uma janela deslizante para contagem de desvios indiretos executados [Tymburibá et al. 2014].

Depois de capturar a última instrução do BBL, a Pintool desloca a janela de instruções à esquerda, preenchendo os bits deslocados à direita com o bit zero (0). O número de bits deslocados corresponde à quantidade de instruções existentes no BBL em análise. Essa operação de deslocamento da janela obrigatoriamente deve ser realizada para todos os BBLs executados pela aplicação, independente de eles possuírem alguma instrução de desvio indireto ou já terem sido executados anteriormente. Após deslocar a janela de instruções, checa-se a última instrução do BBL. Caso ela não corresponda a um desvio indireto, nada mais é preciso ser feito e a execução da aplicação prossegue até que um novo BBL seja buscado. Por outro lado, se a última instrução do BBL corresponder a um desvio indireto, a Pintool grava o valor um (1) no bit mais à direita da janela e calcula a quantidade de desvios indiretos registrados na janela. Caso o valor calculado ultrapasse o limiar estabelecido para a aplicação, a Pintool sinaliza em um arquivo de saída a ocorrência de um ataque ROP e encerra a execução da aplicação.

A seguir, antes de apresentar o código que implementa a janela deslizante no Pin, discutimos brevemente algumas decisões de implementação tomadas com o objetivo de otimizar a execução da Pintool, para garantir o melhor tempo de execução possível.

A instrução POPCNT. Considerando-se que na arquitetura x86 existem instruções de hardware que permitem deslocar os bits de um registrador e contar a quantidades de bits ativos [Intel 2016], a execução das operações de atualização das janelas não acarretou em um *overhead* tão elevado quanto se fossem utilizados laços de repetição. A instrução SHL (deslocamento lógico para a esquerda, em direção aos bits mais significativos), por exemplo, usa um operando para indicar a quantidades de bits a serem deslocados de um registrador. Os bits menos significativos deslocados pela instrução são preenchidos com o valor zero (0). Isso evita a necessidade de executar um laço iterativo para deslocar a estrutura que representa a janela de instruções, o que exigiria um esforço computacional maior. Para contar a quantidade de bits ativos em um operando, foi utilizada a instrução POPCNT (Contagem de população). Trata-se de uma instrução introduzida em 2007 pela AMD em sua microarquitetura denominada Barcelona [AMD 2017]. Os processadores da família Intel Core introduziram a instrução POPCNT junto com a extensão do conjunto de instruções SSE4.2 [Intel 2018a]. Desde então, os processadores produzidos por esses e outros fabricantes contam com uma instrução equivalente. Assim como no deslocamento da janela de instruções, o uso dessa instrução de hardware evita o elevado custo computacional de percorrer a janela para contar os bits ativos, garantindo uma considerável melhoria de desempenho.

Linhas de Cache da CPU. Uma questão estrutural inerente à arquitetura dos processadores que costuma ocasionar a degradação do desempenho de aplicações que executam várias *threads* é o problema do falso compartilhamento (*false sharing*). Essa situação ocorre quando múltiplas *threads* acessam diferentes partes de uma mesma linha de cache da CPU e ao menos um desses acessos é uma escrita [Intel 2018b]. Para manter a coerência dos dados em memória, o computador copia os dados da cache de uma CPU para a outra, mesmo que as *threads* concorrentes não estejam efetivamente compartilhando nenhum dado. Normalmente, problemas de falso compartilhamento entre *threads* podem ser evitados ajustando-se o tamanho das estruturas de dados para que cada estrutura ocupe uma linha de cache diferente. Nossa implementação da janela deslizante trata aplicações que lançam múltiplas *threads* através da criação de uma janela de instruções independente para cada *thread*, assim como realizado com a pilha-sombra, na Seção 1.3.3. No entanto, isso não impede que mais de uma janela ocupe a mesma linha de cache da CPU, o que pode eventualmente acarretar no problema de falso compartilhamento. Para evitar essa situação indesejada, forçamos a alocação de uma estrutura de dados inútil junto com a estrutura que representa a janela de instruções. O tamanho dessa estrutura de dados inerte é calculado para que a soma da área de armazenamento ocupada pela janela com o espaço ocupado por essa estrutura inútil corresponda ao tamanho exato de uma linha de cache da CPU. Assim, a Pintool força o armazenamento dos dados de cada *thread* em uma linha de cache diferente, evitando o problema de falso compartilhamento. Para garantir o correto funcionamento desse mecanismo de prevenção do problema de falso compartilhamento, deve-se registrar no código-fonte da Pintool o tamanho do complemento da linha de cache (linha 11 no código abaixo). Em ambientes Linux, uma forma de identificar o tamanho da linha de cache (em bytes) usada pelo processador do equipamento onde se pretende usar a Pintool é executar o comando *getconf LEVEL1_DCACHE_LINESIZE*.

O código da Pintool que implementa a janela deslizante está listado abaixo. Assim como no exemplo da pilha-sombra, o código começa com a importação de bibliotecas e a declaração de variáveis globais (linhas 1 a 16). Em seguida (linhas 18 a 21), a estrutura usada para representar a janela de instruções das *threads* é declarada, com capacidade de 32 bits (1 bit por instrução). Note que o arranjo **lixo** é criado com o tamanho exato para forçar a janela de cada *thread* a ocupar sua própria linha no cache da CPU, a fim de evitar perdas de desempenho decorrentes do problema de *false sharing*.

```
1  #include "pin.H"           // para usar APIs do Pin
2  #include <stdio.h>         // para usar I/O
3  #include <stdlib.h>        // para usar exit()
4  #include <string.h>        // para converter números para string
5  #include <sstream>         // para converter números para string
6  #include <fstream>         // para imprimir no arquivo de saída
7  #include <sys/time.h>      // para registro do tempo de
       processador usado pelo algoritmo
8  #include <sys/resource.h> // para registro do tempo de
       processador usado pelo algoritmo
9
10 static const UINT32 tam_janela = 32;              // constante
       que indica o tamanho da janela em bits
```

```
11 static const UINT32 COMPLEMENTO_LINHA_CACHE = 60; // tamanho da
      linha da cache (64 bytes) - tamanho da janela
12 static const UINT32 limiar_padrao = 10;           // valor de
      limiar padrao pré-estabelecido para a janela de 32
13 static const UINT32 MASCARA_UM = 1;               // máscara
      usada para setar o bit menos significativo da janela
14 static std::ofstream arquivo_saida;               // arquivo
      onde a saída é escrita
15 static UINT32 limiar;                             // valor de
      limiar checado durante a execução
16 static TLS_KEY chave_tls;                         // chave para
      acesso ao armazenamento local (TLS) das threads
17
18 struct JanelaThread{
19    UINT32 janela_bits; // buffer que guarda os bits (janela)
20    UINT8 lixo[COMPLEMENTO_LINHA_CACHE]; // área inútil usada
         para ocupar uma linha inteira da cache
21 };
```

Assim como no exemplo da pilha-sombra, algumas funções existem para exibir uma mensagem sobre o uso correto dos parâmetros de linha de comando e para converter tipos de valores (linhas 23 a 43). A função **IniciaThread** (linhas 45 a 54), chamada ao iniciar uma nova thread, aloca espaço para a janela da nova *thread* no TLS.

```
23 void Uso(){
24    fprintf(stderr, "\nUso: pin -t <Pintool> [-l <Limiar>] [-o <
         NomeArquivoSaida>] [-logfile <NomeLogDepuracao>] -- <
         Programa alvo>\n\n"
25                    "Opções:\n"
26                    "  -l       <Limiar>\t"
27                    "  -o       <NomeArquivoSaida>\t"
28                    "Indica o nome do arquivo de saida (padrão:
                         $PASTA_CORRENTE/pintool.out)\n"
29                    "  -logfile <NomeLogDepuracao>\t"
30                    "Indica o nome do arquivo de log de depuracao
                          (padrão: $PASTA_CORRENTE/pintool.log)\n\n
                         ");
31 }
32
33 static string converte_double_string(double valor){
34    ostringstream oss;
35    oss << valor;
36    return(oss.str());
37 }
38
39 static string converte_ulong_string(unsigned long int valor){
40    ostringstream oss;
41    oss << valor;
42    return(oss.str());
```

```
43 }
44
45 void IniciaThread(THREADID thread_id, CONTEXT *
      contexto_registradores, int flags_SO, void *v){
46    // Aloca espaço para a janela e guarda endereço em apontador
47    JanelaThread* janela_ptr = new JanelaThread;
48
49    // Inicializa a janela
50    janela_ptr->janela_bits = 0x00000000;
51
52    // Armazena a janela na área de armazenamento (TLS) da thread
53    PIN_SetThreadData(chave_tls, janela_ptr, thread_id);
54 }
```

A função **Fim** (linhas 56 a 69) tem o mesmo propósito da função de mesmo nome no exemplo anterior (Seção 1.3.3). Já a função **DeslocaJanela** (linhas 71 a 108) corresponde ao código de análise. Nessa rotina, a janela deslizante é deslocada, de acordo com os parâmetros recebidos, e um alerta é emitido em caso de superação do limiar de desvios indiretos. O primeiro parâmetro recebido por essa função, **thread_id**, é usado para recuperar a janela da *thread*. O segundo parâmetro, **num_bits_shift**, indica o número de instruções executadas no BBL. Esse valor corresponde ao número de bits que devem ser deslocados na janela de instruções. O terceiro parâmetro, **desvio_indireto**, indica se a última instrução do BBL é um desvio indireto (TRUE) ou não (FALSE). Se a última instrução do BBL for um desvio indireto, o bit menos significativo da janela é ligado.

```
56 void Fim(INT32 codigo, void *v){
57    // salva instante atual para registrar o momento de término
58    time_t data_hora = time(0);
59
60    // calcula consumo total de tempo de CPU pelo processo (usuá
         rio + sistema)
61    struct rusage ru;
62    getrusage(RUSAGE_SELF, &ru);
63    double tempo_fim = static_cast<double>(ru.ru_utime.tv_sec) +
         static_cast<double>(ru.ru_utime.tv_usec * 0.000001) +
64                       static_cast<double>(ru.ru_stime.tv_sec) +
                            static_cast<double>(ru.ru_stime.tv_usec
                             * 0.000001);
65
66    // imprime no arquivo de saída os resultados
67    arquivo_saida << " #### Fim: " << string(ctime(&data_hora));
68    arquivo_saida << " #### Instrumentação finalizada em " <<
         converte_double_string(tempo_fim) << " segundos" << endl
         << endl;
69 }
70
71 void PIN_FAST_ANALYSIS_CALL DeslocaJanela(THREADID thread_id,
      UINT32 num_bits_shift, BOOL desvio_indireto){
72    // variável auxiliar: número de bits setados na janela
```

```
73      UINT32 num_bits_setados;
74
75      // obtém ponteiro para a janela da thread
76      JanelaThread *janela_ptr = static_cast<JanelaThread *>(
            PIN_GetThreadData(chave_tls, thread_id));
77
78      // se o número de bits a deslocar é menor que 32 e maior que
            0
79      if(num_bits_shift < tam_janela){
80          // executa shift de num_bits_shift
81          janela_ptr->janela_bits <<= num_bits_shift;
82      }
83      else{ // se o número de bits a deslocar é maior ou igual ao
            tamanho da janela
84          // zera o buffer
85          janela_ptr->janela_bits = 0x00000000;
86      }
87
88      // seta o bit menos signif. da janela se a última instrução
            do BBL for um desvio indireto
89      if(desvio_indireto){
90          janela_ptr->janela_bits |= MASCARA_UM;
91      }
92
93      // usa a instrução de HW POPCNT para contar o número de bits
            setados na janela
94      num_bits_setados = __builtin_popcount(janela_ptr->janela_bits
            );
95
96      // se o número de bits setados for maior do que o limiar
97      if(num_bits_setados > limiar){
98          // Ativa uma trava interna do Pin para evitar que threads
                concorrentes escrevam simultaneamente no arquivo de saí
                da
99          PIN_LockClient();
100
101         // imprime mensagem no arquivo de saída
102         arquivo_saida << " ####  Suspeita de ataque ROP! O limiar
                de " << converte_ulong_string(static_cast<unsigned long
                 int>(limiar)) <<
103                          " foi superado pelo seguinte valor: " <<
                             converte_ulong_string(static_cast<
                             unsigned long int>(num_bits_setados))
                             << endl;
104
105         // Libera trava
106         PIN_UnlockClient();
107     }
108 }
```

A função **InstrumentaCodigo** (linhas 110 a 121) é registrada junto ao Pin para executar a instrumentação do código. Ela registra a função **DeslocaJanela** junto ao Pin para ser disparada sempre que a última instrução de um BBL estiver para ser executada.

```
110 void InstrumentaCodigo(TRACE trace, void *v){
111    // percorre todos os BBLs
112    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl =
          BBL_Next(bbl)){
113       // se a última instrução do BBL for um desvio indireto
114       if( INS_IsIndirectBranchOrCall(BBL_InsTail(bbl)) ){
115          BBL_InsertCall(bbl, IPOINT_ANYWHERE, (AFUNPTR)
                DeslocaJanela, IARG_FAST_ANALYSIS_CALL,
                IARG_THREAD_ID, IARG_UINT32, BBL_NumIns(bbl),
                IARG_BOOL, TRUE, IARG_END);
116       }
117       else{// se a última instrução do BBL NÃO for um desvio
             indireto
118          BBL_InsertCall(bbl, IPOINT_ANYWHERE, (AFUNPTR)
                DeslocaJanela,  IARG_FAST_ANALYSIS_CALL,
                IARG_THREAD_ID, IARG_UINT32, BBL_NumIns(bbl),
                IARG_BOOL, FALSE, IARG_END);
119       }
120    }
121 }
```

Finalmente, apresentamos a função **main** (linhas 123 a 162), que executa um papel muito semelhante à função de mesmo nome do exemplo anterior (Seção 1.3.3).

```
123 int main(int argc, char *argv[]){
124    // Usado para receber da linha de comandos (opção -o) o nome
          do arquivo de saída. Se não for especificado, usa-se o
          nome "Pintool.out"
125    KNOB<string> KnobArquivoSaida(KNOB_MODE_WRITEONCE, "pintool",
           "o", "pintool.out", "Nome do arquivo de saida");
126
127    // Usado para receber da linha de comandos (opção -l) o valor
           de limiar a ser usado. Se não for especificado, usa-se o
          limiar padrão
128    KNOB<UINT32> KnobEntradaLimiar(KNOB_MODE_WRITEONCE, "pintool"
          , "l", converte_ulong_string(static_cast<unsigned long int
          >(limiar_padrao)), "Valor de limiar a ser usado pela
          protecao");
129
130    // Inicializa o Pin e checa os parâmetros
131    if(PIN_Init(argc, argv)){
132       // imprime mensagem indicando o formato correto dos parâ
             metros e encerra
133       Uso();
134       return(1);
135    }
```

```
136
137     // Abre o arquivo de saída no modo apêndice. Se não for
            passado um nome para o arquivo na linha de comandos, usa "
            Pintool.out"
138     arquivo_saida.open(KnobArquivoSaida.Value().c_str(), std::
            ofstream::out | std::ofstream::app);
139
140     // obtem e imprime no arquivo de saída o momento em que a
            execução está iniciando
141     time_t data_hora = time(0);
142     arquivo_saida << endl << " #### Inicio: " << string(ctime(&
            data_hora));
143
144     // Obtém valor do limiar. Se não for passado na linha de
            comandos, usa limiar padrão
145     limiar = KnobEntradaLimiar.Value();
146     arquivo_saida << " #### Valor do limiar: " <<
            converte_ulong_string(static_cast<unsigned long int>(
            limiar)) << endl;
147
148     // registra a função "Fim" para ser executada quando a aplica
            ção for terminar
149     PIN_AddFiniFunction(Fim, NULL);
150
151     // registra a função "IniciaThread" para ser executada quando
             uma nova thread for iniciar
152     PIN_AddThreadStartFunction(IniciaThread, NULL);
153
154     // registra a função "InstrumentaCodigo" para instrumentar os
             "traces"
155     TRACE_AddInstrumentFunction(InstrumentaCodigo, NULL);
156
157     // inicia a execução do programa a ser instrumentado e só
            retorna quando ele terminar
158     PIN_StartProgram();
159
160     // encerra a execução do Pin
161     return(0);
162 }
```

1.4. Considerações finais

Neste trabalho, a técnica de Instrumentação Dinâmica de Binários e o *framework* Pin foram apresentados. Com o material aqui reunido, o leitor tem o arcabouço necessário para escrever suas próprias ferramentas de IDB. O *framework* Pin mostra-se uma boa opção para o programador de ferramentas de IDB pois, além dos motivos apresentados na Seção 1.1.2.3, ele possui uma grande comunidade online que mantém-se ativa. Diante das capacidades do *framework*, espera-se que ele continue a ser muito utilizado na academia

ou até mesmo que seja ainda mais usado. Isso porque, se as tecnologias emergentes tendem a ser mais complexas, o uso de um *framework* como Pin se torna cada vez mais necessário para estudá-las (Seção 1.1.1). De fato, a variedade de usos de ferramentas de análise dinâmica existentes atualmente indica que há muito progresso tecnológico sendo feito nessa área. A Figura 1.10 ilustra essa variedade.

De maneira geral, através da Figura 1.10, é possível identificar pontos de interesse de pesquisa que podem crescer em relevância nos próximos anos. O primeiro deles se refere ao uso de técnicas de análise dinâmica em aplicações *Web*. Se por um lado a análise estática dessas aplicações era suficiente no passado, atualmente existe a necessidade do uso de análise dinâmica. Isso se deve ao fato de que hoje essas aplicações tendem a utilizar cada vez mais interfaces de interação com o usuário e códigos *Javascript*, que tornam ainda mais complexa a estrutura de navegação [Cornelissen et al. 2009]. Além disso, vemos que os artigos que tratam do tema de análise dinâmica costumam focar nos métodos de avaliação por estudos de caso. À medida que as técnicas de análise dinâmica se tornem mais populares, possivelmente surgirá a necessidade de aumentar o uso de *benchmarks* para avaliação do desempenho das ferramentas, já que a sobrecarga em tempo de execução é fator importante quando se fala de análises dinâmicas e, em particular, instrumentação dinâmica de binários.

Em relação às atividades em que a análise dinâmica são empregadas, as visualizações são as mais populares. Conforme discutido na Seção 1.1.1 e considerando a crescente complexidade dos sistemas computacionais, esse fato provavelmente tenderá a

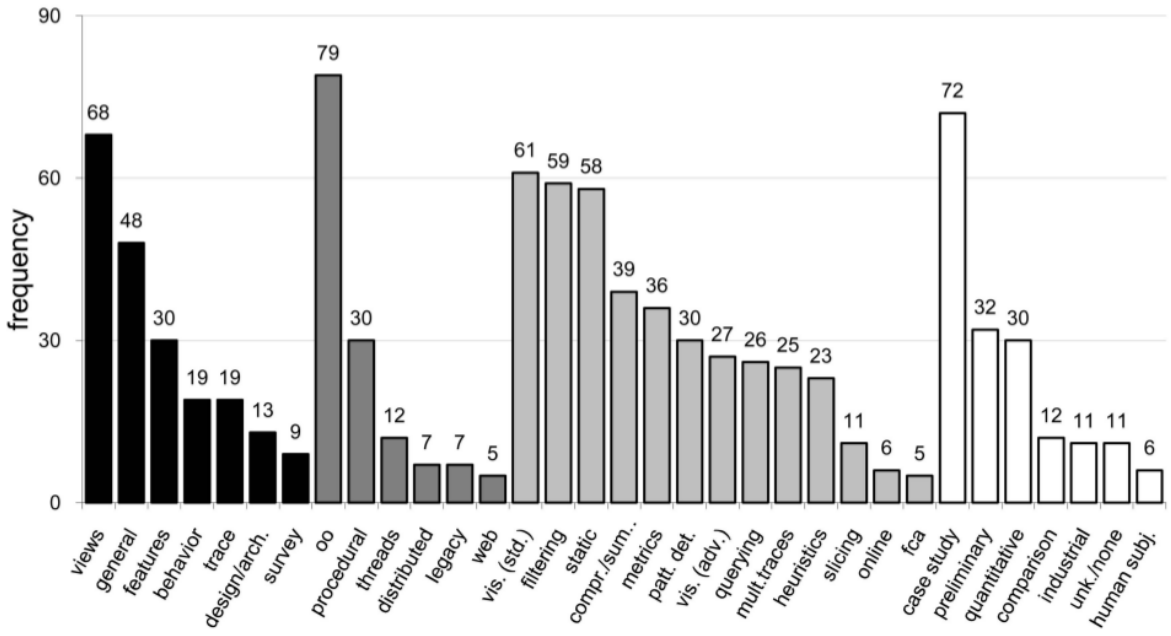


Figura 1.10. Frequência de atributos encontrados nos artigos sobre análise dinâmica até o ano de 2009 [Cornelissen et al. 2009]. As cores indicam a categoria do atributo. Da esquerda para a direita: atividade (preto), aplicações alvo (cinza escuro), método de análise dinâmica utilizado na condução da atividade (cinza claro) e método de avaliação através do qual a abordagem é validada (branco). Essa figura tem como referência 176 **artigos analisados.**

ser mantido no futuro. Entretanto, outras aplicações, como aquelas discutidas na Seção 1.1.2.1, deverão se popularizar com a difusão das técnicas de análise dinâmica.

Por fim, a Figura 1.10 revela que o uso de informações estáticas é o terceiro método mais utilizado. Levando em consideração que as análises estáticas também possuem vantagens (Seção 1.1.2.2), os dados da figura mostram que a abordagem mista (ou híbrida) tem ganhado popularidade. Isso acontece porque, utilizando essa abordagem, é possível combinar vantagens tanto das análises dinâmicas quanto estáticas.

Referências

[AMD 2017] AMD (2017). AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. `https://support.amd.com/TechDocs/24594.pdf`.

[Blake 2011] Blake (2011). MY MP3 Player 3.0 m3u Exploit DEP Bypass. `http://www.exploit-db.com/exploits/17854/`.

[Carlini and Wagner 2014] Carlini, N. and Wagner, D. (2014). Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, pages 385–399.

[Checkoway et al. 2009] Checkoway, S., Feldman, A. J., Kantor, B., Halderman, J. A., Felten, E. W., and Shacham, H. (2009). Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. *EVT/WOTE*, 2009.

[Chen et al. 2009] Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. (2009). Drop: Detecting return-oriented programming malicious code. In *International Conference on Information Systems Security*, pages 163–177. Springer.

[Chen et al. 2011] Chen, P., Xing, X., Mao, B., Xie, L., Shen, X., and Yin, X. (2011). Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 20–29. ACM.

[Cornelissen et al. 2009] Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., and Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702.

[Davi et al. 2009] Davi, L., Sadeghi, A.-R., and Winandy, M. (2009). Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 49–54. ACM.

[Davi et al. 2011] Davi, L., Sadeghi, A.-R., and Winandy, M. (2011). Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM.

[Devor 2013] Devor, T. (2013). Pin cgo tutorial. *CGO'13*.

[Emílio et al. 2015] Emílio, R., Tymburibá, M., and Pereira, F. (2015). Inferência estática da frequência máxima de instruções de retorno para detecção de ataques rop. In *Anais do XV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 2–15. SBC.

[Ferreira et al. 2014] Ferreira, M. F. T. et al. (2014). Rip-rop: uma proteção contra ataques de execução de código arbitrário baseados em return-oriented programming.

[Garnett 2003] Garnett, T. (2003). *Dynamic optimization of IA-32 applications under DynamoRIO*. PhD thesis, Massachusetts Institute of Technology.

[GNU 2018] GNU (2018). The GNU Operating System and The Free Software Movement. `https://www.gnu.org/home.en.html`.

[Göktas et al. 2014] Göktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. (2014). Out of control: Overcoming control-flow integrity. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 575–589. IEEE.

[Guide 2011] Guide, P. (2011). Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2.

[Intel 2004] Intel (2004). Pin Dynamic Binary Instrumentation Tool - Yahoo Groups. `https://groups.yahoo.com/neo/groups/pinheads/info`.

[Intel 2012] Intel (2012). Pin - A Dynamic Binary Instrumentation Tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`.

[Intel 2016] Intel (2016). Intel 64 and IA-32 Architectures Software Developer's Manual. `https://software.intel.com/en-us/articles/intel-sdm`.

[Intel 2018a] Intel (2018a). Intel Architecture Instruction Set Extensions and Future Features Programming Reference. `https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf`.

[Intel 2018b] Intel (2018b). Pin 3.6 user guide. `https://software.intel.com/sites/landingpage/pintool/docs/97554/Pin/html/`.

[Kanellos 2004] Kanellos, M. (2004). AMD, Intel put antivirus tech into chips. `https://www.zdnet.com/article/amd-intel-put-antivirus-tech-into-chips/`.

[Kleen a] Kleen, A. LWN.net advanced usage of last branch records. `https://lwn.net/Articles/680996/`. Acessado em: 19-04-2018.

[Kleen b] Kleen, A. LWN.net an introduction to last branch records. `https://lwn.net/Articles/680985/`. Acessado em: 19-04-2018.

[Laurenzano et al. 2010] Laurenzano, M. A., Tikir, M. M., Carrington, L., and Snavely, A. (2010). Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183. IEEE.

[Luk et al. 2005] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM.

[Min et al. 2013] Min, J.-W., Jung, S.-M., and Chung, T.-M. (2013). Detecting return oriented programming by examining positions of saved return addresses. In *Ubiquitous Information Technologies and Applications*, pages 791–798. Springer.

[Nethercote 2004] Nethercote, N. (2004). Dynamic binary analysis and instrumentation. Technical report, University of Cambridge, Computer Laboratory.

[Nethercote and Seward 2007a] Nethercote, N. and Seward, J. (2007a). How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74. ACM.

[Nethercote and Seward 2007b] Nethercote, N. and Seward, J. (2007b). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM.

[One 1996] One, A. (1996). Smashing the stack for fun and profit. *http://phrack.org/issues/49/14.html*.

[Reddi et al. 2004] Reddi, V. J., Settle, A., Connors, D. A., and Cohn, R. S. (2004). Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, page 22. ACM.

[Rodríguez et al. 2014] Rodríguez, R. J., Artal, J. A., and Merseguer, J. (2014). Performance evaluation of dynamic binary instrumentation frameworks. *IEEE Latin America Transactions*, 12(8):1572–1580.

[Romer et al. 1997] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., Bershad, B., and Chen, B. (1997). Instrumentation and optimization of win32/intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop*, volume 1997, pages 1–8.

[Shacham 2007] Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM.

[Tolomei 2015] Tolomei, G. (2015). In-Memory Layout of a Program (Process). `https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/`.

[Tymburibá et al. 2015] Tymburibá, M., Emilio, R., and Pereira, F. (2015). Riprop: A dynamic detector of rop attacks. In *Proceedings of the 2015 Brazilian Congress on Software: Theory and Practice*, pages 1–8.

[Tymburibá et al. 2016] Tymburibá, M., Moreira, R. E., and Quintão Pereira, F. M. (2016). Inference of peak density of indirect branches to detect rop attacks. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 150–159. ACM.

[Tymburibá et al. 2012] Tymburibá, M., Rocha, T., Martins, G., Feitosa, E., and Souto, E. (2012). Análise de vulnerabilidades em sistemas computacionais modernos: Conceitos, exploits e proteções. In dos Santos, A., Santin, A., Maziero, C., and da S. Gonçalves, P. A., editors, *Minicursos do XII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, chapter 1, pages 2–51. Sociedade Brasileira de Computação, Porto Alegre.

[Tymburibá et al. 2014] Tymburibá, M., Santos, A., and Feitosa, E. (2014). Controlando a frequência de desvios indiretos para bloquear ataques rop. In *Anais do XIV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 223–236. SBC.

[Uh et al. 2006] Uh, G.-R., Cohn, R., Yadavalli, B., Peri, R., and Ayyagari, R. (2006). Analyzing dynamic binary instrumentation overhead. In *WBIA Workshop at ASPLOS*. Citeseer.

[Uhlig and Mudge 1997] Uhlig, R. A. and Mudge, T. N. (1997). Trace-driven memory simulation: A survey. *ACM Computing Surveys (CSUR)*, 29(2):128–170.

[Vendicator 2000] Vendicator (2000). Stack Shield: A stack smashing technique protection tool for Linux. `http://www.angelfire.com/sk/stackshield/`.

Capítulo 2

NoSQL e a Importância da Engenharia de Software e da Engenharia de Dados para o Big Data

Tassio Sirqueira e Humberto Dalpra

Abstract

The world has been producing a large amount of data currently. With the Internet of Things, we have a network of devices capable of collecting, transmitting and processing data and with this large amount of data, storing and processing these is the new challenge. Relational databases are being used by large corporations, however, the high data production implies relational databases, difficulties in scalability and performance, considering the respect of ACID properties and normal forms. We are currently working with data, often without a fixed structure, which culminates in the use of non-relational database management systems (DBMS), also known as NoSQL DBs. To better understand this paradigm change, it is necessary to characterize this data, extracting the main characteristics and defining when to use a relational DBMS or a NoSQL. This paper aims to explain the main characteristics of the NoSQL databases and to mix, theory and practice, to create a CRUD application using the Java programming language, which uses MongoDB, one of the main NoSQL DBMSs on the market. Also the new profiles of software engineers and data will be addressed, and what the Big Data is breaking, opposite paradigms already established.

Resumo

O mundo vem produzindo uma grande quantidade de dados atualmente. Com a internet das coisas, temos uma rede de dispositivos capazes de coletar, transmitir e processar dados e com essa grande quantidade de dados, o armazenamento e processamento destes é o novo desafio. Os bancos de dados relacionais vêm sendo utilizados por grandes corporações, entretanto, a produção elevada de dados imputam, aos bancos relacionais, dificuldades de escalabilidade e performance, considerando o respeito as propriedades ACID e as formas normais. Atualmente já trabalhamos com dados, muitas vezes sem

estrutura fixa, o que culmina no uso de sistemas gerenciadores de bancos de dados (SGBDs) não relacionais, também conhecidos como SGDBs NoSQL. Para melhor compreensão dessa mudança de paradigma, é necessário a caracterização destes dados, extraindo as principais características e definindo quando deve-se utilizar um SGBD relacional ou um NoSQL. Esse trabalho visa explicar as principais características dos bancos de dados NoSQL e miscigenando, teoria e prática, criar uma aplicação CRUD, utilizando a linguagem de programação Java, que utilize o MongoDB, um dos principais SGBDs NoSQL do mercado. Também serão abordados os novos perfis dos engenheiros de software e de dados, e o que o Big Data representa de rompimento, frente a paradigmas já consolidados.

2.1. Introdução

Quando iniciamos o desenvolvimento de um novo projeto de software, nos são imputadas tomadas de decisões importantes como sobre qual linguagem de programação será utilizada no projeto, qual o *framework* e reuso de código será abordado, além dos requisitos que são definidos pelos usuários. Um detalhe importante e pouco discutido em geral é com relação a persistência dos dados, onde é definindo qual o tipo de Sistema Gerenciador de Banco de Dados (SGBD) será utilizado, dentre as várias opções e disponibilidades existentes no mercado da informática. Apesar dos bancos relacionais serem os mais utilizados na maioria dos projetos, alguns detalhes devem ser considerados na tomada de decisão como, por exemplo, a flexibilidade dos dados e a escalabilidade da aplicação.

Com o crescimento da internet e a popularização do chamado "Internet das Coisas" (Gubbi *et al*., 2013), onde objetos físicos possuem tecnologias embarcadas e são capazes de comunicar entre si, passamos a possuir redes que coletam, transmitem e processam dados em grande quantidade. Desde portais de notícias, com milhares de acessos simultâneos, a sistemas de informação, em tempo real, ou aplicações de IoT (*Internet of things* – Internet das Coisas), o volume gerado e a dificuldade de manipulação desses dados crescem a cada dia. Além do grande volume de dados a serem armazenados, temos os desafios de extrair destes dados informações úteis. A forma de armazenamento implica diretamente em como manipular e na facilidade de cumprir com essa função.

Essa nova demanda de flexibilidade e escalabilidade das aplicações por conta do volume de dados, velocidade de processamento e/ou falta de estrutura fixa dos dados, em muitos casos emerge o uso de sistemas gerenciados de banco de dados não relacionais, também conhecidos como SGBDs NoSQL (Leavitt, 2010). NoSQL (*Not Only SQL)*, traduzido como 'Não apenas SQL', ressalta a importância da utilização deste para diferentes tipos de SGBD, promovendo a ferramenta ao armazenamento de dados, de acordo com a necessidade.

Atualmente existem diversos sistemas gerenciadores de banco de dados não relacionais, os quais são classificados de acordo com a forma em que os dados são

persistidos. Entre os principais tipos de SGBD NoSQL temos: i) Chave-valor; ii) Colunar; iii) Grafo; e iv) Documento, o qual será o principal foco na descrição deste trabalho.

Nos SGBDs NoSQL tipo Chave-valor, todos os registros compõem a mesma coleção (Tabela) e a única característica em comum entre os registros é uma chave única (*Hash*), sendo o modelo mais simples e fácil de implementar. São exemplos desse tipo de SGBD NoSQL o Riak[1], Redis[2], LevelDB[3] e RocksDB[4].

Nos SGBDs NoSQL tipo Coluna, todos os registros fazem parte da mesma tabela, sendo que as colunas variam de acordo com os registros. Esse tipo de banco é otimizado para colunas de leitura e gravação, ao contrário dos SGBDs relacionais que são linhas de dados. São exemplos desse tipo de SGBD o HBase[5], Cassandra[6] e SimpleDB[7].

Já nos sistemas de bancos NoSQL tipo Grafo, os registros são representados como nós e os relacionamentos como arestas, formando um grafo. São exemplos desse tipo de SGBD o Neo4j[8], GraphBase[9] e HyperGraphBase[10].

Nos SGBDs NoSQL tipo Documento, cada registro fica armazenado em um arquivo específico, geralmente em formato JSON (*JavaScript Object Notation*) ou XML (*eXtensible Markup Language*), dentro de uma coleção. Diferente dos SGBDs relacionais, o esquema dos documentos pode variar, permitindo maior flexibilidade e o tamanho do documento torna-se variável. Alguns exemplos desse tipo de SGBD são o Couchbase[11], CouchDB[12] e o MongoDB[13].

Cada tipo de SGBD NoSQL possui suas vantagens e desvantagens, de acordo com o seu tipo e sua implementação, uma lista completa de SGBDs NoSQL pode ser consultada em <http://nosql-database.org/>. Ao longo deste trabalho enfocaremos nos bancos NoSQL do tipo Documento, imputando o MongoDB. Atualmente o MongoDB é o SGBD NoSQL mais popular de todos, desenvolvido como um projeto de código aberto,

---

[1] Riak: Disponível em <http://basho.com/products/#riak>.
[2] Redis: Disponível em <https://redis.io/>.
[3] LevelDB: Disponível em <https://github.com/google/leveldb>.
[4] RocksDB: Disponível em <http://rocksdb.org/>.
[5] HBase: Disponível em <http://hbase.apache.org/>.
[6] Cassandra: Disponível em <https://cassandra.apache.org/>.
[7] SimpleDB: Disponível em <https://aws.amazon.com/pt/simpledb/>.
[8] Neo4j: Disponível em <https://neo4j.com/>.
[9] GraphBase: Disponível em <https://graphbase.ai/>.
[10] HyperGraphBase: Disponível em <http://www.kobrix.com/hgdb.jsp>.
[11] Couchbase: Disponível em <https://www.couchbase.com/>.
[12] CouchDB: Disponível em <http://couchdb.apache.org/>.
[13] MongoDB: Disponível em <https://www.mongodb.com/>.

multiplataforma e com suporte as principais linguagens de programação. É importante destacar que os SGBDs NoSQL não vieram para substituir os SGBDs relacionais, mas sim como alternativa para determinadas aplicações.

A Engenharia de Software (ES) atua na área de Software e Serviços (Lopes *et al*. 2005), onde os profissionais devem possuir uma visão que engloba a criação, teste, implantação e manutenção de sistemas de software. Também são visados novos sistemas computacionais para o mercado, sendo vital que estes profissionais tenham a habilidade de captar as necessidades dos clientes, sendo que estas são importantes para chegar-se a melhor solução para os problemas.

As soluções supramencionadas incluem aplicações para empresas tais como sistemas de informação, aplicações de software embarcado, dispositivos móveis ou dispositivos de IoT e sistemas para ambientes industriais e de automação.

O engenheiro de software atua fortemente na interação humano/computador. Em meio as equipes de trabalho, buscam um maior envolvimento com os negócios e ambiente dos clientes, afim de apresentar e implementar as aplicações e soluções necessárias. Sendo assim é importante a atualização de tais profissionais, tendo em vista que os mesmos tendem a buscar o que há de mais novo em relação as novas demandas que surgem no campo da computação. Essas atualizações incluem novas linguagens de programação, novos domínios, como por exemplo o de internet das coisas, e também na área de armazenamento de dados, visto o grande volume de dados manipulados pelas empresas atualmente, popularmente conhecidos como *Big Data*.

Assim, os profissionais devem conhecer as novas tecnologias, no intuito de sempre encontrar as melhores soluções para cada problema. As características dos novos engenheiros de software e de dados serão melhor discutidas à frente, dada que as mesmas são de extrema importância para auxiliar na compreensão exata do contexto. Na próxima seção abordaremos o papel da engenharia de dados e da engenharia de software na computação.

2.2. Engenharia de Dados e de Software

A Engenharia contempla a ciência e tecnologia para criação e construção de sistemas que sanem problemas. Pode-se definir a Engenharia de Dados como uma área da Engenharia dedicada a processar e tratar dados para aplicações que utilizarão *Big Data*, e o banco de dados vem a ser uma representação dinâmica do mundo real, onde os dados podem sofrer alteração temporal. Como descrito por Date (2004), um sistema de banco de dados é apenas um “sistema computadorizado de manutenção de registros”.

Já na Engenharia de Software, os projetos enfocam Teste e Validação de Sistemas, Qualidade de Software, Engenharia Reversa e Reengenharia de Software Orientadas a Objetos, Desenvolvimento de Software Orientado a Objetos, Engenharia de Requisitos

de Software, Paradigma Transformacional de Desenvolvimento, Ferramentas e Técnicas para desenvolvimento de software, Reutilização e Desenvolvimento de Software na Computação Ubíqua.

Sommerville (2011) explica que existem vários tipos de sistemas de software, que vão desde aplicações simples até aplicações complexas de alcance mundial, e todas as aplicações necessitam de engenharia, pois diferentes tipos de software exigem diferentes abordagens. Um sistema de software é desenvolvido a partir da integração entre a engenharia de dados e da engenharia de software.

Durante o desenvolvimento de um software projeta-se o modelo de dados, as classes e suas relações e, nesse instante, é importante que analistas e programadores tenham conhecimento sobre o futuro da aplicação. Sommerville (2011) destaca que o papel da engenharia de software é apoiar o desenvolvimento profissional do software, acimo do desenvolvimento pessoal.

A mudança de tecnologia em banco de dados pode ser um fator crítico em aplicações com alta carga de trabalho e para isso, os profissionais devem saber o impacto que suas escolhas têm sobre o ciclo de vida do software. A mudança do tipo de SGBD não só impacta o banco de dados como a modelagem do sistema como um todo, e daí a importância da comunicação entre a engenharia de dados e a engenharia de software.

Desenvolver uma aplicação que envolve o paradigma de banco não relacional pode ser um desafio para as empresas acostumadas a desenvolverem aplicações tradicionais e a comunicação entre as áreas é que determina o sucesso do desenvolvimento.

Na próxima seção abordaremos um comparativo entre os SGBDs Relacionais e NoSQL, de modo a facilitar e melhorar a compreensão dos termos, definições, vantagens e desvantagens de cada um.

2.3. Banco de Dados Relacional vs. Banco de Dados Não Relacional

Atualmente as empresas sofrem um crescente volume de dados, onde as aplicações devem ser preparadas para suportar esta grande demanda, focando em um tempo de resposta satisfatório, haja vista que há uma quantidade grande de usuários, o que culmina em mudanças na engenharia de dados e software.

Como apresentado por Elmasri (2005), um banco de dados armazena um conjunto de dados do mundo real, o qual é denominado minimundo. O mesmo é projetado, construído e populado com dados de uma aplicação específica.

Nos últimos anos os avanços da tecnologia e a demanda por soluções cada vez mais pujantes, tornaram obsoletas as soluções de banco de dados tradicionais, que em sua natureza permitiam o armazenamento e manipulação de dados apenas em formato

alfanumérico. Essa nova demanda exige o armazenamento de imagens, *streams* de áudio e vídeo, dados de informações geográficas entre outras formas.

Com o crescimento contínuo na geração de dados, esta elevada quantidade impacta diretamente na escolha de um SGBD a ser utilizado no desenvolvimento de um software. Neste momento os engenheiros de dados devem considerar os prós e contras de cada tipo de SGBD, tendo em vista que os banco de dados são essenciais para a sociedade moderna (Elmasri, 2005). Ainda segundo Elmasri (2005), todo banco de dados cria alguma abstração sobre os dados, sendo essa característica importante para definição de qual banco de dados será utilizado em uma solução de software ou sistema.

Atualmente os SGBDs mais utilizados são relacionais, tendo suas primeiras implementações comerciais datadas na década de 80. Entre os SGBDs relacionais mais populares do mercado, podemos destacar o Oracle[14], o SQL Server[15], o MySQL[16] e o PostgreSQL[17]. Esses SGBDs representam o banco de dados como uma coleção de tabelas relacionadas, compostas por atributos com tipos específicos, onde cada registro é uma tupla da tabela, visando a representação de dados do minimundo do problema.

Algumas características devem ser observadas no modelo de banco de dados relacional, as quais indicam as formas normais (normalização) do banco. Essa normalização do banco relacional, busca garantir que todos os dados armazenados não sejam multivalorados, não apresentem dependência funcional parcial ou transitiva. Isso faz com que o banco de dados cresça em número de tabelas e relacionamentos, o que no *Big Data* gera consequências.

Já na modelagem de bancos de dados orientados a documentos por exemplo, pode-se utilizar uma estrutura complexa para armazenar os dados, onde as informações podem ser agrupadas em um único documento, de modo a representar a visão dos dados para os negócios da empresa.

Diferente da modelagem de banco relacional, onde a mesma é realizada no início do projeto, e ainda muita das vezes não temos a visão completa do projeto, em um banco NoSQL há liberdade de esquema, de modo que as alterações degradem menos o software. Vale ressaltar que mesmo com essa flexibilidade, deve haver um padrão a ser seguido, levando em consideração as seguintes características:

- Coerência: refere-se a um documento ser compreensível se analisado individualmente, ou seja, não depender de outros documentos.
- Independência: refere-se a um documento possuir razão própria para existir.

[14] Oracle: Disponível em <https://www.oracle.com/>.
[15] SQL Server: Disponível em <https://www.microsoft.com/pt-br/sql-server/>.
[16] MySQL: Disponível em <https://www.mysql.com/>.
[17] PostgreSQL: Disponível em <https://www.postgresql.org/>.

- Isolamento: refere-se que a modificação de um dado documento, não deve implicar na modificação de outro.

Além disso, como abordado por Cattell (2011), o *Big Data* fez com que, nos últimos anos, as aplicações fossem projetadas para fornecer uma melhor escalabilidade horizontal, possibilitando que os dados dos sistemas fossem distribuídos entre vários servidores (nós). Contudo, os sistemas de banco de dados tradicionais (relacionais) possuem pouca capacidade de expansão horizontal. Essa característica é explicada pelo teorema CAP (Brewer 2000), onde em um banco de dados há a presença de uma partição de rede, vindo a ser necessário a escolha entre a consistência e a disponibilidade dos dados. Uma representação do teorema CAP visto é representada na Figura 2.1.

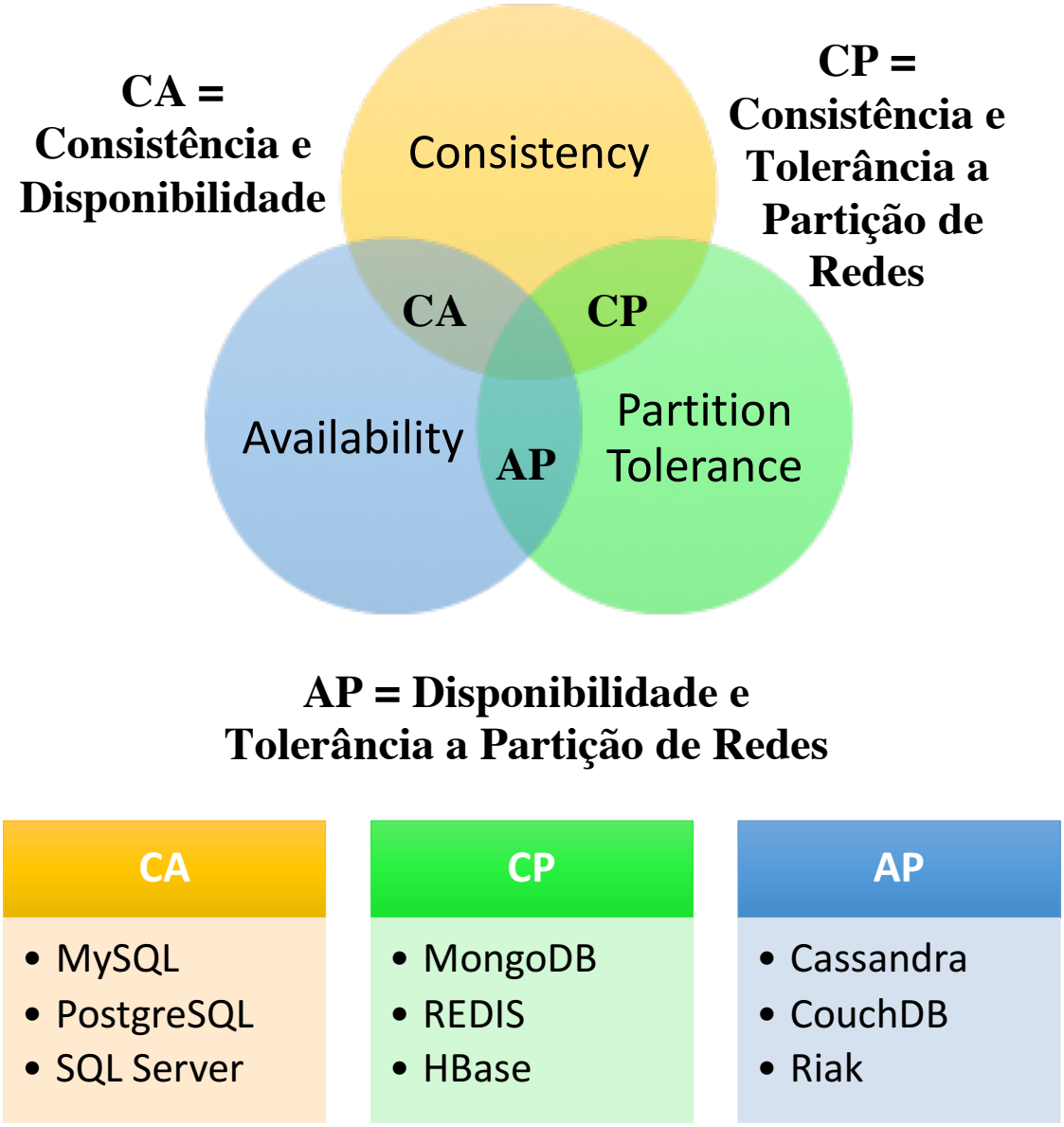


Figura 2.1. Representação do Teorema CAP.

O teorema CAP descreve o comportamento de um sistema distribuído, onde uma coleção de nós interconectados compartilha os dados entre si. Desta forma, um usuário pode conectar-se a qualquer nó desse sistema distribuído e realizar uma operação de escrita. Ao se conectar novamente ao mesmo ou outro nó do sistema distribuído, é possível realizar uma operação de leitura. A disponibilidade ou a consistência desse dado, dentro do sistema distribuído, é o que determina a reação do sistema distribuído.

Sendo assim, o teorema CAP afirma que um dado par de requisições, uma escrita seguida por uma leitura, em um panorama de sistemas distribuídos, garante apenas 2 entre os 3 comportamentos presentes no teorema. Esses comportamentos são definidos como:

- Consistência (Consistency): Consiste em garantir que a operação de leitura exiba o dado mais atualizado, independente do nó em que ele foi gravado. Essa característica garante que o cliente nunca receberá do banco um dado que já foi modificado.
- Disponibilidade (Availability): O comportamento de disponibilidade determina que nenhuma das requisições podem retornar erro e o usuário não pode aguardar indefinidamente pela resposta do SGBD, ou seja, o sistema deve se manter sempre disponível para leitura e gravação em um nó que não possui falha e este nó deve responder em um tempo razoável. Qualquer falha não pode afetar o funcionamento da aplicação que utiliza-a.
- Tolerância a Partição de Rede (Partition Tolerance): A Tolerância a Partição de Rede ou Tolerância a Falhas, determina que o sistema deve continuar operando mesmo após uma falha na rede (perda de conexão), garantindo que pelo menos as operações de leitura ocorram de forma normal.

Seguindo essa definição do teorema CAP, um SGBD pode combinar apenas duas dessas características, totalizando 3 combinações (CA, CP, AP). Essas combinações são explicadas na sequência:

- CA (Consistência e Disponibilidade): esta combinação têm sistemas de banco de dados que enfocam a consistência dos dados armazenados e a disponibilidade destes. Nesse modelo, qualquer falha em um dos nós, o sistema todo fica indisponível até que o nó que falhou volte ao normal. Essa é a configuração clássica dos SGBDs relacionais, sendo chamada de otimista, uma vez que a operação de escrita não gera inconsistência dos dados.
- CP (Consistência e Tolerância a Particionamento): Para sistemas que precisam da consistência forte e tolerância a particionamento, é necessário abrir mão da disponibilidade. Nesse modelo pode ocorrer que uma operação de escrita gere conflito entre os nós do particionamento de rede, sendo a disponibilidade comprometida até que ocorra um consenso entre os nós. Essa é uma abordagem

pessimista, visto que uma operação de escrita pode ser negada, apesar dos SGBDs tentarem evitar ao máximo que isso ocorra.

- AP (Disponibilidade e Tolerância a Particionamento): Quando pensamos em sistemas grandes e complexos, que atuam em todos os horários e dias (24/7) e nunca podem ficar *offline*, dependemos de alta disponibilidade e tolerância a particionamento. Esse modelo sacrifica a consistência, ou seja, o sistema sempre aceita as operações de escrita, mesmo que o sincronismo entre os nós ocorra em outro momento. Nesse período entre a persistência de um dado em um nó específico e seu sincronismo entre os demais nós do *cluster*, existe uma janela de inconsistência, onde uma operação de leitura em um nó que ainda não foi atualizado pode retornar dados desatualizados.

Enfatizando o ponto de vista do desenvolvimento, não existem tantas diferenças entre o CA e o CP, pois no modelo CA um sistema fica indisponível quando há particionamento, dado que possui alta disponibilidade por nó, enquanto no modelo CP o sistema tende a chegar em um consenso, onde o mesmo aceita uma escrita ou não. Na pior situação também é possível que esta fique indisponível para uma parte dos dados.

Essas características definem algumas vantagens ou desvantagens de um SGBD no sistema em que será empregado. Considerando um SGBD NoSQL, mesmo não existindo o particionamento, o SGBD pode priorizar o tempo de resposta e comprometer a consistência, dando prioridade a operações de leitura ao invés das operações de atualização, por exemplo.

O conhecimento do teorema CAP é fundamental para problemas do mundo real, onde a solução do problema atacado envolve o uso de sistemas distribuídos. Só assim os profissionais desse campo irão ter capacidade de desenvolver as melhores soluções.

A expansão do *Big Data* enfoca na necessidade de profissionais que saibam lidar com esse grande volume de dados, extraindo informações úteis e de valor para as empresas, sendo essa área denominada *Data Science* (ciência dos dados).

Data Science está cada vez mais em conjunto com *Big Data* e, conforme explicado por Dhar (2013), refere-se a buscar nos dados o conhecimento na forma de explicações testáveis e previsões sobre o universo analisado, ou seja, visa a extração de conhecimento para possíveis tomadas de decisões, podendo alinhar *Big Data*, aprendizado de máquina (*Machine Learning*) e mineração de dados (*Data Mining*).

Procurar nos dados padrões consistentes não é uma tarefa trivial, dado a volatilidade destes. Essa expansão do *Big Data* e o crescimento dos dispositivos de IoT, ocasionaram em operações com grandes quantidades de dados, muitas vezes semiestruturados, advindos de diferentes formas e formatos.

O objetivo da próxima seção é apresentar problemas reais que envolvem *Big Data* e como, parte do desafio para se encontrar a melhor solução para o problema, envolve o uso de SGBDs NoSQL.

2.4. *Big Data*, limitações dos SGBDs Relacionais e os novos desafios

Big data é uma realidade para muitas empresas, culminando em um desafio. O mercado demanda profissionais que estejam atualizados e saibam lidar com esse novo cenário. *Big Data* não é simplesmente trabalhar com grande volume de dados, mas saber trata-los e manipula-los em diversos formatos diferentes. Exemplos que podem ser citados são dados econômicos e de saúde, muitas vezes encontrados em arquivos de *Excel*, CSV (*Comma-separated values*) ou texto.

Como explicado por Mayer-Schonberger & Cukier (2014), o *Big Data* se baseia na capacidade de uma sociedade em obter informações de modo a gerar novas ideias, e agregar valor para bens e serviços, desafiando a maneira como vivemos e nossas decisões. Para Amaral (2016), todo dado possui um ciclo de vida e devemos extrair o máximo de informação enquanto o dado é valido. Esse ciclo de vida do dado é apresentado na Figura 2.2.

Essa capacidade de extrair informações do *Big Data,* envolve 3 características básicas, chamadas de 3Vs, que são: i) Volume; ii) Velocidade; e iii) Variedade. Também podem ser adicionadas outras características tais como, confiabilidade e valor. Esse processo de extração da informação, envolve a "análise descritiva", respondendo à pergunta: "O que aconteceu e por quê?"; a "análise preditiva", estimando a probabilidade de um dado evento e a "análise prescritiva" fornecendo recomendações (prescrições) específicas ao usuário.

Figura 2.2. Ciclo de vida do dado.

Essa diversidade de dados é algo crescente e até mesmo soluções simples podem se tornar interessantes para o uso de um SGBD NoSQL. Vamos descrever a seguir um exemplo simples de sistema e modelar o mesmo para um SGBD relacional e para um SGBD NoSQL, sendo realizado um comparativo da modelagem entre ambos.

Imagine um sistema onde será armazenado o cadastro de uma pessoa, contendo as seguintes informações básicas: i) nome; ii) sobrenome; iii) e-mail; iv) telefone e v) endereço. Agora imagine que esse sistema utilizará como SGBD um banco de dados relacional, o qual, no processo de modelagem, representará o resultado desse cadastro como uma tabela, conforme ilustrado na Figura 2.3. São exemplos de dados desta tabela do banco os valores exibidos na Tabela 2.1.

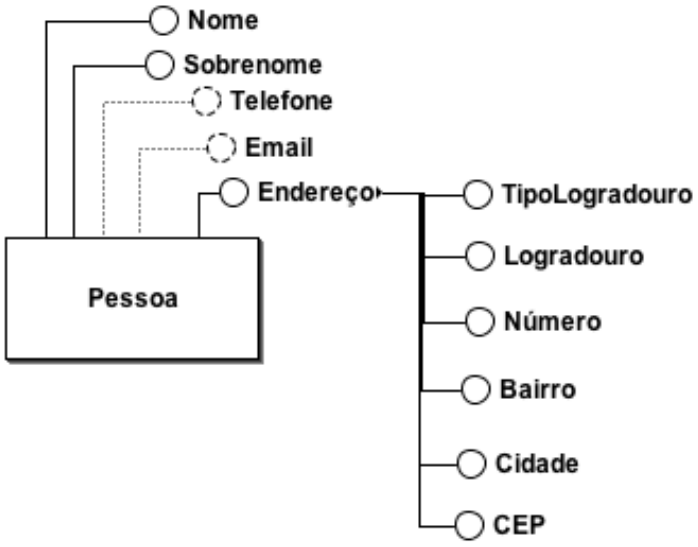


Figura 2.3. MER da Tabela Pessoa.

Tabela 2.1. Dados do Cadastro da Pessoa.

idPessoa	Nome	Sobrenome	E-mail	Telefone	Endereço
1	Tassio	Sirqueira	tassio @tassio.eti.br	3298444 0000	Rua A, 1, Centro, Matias Barbosa, 36120-000
2	Humberto	Dalpra	humbertodalpra @gmail.com	3298833 0000	Rua B, 1, Centro, Juiz de Fora, 36100-000
3	José	Da Silva		2199111 0000	Rua B, 1, Centro, Matias Barbosa, 36120-000
4	João	Nunes			Av Cardoso, 1, Borboleta, Juiz de Fora, 36110-100
5	Maria	Flor	flor.m @mail.com		Rua Bandeirantes, 1, Florais, Juiz de Fora, 36100-110

Conforme supramencionado, para os SGBDs relacionais, o banco de dados deve eliminar atributos multivalorados, a dependência funcional parcial e/ou transitiva,

também conhecida como formas normais ou processo de normalização. Isso representa que a Tabela 2.1 não pode permitir uma coluna chamada “endereço” contendo todos os dados, conforme representado.

Dessa forma, tem-se duas tabelas no sistema, uma para o cadastro da pessoa e outra para o seu endereço, conforme o modelo entidade-relacionamento apresentado na Figura 2.4.

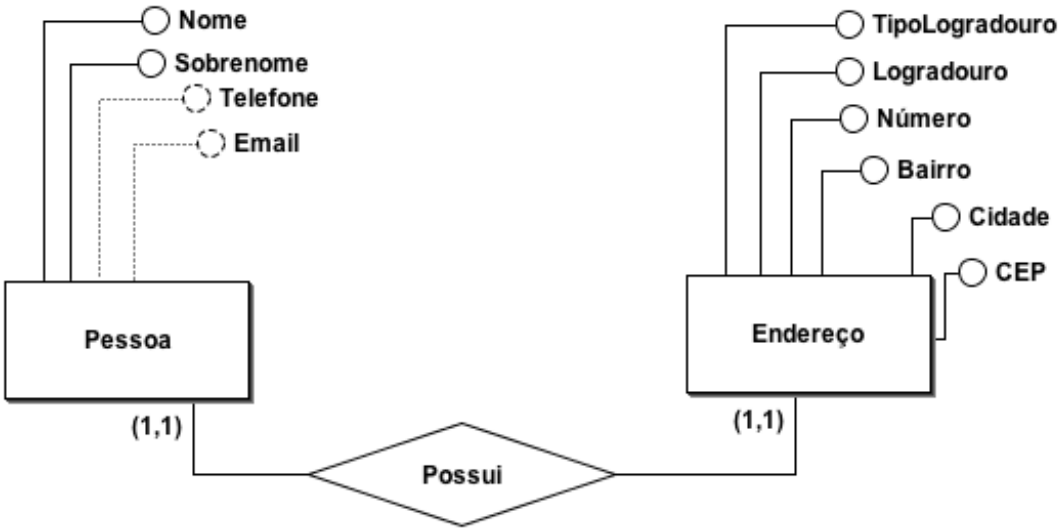


Figura 2.4. MER Pessoa e Endereço.

No exemplo exibido na Figura 2.4 resolvemos parte da normalização, tendo em vista que a tabela Pessoa receberá uma chave estrangeira da tabela Endereço. Dando continuidade. Imagine que esse mesmo cadastro possa permitir a pessoa possuir dois endereços, um residencial e outro comercial, e além disso, que possa ter dois e-mails (por exemplo, pessoal e de trabalho) e três telefones (por exemplo, de casa, celular pessoal e de trabalho).

Afim de sanar o problema do endereço passaríamos a chave estrangeira para a tabela Endereço e para os demais casos teríamos que criar novos atributos na tabela Pessoa. Se o usuário pudesse cadastrar quantos e-mails ou telefones ele deseja-se, isso não seria possível no modelo apresentado na Figura 2.4. Outro problema que pode-se visualizar na Tabela 2.1 é quando o cadastro não possui todos os atributos preenchidos, haja vista que nos SGBDs relacionais essas informações são representadas como vazias nas tuplas.

No modelo da Figura 2.5 tem-se uma representação de como os dados são armazenados em um SGBD NoSQL do tipo Documento, tendo em vista que cada documento pode apresentar uma estrutura de acordo com a necessidade. Assim, um exemplo dos dados completos, para o primeiro registro apresentado na Tabela 2.1, temos como saída o documento em formato JSON, apresentado na Figura 2.6.

O modelo de documento flexível do MongoDB facilita a criação de modelos de dados avançados que refletem como os dados são usados em um aplicativo. Entender como os dados são usados pelo seu aplicativo é fundamental para um *design* efetivo do banco de dados. Ao projetar um banco de dados de documentos, é importante entender quais entidades de aplicativo serão consultadas juntas. O MongoDB possibilita salvar dados comumente usados no mesmo registro. Ao salvar dados comumente usados em um único registro, os desenvolvedores de aplicativos podem obter enormes vantagens de desempenho, tendo apenas que consultar um único registro, em vez de criar consultas com várias associações.

Considerando essa flexibilidade, o exemplo da Tabela 2.1 para o 4º registro, o documento JSON gerado não necessita possuir os mesmos atributos do primeiro, conforme pode-se observar na Figura 2.7.

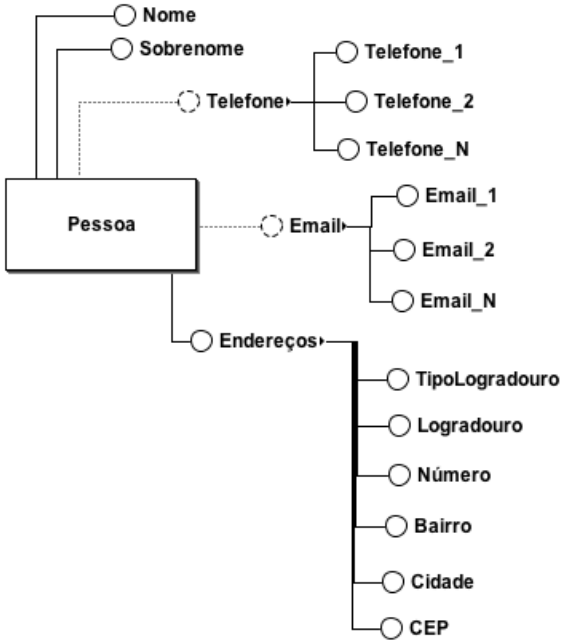


Figura 2.5. MER da tabela Pessoa estendido.

```
{
    "_id" : ObjectId("5aca8e88701e2721003eef97"),
    "nome" : "Tassio",
    "sobrenome" : "Sirqueira",
    "email" : "tassio@tassio.eti.br",
    "telefone" : "32984440000",
    "endereco" : {
        "tipoLogradouro" : "Rua",
        "logradouro" : "A",
        "numero" : "1",
        "bairro" : "Centro",
        "cidade" : "Matias Barbosa",
        "cep" : "36120-000"
    }
},
```

Figura 2.6. JSON armazenando os dados do primeiro registro.

```
{
    "_id" : ObjectId("5aca8e88701e2721003eef98"),
    "nome" : "João",
    "sobrenome" : "Nunes",
    "endereco" : {
        "tipoLogradouro" : "Av",
        "logradouro" : "Cardoso",
        "numero" : "1",
        "bairro" : "Borboleta",
        "cidade" : "Juiz de Fora",
        "cep" : "36110-100"
    }
}
```

Figura 2.7. JSON armazenando os dados do quarto registro.

Essa característica é importante pois, conforme supramencionado, se desejarmos adicionar outros atributos a um registro, podemos faze-lo ao documento ou a sub-atributos do documento sem alterar os demais registros, conforme demonstrado na Figura 2.8.

```
{
    "_id" : ObjectId("5aca912e701e2721288f1f6c"),
    "nome" : "Tassio",
    "sobrenome" : "Sirqueira",
    "email" : "tassio@tassio.eti.br",
    "telefone" : {
        "casa" : NumberLong("3232730000"),
        "celular" : 984440000
    },
    "endereco" : {
        "tipoLogradouro" : "Rua",
        "logradouro" : "A",
        "numero" : "1",
        "bairro" : "Centro",
        "cidade" : "Matias Barbosa",
        "cep" : "36120-000"
    }
}
```

Figura 2.8. JSON armazenando os dados do primeiro registro com dois telefones.

Essa dinamicidade no armazenamento dos dados que o banco NoSQL, do tipo documento, possui, permite armazenar dados com diferentes estruturas na mesma coleção. Apesar do exemplo demonstrado ser algo simples, quando trabalha-se com Big Data e dispositivos de IoT, sendo necessário armazenar esses dados, a variedade de dado e o tipo deles tendem a não manter-se constantes. Os dados em bases relacionais deixam de seguir a normalização ou o crescimento da base e o número de atributos vazios tornam-se enormes.

Analisando dados de saúde, por exemplo, advindos de um monitor multiparamétrico (utilizados por pacientes nas Unidades de Terapia Intensiva), observa-se que os dados recebidos podem variar conforme os sensores acoplados aos pacientes e aos alertas configurados.

Todos os equipamentos de saúde que atuam com o protocolo HL7, não seguem um padrão fixo de envio dos dados, o que dificulta seu armazenamento. Coletar dados de pacientes em UTIs que funcionam a todos os dias e horários (24/7) gera um *Big Data*, o qual não pode ser descartado, uma vez que os dados coletados devem ser mantidos pelos hospitais e servem como base de conhecimento para o corpo médico acompanhar o estado de saúde dos pacientes.

Áreas como saúde, economia, geologia e climatologia já enfrentam o problema do *Big Data*. Ao desenvolvermos aplicações para essas áreas, devemos identificar os pontos cruciais para o sucesso da solução. Na próxima seção abordaremos de modo mais profundo os SGBDs NoSQL, em especial o MongoDB.

2.5. Introdução ao NoSQL e MongoDB

Atualmente, banco de dados desenvolvidos com o uso do NoSQL permitem as empresas trabalharem com dados semiestruturados e com grande poder de escalabilidade para o universo do *Big Data*. Sistemas como o Facebook, que possuía em 2013 uma média de 2,4 bilhões de itens compartilhados por dia, entre os seus usuários, são dependentes de soluções NoSQL.

Essa quantidade e velocidade de produção dos dados, imputam em que, os mecanismos de armazenamento de dados, satisfaçam as necessidades de diferentes aplicações. Dentre as três soluções mais comuns pode-se citar o armazenamento direto no sistema de arquivos, bancos de dados relacionais e bancos de dados NoSQL. O termo "NoSQL" significa "*Not only SQL*" (não apenas SQL), onde neste pode-se aplicar o conceito *SQL* (*Structured Query Language* ou Linguagem de Consulta Estruturada).

Grolinger *et al*. (2013) discutem que o termo NoSQL foi usado pela primeira vez em 1998 para um banco de dados relacional, o qual omitiu o uso do SQL e foi retomado em 2009, quando passou a ser realmente utilizado por devido a necessidades que partiram de grandes instituições tais como Google, Yahoo, Facebook e Twitter. Grolinger *et al*. (2013) ainda afirmam que os sistemas de gerenciamento de banco de dados relacional, tradicionais, foram projetados em uma era em que o hardware disponível, bem como os requisitos de armazenamento e processamento, eram muito diferentes do que são hoje. Essa solução tem encontrado muitos desafios para atender aos requisitos de desempenho e dimensionamento do chamado "*Big Data*".

Algumas das razões pelas quais os SGBDs NoSQL vêm ganhando mercado, são explicadas por Strauch *et al.,* (2011), os quais abordam que os SGBDS NoSQL apresentam características importantes para o *Big Data*, sendo elas: i) evitar a complexidade desnecessária; ii) alta taxa de transferência; iii) escalabilidade horizontal; iv) evitar o mapeamento objeto-relacional que pode custar caro; v) a complexidade e o custo da configuração de *clusters* de banco de dados; vi) melhor desempenho, entre outras características.

Devido à variedade das abordagens NoSQL e a sobreposições em relação aos requisitos não-funcionais e ao conjunto de recursos, pode ser difícil obter e manter uma visão geral dos bancos de dados não relacionais. Diferente dos SGBDs relacionais, no NoSQL não há suporte ao ACID (*Atomicity, Consistency, Isolation, Durability)*. Também não há dependência de tabelas e colunas fixas, bem como não há o uso padrão das consultas SQL.

Dentre os projetos NoSQL mais notáveis até o momento, podemos citar o projeto de software livre MongoDB, que nada mais é que um banco de dados orientado a documentos, o qual armazena dados em coleções de documentos semelhantes a JSON, compostos por nomes de campos e um tipo específico de valor.

A distinção do SGBD MongoDB em relação a outros bancos de dados NoSQL é a poderosa linguagem de consulta baseada em documento, a qual torna fácil a transição de um banco de dados relacional para o MongoDB, haja vista que as consultas ou instruções SQL são convertidas à respectiva função. Ele também apresenta simplicidade na instalação e utilização.

Outro detalhe importante, conforme supramencionado, é que banco de dados relacionais permitem a escalabilidade, contudo, quanto maior o tamanho do banco de dados, maior é o custo de manter esta, seja para adicionar novas máquinas ou para manter a equipe responsável pela administração do banco. Já os bancos de dados não relacionais permitem uma escalabilidade mais barata e menos trabalhosa, visto que as máquinas não precisam ser poderosas e a equipe de administração do banco necessita de um número menor de pessoas. Tais características são associadas ao teorema CAP.

Na próxima seção será apresentado o banco de dados NoSQL MongoDB e suas principais características.

2.5.1. Apresentando o MongoDB

Banco de Dados Orientados a Documentos contém todas as informações importantes de uma entidade em um único documento, cada grupo de entidades formam as coleções características, tais como ser livre de esquemas, possuir identificadores únicos universais (*UUID*), possibilitar a consulta de documentos através de métodos avançados de agrupamento e filtragem (*MapReduce*), até permitir redundância e inconsistência, são os que definem essa classe de SGBD.

Esses bancos de dados orientados a documentos também são chamados de Bancos NoSQL (*Not Only SQL*). O termo NoSQL é devido à ausência do SQL (*Structured Query Language*), mas esse tipo de Banco de Dados não se resume apenas a isso. Alguns chegaram a defender o termo NoREL (*Not Relational*), o qual não foi muito recebido. Sintetizando o conceito, esse tipo de Banco de Dados não embute as ideias do modelo relacional e nem a linguagem SQL. Podemos citar que uma diferença fundamental entre

os dois modelos surge na criação de relacionamentos nos bancos de dados relacionais, os quais diferem dos bancos orientados a documentos, os quais não fornecem relacionamentos entre documentos. Os bancos de dados de documentos integram esses dados ao próprio documento.

Atualmente existem diversos Banco de Dados NoSQL, tais como os que foram apresentados na seção 2.1. O MongoDB é um SGBD voltado à alta performance, sem esquema fixo e orientado a documentos. Os documentos englobam formato BSON (*Binary JSON*), semelhante ao JSON, possibilitando que a estrutura e os campos do documento variem de uns para os outros ao longo do tempo.

O BSON é uma extensão do JSON, codificando de forma binária os documentos. Três características do formato BSON são: i) leveza; ii) transportabilidade; e iii) eficiência. Cada documento BSON permite a cada atributo possuir um tipo especifico de valor, seguindo a Tabela 2.2.

Tabela 2.2. Formatos suportados pelo BSON.

Tipo	Número	Apelido	Nota
Real	1	“double”	
Texto	2	“string”	
Objeto	3	“object”	
Vetor	4	“array”	
Dado binário	5	“binData”	
Indefinido	6	“undefined”	Depreciado
ID Objeto	7	“objectId”	
Booleano	8	“bool”	
Data	9	“date”	
Nulo	10	“null”	
Expressão regular	11	“regex”	
DBPointer	12	“dbPointer”	Depreciado
JavaScript	13	“javascript”	
Símbolo	14	“symbol”	Depreciado
JavaScript com escopo	15	“javascriptWithScope”	
Inteiro de 32-bit	16	“int”	

TimeStamp	17	“timestamp”	
Inteiro de 64-bit	18	“long”	
Decimal128	19	“decimal”	A partir da 3.4
Chave mínima	-1	“minKey”	
Chave máxima	127	“maxKey”	

O MongoDB é um banco de dados distribuído, com escalabilidade horizontal focada na distribuição geográfica, fornecendo mecanismos para consultas *ad hoc*, os quais contemplam indexação e agregação em tempo real. Ele possui uma API (*Application Programming Interface*) própria para o gerenciamento das transações no banco, onde geralmente os dados são manipulados em memória RAM, buscando maior velocidade nas operações de uma aplicação CRUD (*Create, Read, Update* e *Delete*). Tal API está disponível para o Java em seu repositório no GITHUB[18].

O MongoDB versão *Community* está disponível para instalação nos principais sistemas operacionais, sendo eles o Debian 7+, Ubuntu 12.04+, Red Hat/CentOS 6+, OS X 10.7+ e o Windows 7/Server 2008 R2+. O mesmo também possui drivers para as principais linguagens de programação do mercado, tais como o C, C++, Java, C#, Python, PHP, entre outras.

Atualmente, conforme pesquisa do StackOverflow 2017[19], o banco de dados MongoDB foi o mais votado dos bancos NoSQL em utilização, ocupando a 5ª posição do ranking geral de banco de dados. Nesta mesma pesquisa o banco ocupou a 3ª e 1ª posição no ranking, o qual considera as categorias de banco de dados mais amados e mais desejados respectivamente. Isso demostra o interesse da comunidade pelo banco MongoDB e como o mesmo vem crescendo em utilização e público.

A versão atual, estável, é a 3.6. Para a versão 4.0 Horowitz (2018) anunciou que o banco de dados adicionará suporte a transações com vários documentos, tornando-o o único banco de dados a combinar velocidade, flexibilidade e poder do modelo de documento com garantias ACID. Essa modificação trará ao banco mais familiaridade com os desenvolvedores que já estão acostumados a trabalhar com os tradicionais bancos de dados relacionais.

Na próxima seção abordaremos a instalação e preparação do servidor para uso.

[18] MongoDB Java Driver: Disponível em <https://mongodb.github.io/mongo-java-driver/>.

[19] Pesquisa StackOverflow 2017: Disponível em <https://insights.stackoverflow.com/survey/2017>.

2.5.2. Preparando o Ambiente

Ao abordarmos a utilização do MongoDB como banco de dados, é importante enfatizar que algumas configurações devem ser feitas durante e após a sua instalação.

A instalação do MongoDB pode ser realizada mediante ao *download* do mesmo em <http://www.mongodb.org/downloads>, devendo-se optar pela versão do sistema operacional em que será utilizado o SGBD, ou ainda via 'apt', 'yum' ou 'brew' no caso das distribuições GNU/Linux ou Mac OS X.

Para o sistema operacional Windows é disponibilizado um instalador MSI. Já para o Mac OS X e GNU/Linux, temos como opção o SGBD compactado em TGZ, além das opções supracitadas (brew, apt ou yum). Mais informações sobre a forma de instalação para cada sistema operacional está disponível no endereço <https://docs.mongodb.com/master/administration/install-community/>. Nessas instalações são ofertados 5 pacotes, conforme demonstrado na Tabela 2.3.

Tabela 2.3. Pacotes que compõem o MongoDB.

Pacote	Descrição
mongodb-org	Um meta-pacote que instalará automaticamente os pacotes de quatro componentes listados abaixo.
mongodb-org-server	Contém o daemon mongod e os scripts de configuração e de inicialização.
mongodb-org-mongos	Contém o daemon mongos*.
mongodb-org-shell	Contém a interface de acesso ao mongo.
mongodb-org-tools	Contém as seguintes ferramentas do MongoDB: mongoimport bsondump, mongodump, mongoexport, mongofiles, mongorestore, mongostat e mongotop.

O *'mongos' é um serviço de roteamento para configurações de compartilhamento do MongoDB, que processa consultas da camada de aplicação e determina o local desses dados no *cluster* fragmentado. Da perspectiva do aplicativo, uma instância do 'mongos' se comporta de maneira idêntica a qualquer outra instância do MongoDB.

O SGBD é configurado via passagem de parâmetros na inicialização ou via arquivo de configuração (mongod.conf ou mongod.cfg, para GNU/Linux e OS X ou Windows). Nesse arquivo de configuração são especificados os valores para o destino dos logs do SGDB e onde os dados serão armazenados (dbpath), entre outras configurações possíveis. O mesmo ocorre para o serviço 'mongos'. O requisito básico que deve ser especificado na iniciação do 'mongod' (serviço do MongoDB) é o caminho

do banco. Uma especificação completa do arquivo de configuração do SGBD, pode ser encontrada em sua documentação[20].

Por padrão o SGBD utiliza a porta '27017' para comunicação, via *shell* ou por meio de alguma ferramenta IDE, como o 'Studio 3T'[21] ou o 'NoSQLBooster'[22], o qual será utilizado ao longo deste trabalho. Contudo, após a instalação devemos realizar as configurações de segurança para o administrador e criar os novos usuários e seus respectivos bancos.

O SGBD MongoDB, por padrão, não possui nenhum usuário ou senha para acesso aos bancos, o qual deve ser configurado logo após a instalação. Para isto, podemos acessar o SGBD utilizando a IDE 'NoSQLBooster', especificando a URI de conexão "mongodb://localhost:27017". Onde 'localhost' pode ser substituído pelo endereço IP do servidor, caso não seja a máquina local.

Para criarmos usuários no MongoDB, devemos utilizar a função 'db.createUser()'. Para a criação do usuário devemos informar o nome, a senha e os *roles* (papéis). Em *roles* devemos informar quais os privilégios para o usuário e em qual base de dados ele terá tais privilégios. Os *roles* possíveis para os privilégios em um usuário administrador de um *cluster* são apresentados a Tabela 2.4:

Tabela 2.4. *Roles* para administrador do *cluster*.

Papel	Descrição
clusterAdmin	É o maior privilégio para usuários administradores do *clusters*. Inclui todos os privilégios que serão listados abaixo e também possui o privilégio da ação de 'dropDatabase'.
clusterManager	Fornece o gerenciamento e monitoramento das ações no *cluster*. Um usuário com esse papel pode acessar as configurações de bases locais, que são usadas em réplicas e fragmentação.
clusterMonitor	Fornece apenas leitura para ferramentas de monitoramento no *cluster*.
hostManager	Permite monitorar e gerenciar os servidores.

Outras regras importantes são referentes ao gerenciamento das bases de dados, onde podemos destacar as regras apresentadas na Tabela 2.5.

[20] Documentação do MongoDB. Disponível em <https://docs.mongodb.com/manual/reference/configuration-options/>.
[21] Studio 3T. Disponível em <https://studio3t.com/>.
[22] NoSQLBooster. Disponível em <https://nosqlbooster.com/home>.

Tabela 2.5. *Roles* para administrador da base de dados.

Papel	Descrição
root	Fornece acesso às operações e a todos os recursos das funções 'readWriteAnyDatabase', 'dbAdminAnyDatabase', 'userAdminAnyDatabase', funções 'clusterAdmin', restauração e backup do sistema.
userAdminAnyDatabase	Define as mesmas permissões que o 'userAdmin', contudo aplicadas a qualquer base de dados.
dbAdminAnyDatabase	Fornece o mesmo acesso às operações de administração do banco de dados que o 'dbAdmin' para todas as bases de dados.
userAdmin	Permite a esses usuários criar e modificar configurações e determinar usuários para um banco de dados.
dbAdmin	Define que o usuário será o administrador de uma base de dados específica.
dbOwner	Define que o usuário será o proprietário da base de dados e terá permissão administrativa da mesma. Essa regra combina os privilégios concedidos pelas funções 'readWrite', 'dbAdmin' e 'userAdmin'.
readWrite	Permite ao usuário as operações de leitura e gravação na base de dados.
read	O usuário terá a permissão de somente leitura na base de dados.

Essas regras são definidas na base de dados "admin" do SGBD, sendo que uma lista completa contendo outras regras está disponível no site da documentação[23]. Para definirmos um usuário super-administrador para o SGBD, podemos especificá-lo com suas regras, conforme demonstrado na Figura 2.9.

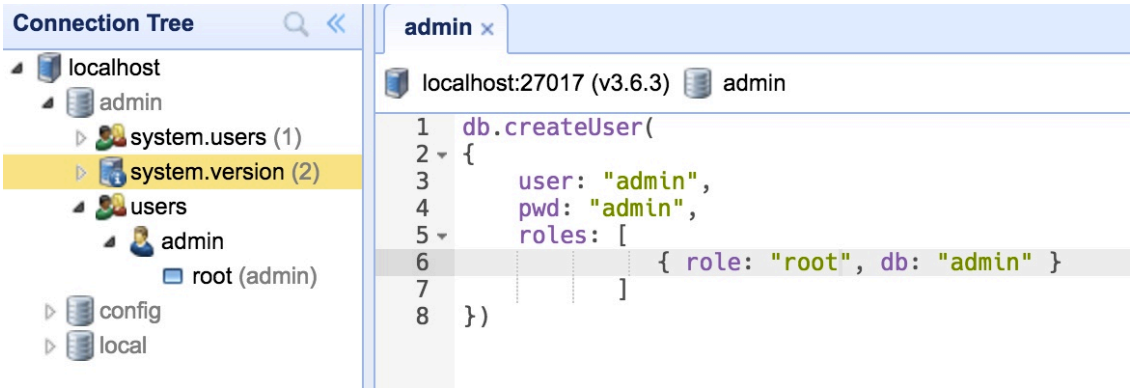


Figura 2.9. Usuário super-administrador do SGBD.

[23] Permissões do SGBD: Disponível em <https://docs.mongodb.com/manual/reference/built-in-roles/>.

Após configurarmos o super-administrador devemos criar um banco de dados e um usuário responsável por este banco, conforme demostra a Figura 2.10.

Figura 2.10. Usuário proprietário da base JAI.

Para que as novas regras de autenticação passem a ser aplicadas pelo SGBD, é necessário reiniciar o serviço do banco passando como parâmetro, nas configurações, a opção '--auth'. Assim o SGBD exigirá que qualquer operação seja autenticada por algum usuário que possua à devida permissão.

Além da opção para adição de usuários ao SGBD, temos as opções para remoção ('db.dropUser()' e 'db.dropAllUsers()'), gerência de permissões ('db.grantRolesToUser()', 'db.revokeRolesFromUser()' e 'db.auth()') e de atualizações dos usuários ('db.updateUser()', 'db.changeUserPassword()', 'db.getUser()' e 'db.getUsers()'). A utilização destas funções pode ser encontrada em <https://docs.mongodb.com/manual/reference/method/js-user-management/>.

Na próxima seção será iniciado o desenvolvimento de uma aplicação Java, onde serão demostrados as formas de conexão com o SGBD e os primeiros passos para manipulação do banco de dados.

2.5.3. Criando o Primeiro Exemplo

Mediante a configuração do SGBD MongoDB, criação de um usuário e banco de dados para utilização, chegamos no momento de criar o primeiro exemplo. Para este utilizaremos uma aplicação Java, pela qual será demonstrado como realizar a conexão com o SGBD, a modelagem do sistema e como fazemos a manipulação dos dados no Java, o qual está direcionado para o MongoDB.

A aplicação consistirá em um CRUD, demostrando a criação, leitura, atualização e deleção de dados em um banco de dados NoSQL (MongoDB). Essa aplicação consiste em armazenar dados de um autor, o qual terá os mesmos dados da Tabela 2.1, e de suas publicações. O sistema será composto de duas coleções, uma contendo os dados dos autores e outra contendo as publicações.

As conversões dos dados de entrada serão inicialmente modeladas de forma similar ao HashMap, utilizando o 'BasicDBObject', o qual converte os dados para BSON, antes de sua gravação no banco e posterior recuperação.

Para iniciar o desenvolvimento da aplicação são necessárias as APIs do MongoDB, conforme já mencionado, em especial a "bson", "mongodb-driver" e "mongodb-driver-core", atualmente na versão 3.6.3. Essas bibliotecas podem ser importadas para o projeto via Maven[24] ou manualmente.

Importadas as bibliotecas ao projeto podemos definir a conexão com a base de dados, conforme Figura 2.11.

```
MongoClientURI mongoURI = new MongoClientURI("mongodb://jai:jai@localhost:27017/?authSource=JAI");
MongoClient mongodb = new MongoClient(mongoURI);
MongoDatabase db = mongodb.getDatabase("JAI");
```

Figura 2.11. Conexão com o MongoDB.

Conforme a Figura 2.11, na primeira linha temos o URI (*Uniform Resource Identifier*) onde são passados o nome do usuário do banco (jai), a senha deste usuário (jai), o host onde está sendo executada a instancia do MongoDB (localhost), a porta de conexão com o SGBD (27017) e a base em que o usuário irá autenticar-se (JAI). A segunda linha da Figura 2.11 é responsável por realizar a conexão. Na última linha é definida a base de dados que será utilizada pela aplicação (JAI).

Um 'BasicDBObject' é um objeto no formato BSON, que pode conter uma única informação por campo, uma lista ou um outro objeto BSON. Um exemplo da construção desse tipo de objeto pode ser visto na Figura 2.12.

```
BasicDBObject pessoa = new BasicDBObject();
pessoa.put("nome", "Humberto");
pessoa.put("sobrenome", "Dalpra");
pessoa.put("email", "humbertodalpra@gmail.com");

BasicDBObject tel = new BasicDBObject();
tel.put("celular", Long.parseLong("32988330000"));
pessoa.put("telefone", tel);

BasicDBObject end = new BasicDBObject();
end.put("tipoLogradouro", "Rua");
end.put("logradouro", "B");
end.put("numero", "1");
end.put("bairro", "Centro");
end.put("cidade", "Juiz de Fora");
end.put("cep", "36100-000");

pessoa.put("endereco", end);
```

Figura 2.12. Objeto BSON.

Neste exemplo da Figura 2.12 tem-se um objeto BSON 'pessoa', o qual contém o nome, sobrenome, e-mail, telefone e endereço. Entretanto, o telefone e o endereço também são objetos BSON que compõem o objeto pessoa.

[24] Driver MongoDB para Java: Disponível em <https://mongodb.github.io/mongo-java-driver/>.

Cada objeto BSON é armazenado na sua devida coleção, onde são agrupados documentos similares. A especificação da coleção e o salvamento do objeto BSON (pessoa) criado são apresentados na Figura 2.13.

```
MongoCollection<BasicDBObject> colecaoPessoa = db.getCollection("Pessoa", BasicDBObject.class);
colecaoPessoa.insertOne(pessoa);
```

Figura 2.13. Especificação da coleção e salvamento do Objeto BSON.

Além do método 'insertOne', responsável por inserir um documento em uma coleção, temos métodos para remoção, deleção, busca e atualização, os quais podem operar sobre um único documento ou um conjunto de documentos, conforme será apresentado na próxima seção, onde também será explicada a manipulação de documentos do SGBD MongoDB.

2.5.4. Manipulando as Informações

Algumas definições são importantes para a manipulação de dados, como por exemplo os conceitos vinculados ao SGBD relacional e como são representados no SGBD NoSQL. Fazendo uma associação entre os SGBDs relacionais e os SGBDs NoSQL do tipo documento, temos uma representação, a qual é apresentada na Figura 2.14.

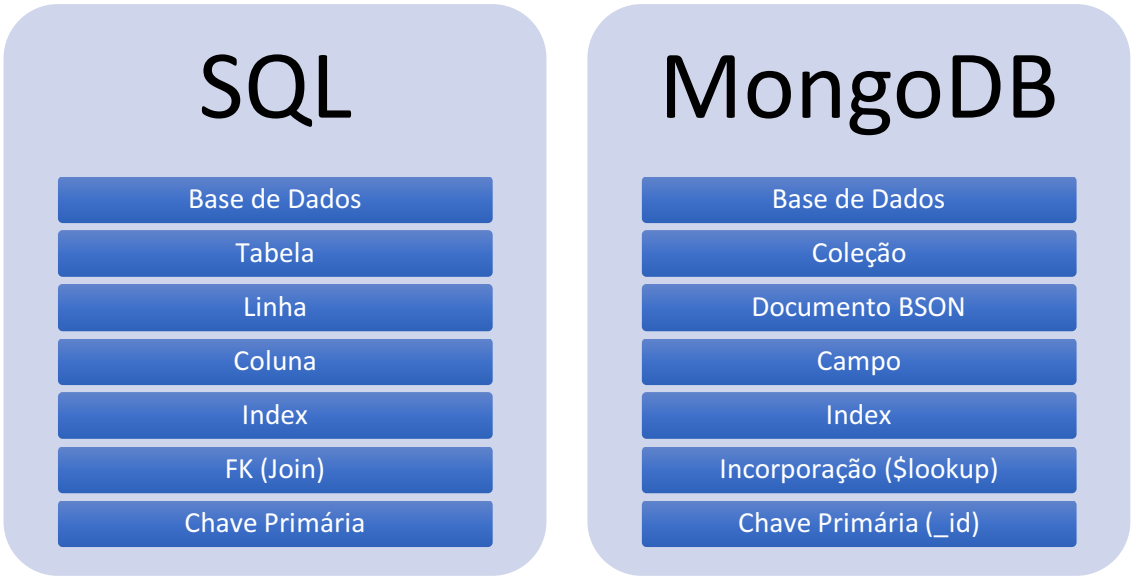


Figura 2.14. Conceitos SQL e NoSQL.

Nessa associação temos que o conceito de base de dados é igual a ambos os tipos de SGBD. Já o conceito de tabela dos SGBDs relacionais é substituído pelo de coleção, visto que ao invés de linhas dos dados temos agora documentos em formato BSON. Cada coluna passa a ser representada como um campo do documento BSON, que pode conter um único valor, uma lista de valores ou um objeto, conforme explicado na seção anterior.

Nos bancos de dados NoSQL, do tipo documento, não existe o conceito de chave estrangeira, contudo é possível realizar a incorporação de documentos entre diferentes

coleções, onde essa definição será explicada na seção 2.6.1 e, por fim, a chave primaria equivale ao campo '_id' do documento BSON.

Prosseguindo com o desenvolvimento da aplicação CRUD, a qual vem usando a linguagem de programação Java, abordaremos na sequência a manipulação dos dados em banco, considerando a persistência de um lote de documentos, a busca de um registro único e a busca de múltiplos registros, atualização de um registro possibilitando alteração de estrutura, entre outros métodos importantes. Uma lista dos principais métodos de manipulação de dados do SGBD MongoDB pode ser observado na Tabela 2.6.

Tabela 2.6. Métodos de manipulação de documentos.

Nome	Descrição
insertOne()	Insere um novo documento em uma coleção.
insertMany()	Insere um conjunto de novos documentos em uma coleção.
insert()	Cria um novo documento em uma coleção.
updateOne()	Modifica um único documento em uma coleção.
updateMany()	Modifica um conjunto de documentos em uma coleção.
update()	Modifica um documento em uma coleção.
deleteOne()	Exclui um único documento em uma coleção.
deleteMany()	Exclui um conjunto de documentos em uma coleção.
remove()	Exclui documentos de uma coleção.
find()	Executa uma consulta em uma coleção ou visão e retorna um cursor de objetos.
findOne()	Executa uma consulta e retorna um único documento.
findOneAndDelete()	Encontra um único documento e o apaga.
findOneAndReplace()	Encontra um único documento e o substitui.
findOneAndUpdate()	Encontra um único documento e o atualiza.

Demonstrando o uso de algumas dessas funções no CRUD Java, além do método 'inserOne', apresentado na seção anterior, o método 'find' permite listar uma coleção e retornar todos os seus documentos, caso não seja incorporada nenhuma condição. Seu uso pode ser visto na Figura 2.15.

```
MongoClientURI mongoURI = new MongoClientURI("mongodb://jai:jai@localhost:27017/?authSource=JAI");
MongoClient mongodb = new MongoClient(mongoURI);
MongoDatabase db = mongodb.getDatabase("JAI");
MongoCollection<BasicDBObject> colecaoPessoa = db.getCollection("Pessoa", BasicDBObject.class);

for (BasicDBObject doc : colecaoPessoa.find()) {
    System.out.println(doc);
}
```

Figura 2.15. Listagem de uma coleção no MongoDB.

Já o método 'updateOne' permite atualizar um único documento, onde esta atualização pode incluir ou não mudança em sua estrutura. Nesse método são passados dois parâmetros, um que atuará como uma *query* de busca e outro que serão os novos valores do Documento BSON. Um exemplo de atualização utilizando Java é apresentado na Figura 2.16.

```
MongoClientURI mongoURI = new MongoClientURI("mongodb://jai:jai@localhost:27017/?authSource=JAI");
MongoClient mongodb = new MongoClient(mongoURI);
MongoDatabase db = mongodb.getDatabase("JAI");
MongoCollection<BasicDBObject> colecaoPessoa = db.getCollection("Pessoa", BasicDBObject.class);

BasicDBObject query = new BasicDBObject();
query.put("nome", "João");

BasicDBObject pessoa = new BasicDBObject();
pessoa.put("nome", "João");
pessoa.put("sobrenome", "Silva");

BasicDBObject end = new BasicDBObject();
end.put("tipoLogradouro", "Av");
end.put("logradouro", "Cardoso");
end.put("numero", "1");
end.put("bairro", "Borboleta");
end.put("cidade", "Juiz de Fora");
end.put("cep", "36110-100");
pessoa.put("endereco", end);

BasicDBObject update = new BasicDBObject();
update.put("$set", pessoa);

colecaoPessoa.updateOne(query, update);
```

Figura 2.16. Atualização de um documento BSON em Java.

Nesse exemplo de atualização, observe que passou-se a *query* e no update indicou-se o '$set', que é um operador que substitui o valor de um campo pelo novo valor especificado. Por fim, uma aplicação CRUD deve conter um método de deleção de documentos, o qual pode ser visto na Figura 2.17. Para a remoção de um documento é especificado a condição que o documento deve possuir.

```
MongoClientURI mongoURI = new MongoClientURI("mongodb://jai:jai@localhost:27017/?authSource=JAI");
MongoClient mongodb = new MongoClient(mongoURI);
MongoDatabase db = mongodb.getDatabase("JAI");
MongoCollection<BasicDBObject> colecaoPessoa = db.getCollection("Pessoa", BasicDBObject.class);

BasicDBObject query = new BasicDBObject();
query.append("nome", "João");
colecaoPessoa.deleteOne(query);
```

Figura 2.17. Remoção de um documento BSON em Java.

Além das funções de manipulação de dados apresentadas no CRUD, temos as de manipulações de coleções e as de busca condicionadas, onde são especificados filtros para retorno dos resultados. Para apresentarmos essa manipulação de coleções e as buscas condicionadas, considera-se os dados da Tabela 2.7, representando uma tabela de um SGBD relacional e a Figura 2.18, representando os mesmos dados em BSON.

Tabela 2.7. Tabela 'Publicacoes' em um SGBD relacional.

IdPublicacoes	Autor	Artigo
1	Humberto & Tassio	Using Ontology and Data Provenance to Improve Software Processes
2	Tassio	A Software Framework for Data Provenance
3	Stonebraker	Newsql: An alternative to nosql and old sql for new oltp apps

```
/* 1 createdAt:17/04/2018 09:59:27*/
{
    "_id" : ObjectId("5ad5efaf90e8f81d50f67d00"),
    "artigo" : "Using Ontology and Data Provenance to Improve Software Processes",
    "autor" : [
        "Tassio",
        "Humberto"
    ]
},

/* 2 createdAt:17/04/2018 09:59:27*/
{
    "_id" : ObjectId("5ad5efaf90e8f81d50f67d01"),
    "artigo" : "A Software Framework for Data Provenance",
    "autor" : "Tassio"
},

/* 3 createdAt:17/04/2018 09:59:28*/
{
    "_id" : ObjectId("5ad5efb090e8f81d50f67d02"),
    "artigo" : "Newsql: An alternative to nosql and old sql for new oltp apps",
    "autor" : "Stonebraker"
}
```

Figura 2.18. Dados da coleção 'Publicacoes' em BSON.

A Tabela 2.8 apresenta uma associação entre as *querys* SQL e a mesma para o SGBD MongoDB, evidenciando as diferenças, onde a coluna SQL é referente a Tabela 2.7 e a coluna MongoDB à Figura 2.18.

Tabela 2.8. SQL e Declarações do MongoDB.

SQL	MongoDB
CREATE TABLE publicacoes (id INT NOT NULL AUTO_INCREMENT, autor Varchar(255), artigo Varchar(255), PRIMARY KEY (id))	db.createCollection("publicacoes")

ALTER TABLE publicacoes ADD data DATETIME	db.publicacoes.updateMany({ }, { $set: { data: new Date() } })
ALTER TABLE publicacoes DROP COLUMN data	db.publicacoes.updateMany({ }, { $unset: { "data": "" } })
DROP TABLE publicacoes	db.publicacoes.drop()
SELECT artigo FROM publicacoes	db.publicacoes.find({ }, { artigo: 1, autor: 0, _id: 0 })
SELECT * FROM publicacoes WHERE autor = "Stonebraker "	db.publicacoes.find({ autor: " Stonebraker " })
SELECT COUNT(*) FROM publicacoes	db.publicacoes.count()

As declarações apresentadas na Tabela 2.8 são manipulações básicas das coleções e dos documentos. Na seção 2.6 apresentam-se os operadores do método 'find', além dos métodos de agregação, agrupamento, ordenação, distinção e o uso de índices.

Na próxima seção aborda-se algumas considerações adicionais sobre SGBDs orientados a documentos, especificando como uma aplicação Java OO pode operar sobre documentos BSON.

2.5.5. Outras Considerações sobre Banco de Dados Orientados a Documentos

Os bancos de dados orientados a documentos são apenas um tipo de SGBD NoSQL e possuem vantagens e desvantagens, conforme apresentado de acordo com o teorema CAP.

Os documentos BSON criados a partir do 'BasicDBObject' podem ser criados também via objetos 'Map' do Java, antes de salvos convertidos. Tais objetos são mapeados onde cada chave possui seu respectivo valor, ou seja, através da chave é possível acessar o valor especificado, sendo que a chave não pode repetir-se, ao contrário do valor.

Considerando uma implementação Java OO (Orientada a Objeto) para o exemplo da Tabela 2.7, ao realizar a construção do objeto 'Publicações', representado na Figura 2.19, os valores são transformados em 'Map' (Figura 2.20) que por sua vez são passados para 'BasicDBObject' antes do salvamento (Figura 2.21). Um processo inverso ocorre ao recuperar os dados do banco em BSON e transforma-los em objetos Java (Figura 2.22).

```
List autor = new ArrayList();
autor.add("Tassio");
autor.add("Humberto");
Publicacoes pub = new Publicacoes("Using Ontology and Data Provenance"
        + " to Improve Software Processes", autor);
```

Figura 2.19. Instanciação de um objeto 'Publicacoes'.

```
public Map converterMap(Publicacoes pub) {
    Map mapPublicacao = new HashMap();
    mapPublicacao.put("artigo", pub.getArtigo());
    mapPublicacao.put("autor", pub.getAutor());
    return mapPublicacao;
}
```

Figura 2.20. Transformação do objeto em Map.

```
public void save(Map value) {
    BasicDBObject document = new BasicDBObject(value);
    dbCollection.save(document);
}
```

Figura 2.21. Transformação do Map para BasicDBObject.

```
public Publicacoes converterPublicacoes(DBObject dbo) {
    Publicacoes pub = new Publicacoes();
    pub.setId((ObjectId) dbo.get("_id"));
    pub.setAutor((List) dbo.get("autor"));
    pub.setArtigo(dbo.get("artigo").toString());
    return pub;
}
```

Figura 2.22. Transformação BSON para objeto.

Em Gosling *et al*. (2014) podem ser encontrados detalhes sobre orientação à objeto em Java e a manipulação de objeto Map. Esse processo de transformação de classes OO para o MongoDB e seu retorno, é uma solução para sistema já existente e que demanda a migração dos dados para um SGBD NoSQL. Um exemplo completo dessa implementação e do CRUD apresentado na seção anterior estão disponíveis no GITHUB no endereço <https://github.com/tassioferenzini/JAI >.

2.6. Operadores

O NoSQL ganha espaço no trabalho com dados não estruturados, oferecendo condições para trabalhar com grande volume de dados, de forma eficiente. Contudo, vale enfatizar que muitas das vezes são necessárias a realização de operações que exigem maior grau de conhecimento do administrador de banco de dados e as buscas, apenas pela chave, não são suficientes.

Nesse ponto, o administrador do banco deve trabalhar com operadores de comparação, resultados distintos (*distinct*), expressões regulares, operadores lógicos e unários, além de operações com texto, onde tais operadores são passados ao método 'find', o qual é responsável pela busca nos documentos. Alguns dos operadores mais comuns para manipulação de documentos BSON são demonstrados na Tabela 2.9.

Tabela 2.9. Operadores para manipulação de documentos BSON.

Operador	Explicação
$eq	Retorna os documentos que possuem valores iguais a um valor especificado. (=)

$gt	Retorna os documentos que possuem valores que são maiores que um valor especificado. (>)
$gte	Retorna os documentos que possuem valores que são maiores ou iguais que valor especificado. (>=)
$in	Retorna todo os documentos que possuem qualquer um dos valores especificados em um *array*.
$lt	Corresponde a valores que são menores que um valor especificado.
$lte	Corresponde a valores que são menores ou iguais que um valor especificado.
$and	Junta cláusulas de consulta com um AND lógico e retorna todos os documentos que correspondem às condições de ambas as cláusulas.
$not	Inverte o efeito de uma expressão de consulta e retorna os documentos que não correspondem à expressão de consulta.
$nor	Junta-se a cláusulas de consulta com um NOR lógico e retorna todos os documentos que não correspondem a ambas as cláusulas.
$or	Junta cláusulas de consulta a um OU lógico e retorna todos os documentos que correspondem às condições de qualquer uma das cláusulas.
$exists	Retorna documentos que possuem o campo especificado.

Além do método 'find', esses operadores também podem ser utilizados em métodos para documentos distintos ou agrupamentos, os quais serão apresentados na próxima seção, junto com o uso destes. Além dos operadores supracitados estão disponíveis outros para uso e podem ser encontrados com suas descrições em <https://docs.mongodb.com/manual/reference/operator/>.

2.6.1. Manipulações Avançadas

Como explicado anteriormente, o objetivo desta seção é apresentar os operadores e como podemos realizar buscas avançadas do NoSQL, onde também far-se-á um comparativo com o SQL tradicional.

Buscas avançadas são comuns em todos os tipos de aplicações, seja por meio de operadores lógicos ou matemáticos, ou para manipulação de texto, como o tradicional "*like*" do SQL. O MongoDB possui alguns métodos específicos para consultas avançadas, onde tais métodos podem ser vinculados a outros para a busca, distinção, agrupamento, agregação ou ordenação de documento, conforme já mencionado. Na Tabela 2.10 apresentam-se exemplos desses métodos e será feita uma associação com o SQL.

Tabela 2.10. Consultas SQL e os respectivos métodos correspondentes para o MongoDB.

SQL	MongoDB
SELECT * FROM publicacoes WHERE autor like "Ta%"	db.publicacoes.find({ autor : /^Ta/ })

SELECT * FROM publicacoes WHERE autor = "Tassio" ORDER BY artigo ASC	db.publicacoes.find({ autor: "Tassio" }).sort({ artigo: 1 })
SELECT * FROM publicacoes LIMIT 1	db.publicacoes.find().limit(1) ou db.publicacoes.findOne()
SELECT * FROM publicacoes LIMIT 2 SKIP 1	db.publicacoes.find().limit(2).skip(1)
SELECT DISTINCT(autor) FROM publicacoes	db.publicacoes.distinct("autor")
SELECT autor FROM publicacoes GROUP BY autor	db.publicacoes.aggregate([{ $group : { _id : { autor: "$autor"}}}])
SELECT * FROM pessoas pes INNER JOIN publicacoes pub ON pes.nome=pub.autor	db.Pessoa.aggregate([{ $lookup: { from: "Publicacoes", localField: "nome", foreignField: "autor", as: "Artigos" } }])

Um método que vale destaque é o ‘aggregate’, que serve tanto para agrupar documentos por um campo especifico, como para relacionar documentos de coleções diferentes. Essa agregação entre coleções trabalha de forma similar ao JOIN do SQL, onde os documentos agregados passam a compor um novo campo no BSON retornado pelo SGBD. Na Figura 2.23 temos uma agregação entre as coleções “Pessoa” e “Publicacoes”, onde a chave ‘nome’ da tabela “Pessoa” se relaciona com a chave ‘autor’ da tabela “Publicacoes”. Para cada documento de entrada, o ‘$ lookup’ adiciona um novo campo de matriz cujos elementos são os documentos correspondentes da coleção agregada. O resultado dessa agregação pode ser visto na Figura 2.24.

O método ‘aggregate’ também possui como opções o ‘$match’ e o ‘$project’, onde o ‘$match’ filtra os documentos para retornar somente os documentos que correspondem à condição definida, enquanto o ‘$project’’ retorna os documentos com os campos solicitados, sendo que estes campos podem existir nos documentos ou serem criados na agregação, conforme pode-se visualizar na Figura 2.25.

```
db.Pessoa.aggregate([{
    $lookup: {
    from: "Publicacoes",
    localField: "nome",
    foreignField: "autor",
    as: "Artigos"
    }

}])
```

Figura 2.23. Agregação entre coleções.

```
    "_id" : ObjectId("5ad5efaf90e8f81d50f67cfd"),
    "nome" : "Tassio",
    "sobrenome" : "Sirqueira",
    "email" : "tassio@tassio.eti.br",
    "telefone" : {
        "casa" : NumberLong("3232730000"),
        "celular" : 984440000
    },
    "endereco" : {
        "tipoLogradouro" : "Rua",
        "logradouro" : "A",
        "numero" : "1",
        "bairro" : "Centro",
        "cidade" : "Matias Barbosa",
        "cep" : "36120-000"
    },
    "Artigos" : [
        {
            "_id" : ObjectId("5ad5efaf90e8f81d50f67d00"),
            "artigo" : "Using Ontology and Data Provenance to Improve Software Processes",
            "autor" : [
                "Tassio",
                "Humberto"
            ]
        },
        {
            "_id" : ObjectId("5ad5efaf90e8f81d50f67d01"),
            "artigo" : "A Software Framework for Data Provenance",
            "autor" : "Tassio"
        }
    ]
},
```

Figura 2.24. Resultado da agregação entre coleções.

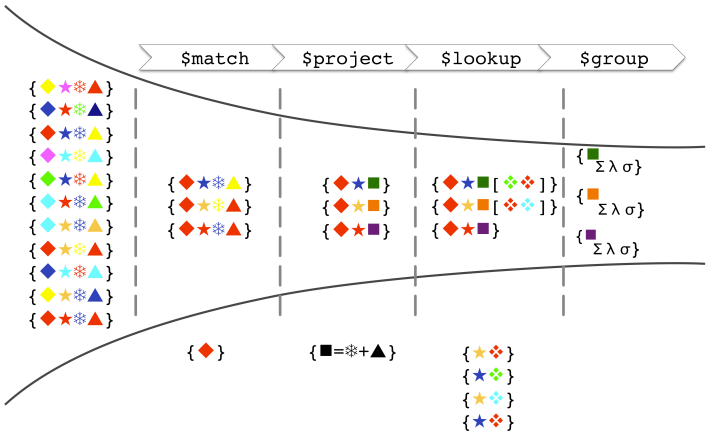


Figura 2.25. Detalhes do método 'aggregate'.

Além dos métodos apresentados, o MongoDB possui duas ferramentas para exportação e importação de dados para o SGBD, sendo estas conhecidas como 'mongoexport' e 'mongoimport', respectivamente.

O 'mongoexport' é um utilitário do SGBD que permite exportar os dados armazenados em uma instância do MongoDB, em JSON ou CSV. Já o utilitário 'mongoimport' faz o papel inverso, importando o conteúdo de arquivos JSON ou CSV para o SGBD. As figuras 2.26 e 2.27 representam respectivamente os processos de exportação e importação de uma coleção do SGBD.

```
mongoexport --db JAI --collection Pessoa --out pessoa.json;
```

Figura 2.26. Exportação de dados no MongoDB.

```
mongoimport --db JAI --collection Pessoa --file pessoa.json
```

Figura 2.27. Importação de dados no MongoDB.

2.6.2. Otimização

Quando trabalha-se com banco de dados relacionais o uso de índices são frequentes, principalmente em tabelas com grande quantidade de informações. Em bancos NoSQL o uso de índices não é diferente.

A indexação de um banco permite otimiza-lo, apresentando as consultas de forma mais rápidas e reduzindo a carga de processamento do servidor, além de garantir que uma chave não esteja duplicada. Contudo, cuidados devem ser adotados pois, pode haver o aumento do tempo de embutir no banco, aumento do consumo de memória para armazenamento em disco e para manipulação. O administrador do banco deve preocupar-se em manter o banco em estado consistente.

Os índices suportam a execução eficiente de consultas, ou seja, em uma coleção de dados sem índices é necessário verificar todos os documentos, para selecionar os documentos que correspondem à instrução de consulta. Os índices são estruturas de dados especiais que armazenam uma pequena parte do conjunto de dados da coleção em um formato fácil de percorrer.

No MongoDB o índice armazena o valor de um campo específico ou conjunto de campos, ordenado pelo valor do campo. A ordenação das entradas de índice suporta correspondências de igualdades eficientes e operações de consulta baseadas em intervalo. Além disso, o MongoDB pode retornar resultados classificados usando a ordenação no índice.

O SGBD oferece alguns métodos para manipulação de índices, conforme pode-se observar na Tabela 2.11, que podem operar sobre uma única chave ou múltiplas chaves.

Tabela 2.11. Manipulação de índices no MongoDB.

Método	Descrição
createIndex()	Cria um índice em uma coleção.
createIndexes()	Cria um ou mais índices em uma coleção.
reIndex()	Recria todos os índices existentes em uma coleção.
getIndexes()	Retorna um vetor que descrevem os índices existentes em uma coleção.
dropIndex()	Remove um índice específico em uma coleção.
dropIndexes()	Remove todos os índices de uma coleção.

Conforme mencionado, a criação de índices pode ser sobre uma chave específica dos documentos de uma coleção ou sobre múltiplas chaves, conforme apresenta as figuras 2.28 e 2.29 respectivamente, onde o valor ‘1’ especifica que o índice será ascendente e o ‘-1’ descendente.

```
db.getCollection("Pessoa").createIndex({ "endereco.cep": 1 })
```

Figura 2.28. Importação de dados no MongoDB.

```
db.getCollection("Pessoa").createIndex({ "endereco.cep": 1, "endereco.numero": -1 })
```

Figura 2.29. Importação de dados no MongoDB.

Um índice no MongoDB pode ser geoespacial, texto, único, parcial ou esparso, onde detalhes podem ser consultados na documentação[25]. Além dos índices, o SGBD oferece o método 'mapReduce'. O Map-reduce é um paradigma de processamento de dados para condensar grandes volumes de dados em resultados agregados, conforme exibe-se na Figura 2.30.

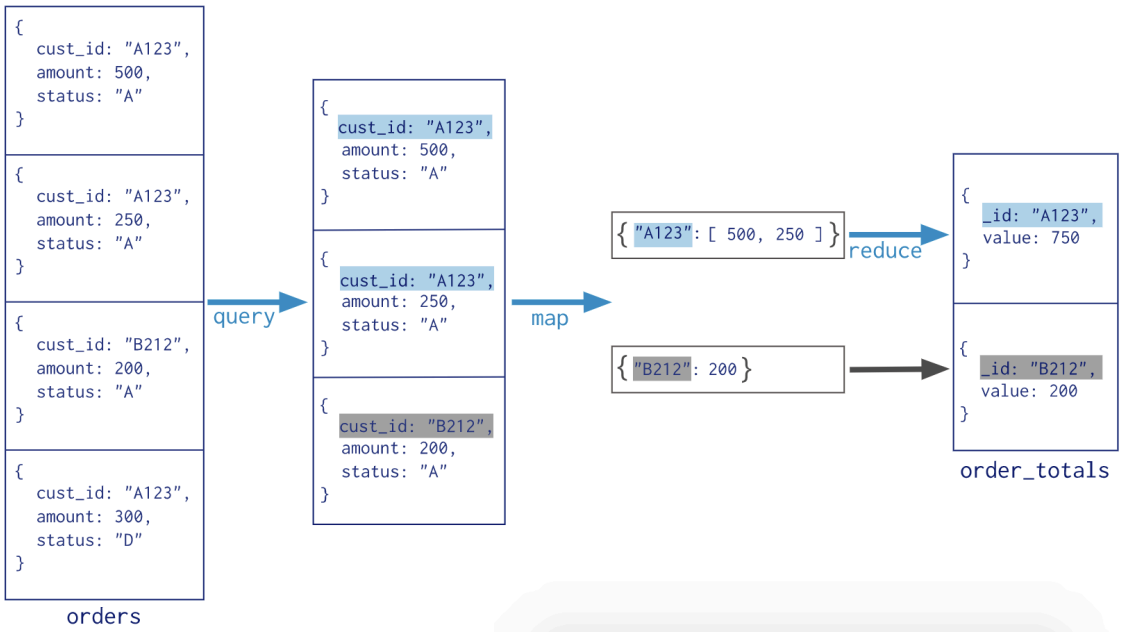


Figura 2.30. MapReduce no MongoDB.

Para a execução de um 'mapReduce', deve-se criar uma função Map, responsável por mapear os objetos que serão agrupados, e uma função de redução, a qual definirá o resultado esperado pela redução. Um exemplo do método 'mapReduce' para a tabela 2.1 retornando as cidades e quantas vezes elas aparecem na coleção 'Pessoa' é apresentado na Figura 2.31.

```
var map = function(){
  emit(this.endereco.cidade, 1);
}
var reduce = function(key, values){
  var res = 0;
  values.forEach(function(v){ res += 1});
  return {count: res};
}
db.Pessoa.mapReduce(map, reduce, { out: "map_cidade" });
```

Figura 2.31. Exemplo de MapReduce no MongoDB.

[25] Índices no MongoDB: Disponível em <https://docs.mongodb.com/manual/indexes/ - index-types>.

2.7. O Futuro do NoSQL

Os bancos NoSQL estão mais evoluídos e a "guerra fria" entre NoSQL vs. SQL está ficando antepassada. Atualmente aplicações tendem a ser "poliglotas" e buscar o melhor, vindo a utilizar a respectiva tecnologia de banco de dados que adequa-se a necessidade da aplicação.

Os bancos criados utilizando NoSQL não foram desenvolvidos em substituição aos bancos relacionais, haja vista que os mesmos trabalham com dados em que os relacionais tendem a apresentar gargalos. Todavia os bancos NoSQL precisam amadurecer em alguns quesitos, já sendo notórias algumas contribuições.

Atualmente os SGBDs NoSQL estão ganhando frente entre os produtos de bancos de dados mais usados mundialmente, o que demonstra que o mesmo vem sendo adotado em massa por especialistas em TI e, portanto, pode ser também uma boa opção para o projeto sendo desenvolvido atualmente.

Segundo a pesquisa desenvolvida em 2017 pelo Stack Overflow[26], de 29452 usuários que responderam sobre qual SGBD tem utilizado em seus projetos, 21% responderam o MongoDB, sendo esse o SGBD o mais bem posicionado na pesquisa entre os SGBD NoSQL, ocupando a 5ª colocação.

Nessa mesma pesquisa, o MongoDB ocupou a 3ª colocação entre os SGBD mais amados pelos desenvolvedores, atrás do Redis (SGBD NoSQL) e do PostgreSQL. Além disso, foi apontado como o SGBD mais desejado pelos desenvolvedores, conforme Figura 2.32.

Na pesquisa do ano anterior feita pelo Stack Overflow, o Redis era o banco de dados mais amado, o que significa que, proporcionalmente, mais desenvolvedores queriam continuar trabalhando com ele do que qualquer outro banco de dados. Com o apontamento dos desenvolvedores em indicar o MongoDB em 2017 como o mais desejado, caracteriza que para este ano de 2018 esses resultados mudem.

O ponto principal desta pesquisa é o avanço dos SGBDs NoSQL, que até então não despontavam. Isso representa uma quebra de paradigmas, é a mudança de visão dos profissionais de TI em busca de novas soluções no campo da engenharia do conhecimento.

[26] Pesquisa Stack Overflow: Disponível em <https://insights.stackoverflow.com/survey/2017>.

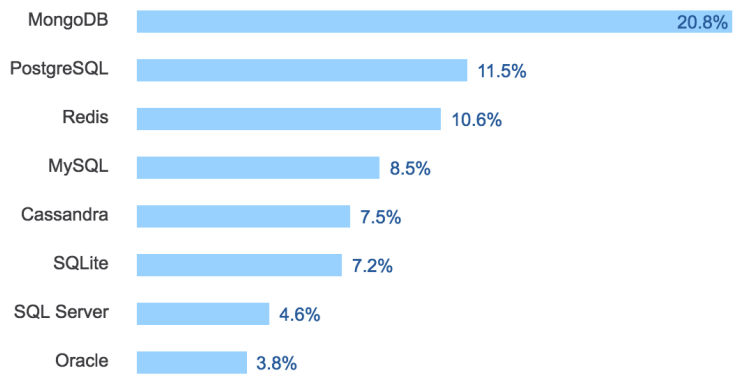


Figura 2.32. SGBD mais desejado pelos desenvolvedores.

Cabe ressaltar que SGBDs NoSQL não são para substituir os SGBDs relacionais, conforme já citado, e também não é a solução de todos os problemas. As características do NoSQL tornam-no atrativo na resolução de alguns problemas. Na próxima seção abordaremos a aplicabilidade dos SGBD NoSQL e quando essa tecnologia de banco de dados deve ser aplicada.

2.7.1. Aplicabilidade do NoSQL nos problemas do Mundo Real

Atualmente diversas áreas já abordam grandes volumes de dados, podendo citar-se como exemplo empresas do ramo financeiro, logística, saúde e redes sociais. Várias empresas incorporadas a estes ramos já vêm investindo em banco de dados NoSQL, sendo essa necessidade decorrente de grande volume de dados, falta de estrutura nos dados, descentralidade ou estabilidade dos dados persistidos.

Atualmente, o volume de dados de certas organizações, vem expandindo-se rapidamente, onde a utilização dos SGBDs relacionais tem mostrado-se muito problemática e não tão eficiente.

Dentre os principais problemas na utilização do Modelo Relacional estão a dificuldade de conciliar o tipo de modelo com a demanda da escalabilidade, a qual é cada vez mais frequente. Afim de exemplificar, suponhamos que o sistema está trabalhando sobre um SGBD relacional, onde houve um crescimento de usuários, o que culmina em uma queda de performance. Para resolver este problema, tende-se a fazer um upgrade no servidor ou aumentar o número destes.

A solução apresentada só funcionaria caso o número de usuários estagnasse ou crescesse vagarosamente, haja vista que o problema passa a se concentrar no acesso à base de dados. Neste caso, solucionar-se-ia o problema aumentando o poder do servidor, mediante ao aumento da memória, processador e armazenamento. Esta solução é chamada de Escalabilidade Vertical. Também poder-se-ia aumentar o número de máquinas no servidor web, onde esta alternativa é denominada Escalabilidade Horizontal, conforme explicado no teorema CAP.

Ao observar-se o intenso volume de dados e sua natureza estrutural, os desenvolvedores notam a dificuldade de se organizar os dados no Modelo Relacional sobre sistemas distribuídos. Neste ponto, o foco das soluções não-relacionadas está direcionado.

No intuito de solucionar diversos problemas relacionados à escalabilidade, performance e disponibilidade, projetistas do NoSQL trabalharam em uma alternativa de alto armazenamento, com velocidade e grande disponibilidade, abstraindo certas regras e estruturas que norteiam o Modelo Relacional.

A proposta dos bancos NoSQL não visa extinguir o Modelo Relacional, mas sim utilizá-lo em casos onde é necessária uma maior flexibilidade na estruturação do banco e melhor desempenho. Podemos utilizar um banco NoSQL e um relacional em conjunto, onde o SGBD NoSQL é indicado para que sistema que necessitem de alta disponibilidade ou escalabilidade e os dados que não necessitam dessa abordagem, podem ir para uma base relacional. Dessa forma temos uma utilização muito mais eficiente e econômica dos recursos.

Essa abordagem de um SGBD NoSQL foi utilizada com sucesso em uma plataforma de análise de dados econômicos, onde são incorporados ao sistema aproximadamente 27 milhões de documentos (registros) por mês, fazendo que o armazenamento, a escalabilidade e o desempenho levassem a plataforma a depender de uma solução alternativa aos clássicos SGBDs relacionais.

Esse mesmo problema vêm sendo enfrentado por uma aplicação médica, onde dados são coletados e apresentados em tempo real, e devem ser armazenados para histórico médico. Nesse caso, o principal problema é a estrutura dos dados, pois são provenientes de um equipamento com sensores ligados ao paciente, e cada equipamento transmite os dados conforme os sensores que estão ligados ao paciente, tornando a composição dos dados dinâmica.

Algumas empresas como Google, Facebook, GitHub e Globo já vem utilizando soluções NoSQL em seus produtos e como apontado pelo Stack Overflow, o crescimento e adoção desses SGBDs NoSQL são necessários em diversos cenários para solução de problemas computacionais.

Atualmente são os principais SGBD quando o assunto é *Big Data*, o que obrigas os profissionais da área a se atualizarem para esta nova tecnologia. Manipular dados considerados do *Big Data* já dissemina uma área própria dentro da engenharia de dados, chamada de '*Data Science*'.

Conforme abordado por Matos (2016), a Ciência de Dados (*Data Science*) é o domínio científico dedicado à descoberta de conhecimento através da análise de dados, onde Cientistas de Dados utilizam técnicas matemáticas e algoritmos para encontrar soluções de problemas de negócio ou científico.

Engenheiros de Dados que não atentarem as essas mudanças se limitarão a problemas triviais. O propósito do Engenheiro de Dados, é fornecer soluções que possam enriquecer os dados por ele gerenciado.

Na próxima seção abordaremos o perfil dos novos profissionais que atuarão nesta área.

2.7.2. O Novo Perfil dos Profissionais da Computação

Conforme supracitado, o NoSQL representa uma mudança de paradigma e, atualmente, grande parte dos cursos de computação ainda seguem a tradição de apresentarem apenas bancos de dados relacionais, sua modelagem e o padrão SQL.

Mediante ao crescimento do NoSQL é necessário compreender essa filosofia, frente as novas demandas do mercado e a necessidade de profissionais que possuam esse novo conhecimento. Esse novo perfil de profissional, o qual conhece banco de dados NoSQL, implica na atualização das grades dos cursos de computação e na atualização de centenas à milhares de profissionais que já atuam com banco de dados relacional.

Atentando mais especificamente em Engenharia de Software, a qual encontra-se na área de Software e Serviços, a descrição do profissional desejado compreende especificações entre a visão, o papel e o estilo de vida. A visão engloba criação, teste, instalação e manutenção do software, visando novos sistemas de informação para o mercado. A inteligência tecnológica em quaisquer dos ambientes de desenvolvimento e domínios de aplicação também é vital, desde que a habilidade de captar as necessidades dos clientes seja tão importante.

Já no papel vislumbra-se que o Engenheiro de Software desenhe, construa, teste, implemente e mantenha aplicações que atendam necessidades específicas do cliente. As aplicações mencionadas incluem aplicações para empresas, aplicações de software embebido, tais como para dispositivos móveis e planejamento de recursos de empresas, sistemas em ambiente de negócios e industrial. Conhecimento sobre a interação humano/computador é também parte do papel a ser englobado pelo Engenheiro de Software, o qual tende a envolver a psicologia humana, ergonomia, assim como desenvolvimento de aplicações.

Quanto ao estilo de vida, na maioria dos casos o trabalho será constituído a cabo em equipe, sendo possível que equipes trabalhem em múltiplos sítios e comuniquem-se através de dispositivos de comunicação. Inicialmente à atividade, é necessário obter características técnicas com o resto da equipe, sendo necessário, ao passar do tempo, um maior envolvimento com os negócios e ambiente dos clientes, afim de demonstrar e implementar as aplicações e soluções.

Os engenheiros de software que irão trabalhar com o sistema para o *Big Data* ou com sistemas que demandem o uso de um SGBD NoSQL, precisam conhecer essa

tecnologia para terem condições de modelar o sistema seguindo esse novo paradigma e saberem também, quando um projeto irá demandar uma solução flexível e escalável, evitando assim que projetos fracassem ou ultrapassem o orçamento por conta de reengenharia.

Já os engenheiros de dados são os responsáveis por manter essa estrutura de dados, garantindo a escalabilidade, consistência dos dados e fornecendo mecanismos para os analistas de dados extraírem informações úteis das bases, o que não é uma tarefa trivial frente ao *Big Data*.

Para que soluções envolvendo o uso de SGBDs NoSQL deem certo, é necessário além da comunicação entre os engenheiros de dados e de software, que ambos estejam alinhados com essa tecnologia, buscando extrair o melhor de cada SGBD para cada solução provida. Profissionais que não se atualizarem, ficaram limitados a sistemas triviais.

2.8. Considerações Finais

O objetivo desse minicurso é apresentar, de modo teórico e prático, o uso de NoSQL e as mudanças que envolvem a engenharia de dados e a engenharia de software para o uso dessa nova tecnologia. Na parte prática foi apresentado como modelar e construir aplicações em Java utilizando o MongoDB, buscando alinhamento da teoria à prática.

A alteração do uso de um banco de dados relacional para um banco de dados não-relacional é desafiadora, sendo necessário um estudo profundo sobre os bancos de dados não relacionais que vêm a cumprir todas as demandas de um dado usuário e/ou desenvolvedor, dado que cada solução demanda suas próprias características.

Criar um banco de dados não relacional tende a fornecer os mesmos recursos e operações de consulta abrangidos por um banco de dados relacional, sendo que em geral a performance é melhor. A utilização do banco de dados relacional não tende a ser findada, haja vista que o mesmo fornece um conjunto de recursos incomparável, tal como a integridade e a confiabilidade dos dados, motivo pelos quais são amplamente utilizados.

O NoSQL é uma tecnologia que vem divulgando-se no mercado, principalmente quando trata-se de *Big Data*. Os profissionais da área de computação devem ser capazes de trabalhar com a integração entre banco de dados relacional e banco de dados NoSQL (não relacional), em busca das melhores soluções.

O MongoDB é um banco de dados NoSQL bastante popular, assim como a linguagem de programação Java. Contudo, existem diversos SGBDs NoSQL além do MongoDB, voltados para diferentes soluções e atualmente disponíveis para uso pelas principais linguagens de programação.

Nesse minicurso o intuito é abranger um público que posteriormente possa vir a utilizar outro SGBD NoSQL, assim como outras linguagens de programação como o Python, que é bastante comum em aplicação que utiliza o SGBD NoSQL.

Ao analisar os dados imputados neste trabalho, observa-se um melhor desempenho do MongoDB e uma modelagem fora do tradicional. É importante ressaltar que isso não implica em uma superioridade total do MongoDB, porém demonstra algumas das funções atendidas por este.

Esse trabalho apresenta uma visão geral da área e da integração entre a engenharia de dados e de software, para soluções de problemas do *Big Data* ou para sistemas especializadas. Esperamos que em breve o ensino de SGBD NoSQL seja disciplina básica entre os cursos de graduação, assim como é hoje com os SGBDs relacionais, capacitando os novos profissionais da computação, para a resolução dos desafios do *Big Data*.

Referências

_____. (2018). MongoDB Docs. Disponível em: <https://docs.mongodb.com/>. Acessado em: 19 de abr. de 18.

_____. Engenharia de Software, Banco de Dados e Interação Humano Computador. Disponível em: <http://ppgcc.dc.ufscar.br/pesquisa/linhas-de-pesquisa/engenharia-de-software>. Acessado em: 26 de fev. de 18.

Amaral, F. (2016). Introdução à Ciencia de Dados: mineração de dados e big data. Alta Books Editora.

Brewer, E. A. (2000, July). Towards robust distributed systems. In PODC (Vol. 7).

Cattell, R. (2011). Scalable SQL and NoSQL data stores. Acm Sigmod Record, 39(4), 12-27.

Date, C. J. (2004). Introdução a sistemas de bancos de dados. Elsevier Brasil.

Dhar, V. (2013). Data science and prediction. Communications of the ACM, 56(12), 64-73.

Elmasri, R., Navathe, S. B., & Pinheiro, M. G. (2005). Sistemas de banco de dados.

Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2014). The Java Language Specification, Java SE 8 Edition (Java Series).

Grolinger, K., Higashino, W. A., Tiwari, A., & Capretz, M. A. (2013). Data management in cloud environments: NoSQL and NewSQL data stores. Journal of Cloud Computing: advances, systems and applications, 2(1), 22.

Gubbi, J., Buyya, R., Marusic, S., & Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. Future generation computer systems, 29(7), 1645-1660.

Horowitz, E. (2018). Multi-document transactions in MongoDB. Disponível em: <https://www.mongodb.com/blog/post/multi-document-transactions-in-mongodb>. Acessado em: 15 de abr. de 18.

Leavitt, N. (2010). Will NoSQL databases live up to their promise?. Computer, 43(2).

Lopes, T., Cavaleiro, J., Rocha, J., Rodrigues, P. (2005). Perfis, Papéis e Competências em Engenharia de Software. FEUP. Universidade do Porto. Portugal.

Matos, D. (2017). Cientista de Dados x Engenheiro de Dados. Disponível em: <http://www.cienciaedados.com/cientista-de-dados-x-engenheiro-de-dados/>. Acessado em: 26 de fev. de 18.

Mayer-Schonberger, V., & Cukier, K. (2014). Big data: como extrair volume, variedade, velocidade e valor da avalanche de informação cotidiana (Vol. 1). Elsevier Brasil.

Paniz, D. (2016). NoSQL: Como armazenar os dados de uma aplicação moderna. Editora Casa do Código.

Sommerville, I. (2011). Engenharia de software. 9 ed. Pearson Prentice Hall.

Strauch, C., Sites, U. L. S., & Kriha, W. (2011). NoSQL databases. Lecture Notes, Stuttgart Media University, 20.

Capítulo 3

Protocolos de Aplicação para a Internet das Coisas: conceitos e aspectos práticos

Alexandre Sztajnberg, Roberto da Silva Macedo e Matheus Stutzel

Abstract

Regardless of the use of frameworks, middleware and communication technologies, the interaction between applications, services and devices in the Internet of Things is mediated by Application Protocols. The use of these protocols is facilitated by libraries, and coding is usually similar to known mechanisms, such as sockets. In this chapter we present an overview of the main Application Protocols currently available, HTTP/REST, CoAP, MQTT, AMQP and XMPP. Also, the differences between the protocols and applications for which they are most indicated are discussed. The WebSocket mechanism is also introduced to show how the protocols can be used from browsers. Examples and their execution environments illustrate the presented protocols. A more structured example completes the chapter.

Resumo

Independentemente do uso de frameworks, middleware e tecnologias de comunicação, a interação entre aplicações, serviços e dispositivos na Internet das Coisas é mediada por Protocolos de Aplicação. O uso destes protocolos é facilitado por bibliotecas, e a codificação geralmente semelhante à mecanismos conhecidos, como sockets. Neste capítulo é apresentada uma visão dos principais Protocolos de Aplicação atualmente disponíveis, HTTP/REST, CoAP, MQTT, AMQP e XMPP. Também, são discutidas as diferenças entre os protocolos e aplicações para os quais são mais indicados. O mecanismo de WebSocket é também introduzido para mostrar como os protocolos podem ser usados a partir de navegadores. Exemplos e seus ambientes de execução ilustram os protocolos apresentados. Um exemplo mais estruturado completa o capítulo.

3.1. Introdução

A Internet das Coisas (*Internet of Things*, IoT) promete facilitar a interação automática e o acesso a dispositivos industriais, de automação e domésticos, como câmeras

de vigilância, dispositivos sensores, atuadores ou monitores. Isso permitirá o desenvolvimento de aplicações que podem explorar uma quantidade enorme de dados gerados por esses dispositivos [1], [2]. Estas aplicações podem surgir de muitos domínios, como automação industrial e residencial, gestão inteligente de ambientes, saúde e cidades inteligentes ([3], [1], [4], [5], [6]).

Um dos desafios para o desenvolvimento da IoT é o número crescente e a heterogeneidade de dispositivos. Em 2010, o número de dispositivos conectados à Internet já estava na ordem de 10^9 a e espera-se que ele alcance 10^{11} até 2020 [7].

A diversidade de tecnologias usadas na IoT requer o suporte de serviços de *middleware* e estruturas de desenvolvimento com abstrações bem definidas. O *middleware* pode abstrair a complexidade tecnológica e um *framework* pode agilizar o desenvolvimento das aplicações.

Por exemplo, usar uma abordagem orientada a serviços, manipulando dispositivos como serviços de software, pode facilitar a integração e fazer uso de padrões de interação conhecidos, como *register-lookup-use* [8]. No entanto, as técnicas usadas para encapsular dispositivos em artefatos de software precisam lidar com elementos típicos de dispositivos, como sensores/atuadores e atributos específicos, como localização, estado, frequência ou intervalo de medição [8]. Além disso, mais importante, alguns dispositivos podem requerer o uso de estilos de interação diferentes, *request-response* ou *publish-subscribe*, o que pode adicionar complexidade [9], [10] [11], [12].

Iniciativas e soluções como FIWARE [13], SOFIA [14], OpenIoT [15], Particle [16], KNOT [17] e Eclipse IoT [18] de um modo geral oferecem plataformas e tecnologias que permitem o desenvolvimento de aplicações para a IoT em alto nível de abstração. *Frameworks* propostos para IoT geralmente contemplam serviços de suporte, formas de integração e interação destes serviços, e de dispositivos, com módulos-cliente das aplicações [8], [9], [19]. Também são geralmente oferecidas opções de persistência de dados entre aplicações, serviços e dispositivos, em repositórios de dados locais ou na nuvem [2]. Com isso podem ser utilizadas técnicas inteligentes atuando sobre grandes massas de dados coletados.

Órgãos de padronização como ITU-T, IEEE e consórcios como o oneM2M e OASIS trabalham na padronização de uma arquitetura para a Internet das Coisas, em suas camadas de infraestrutura, segurança e privacidade, rede, comunicação máquia-máquina (*machine-to-machine*) e serviços para as aplicações [20], [21], [22], [23], [24], [25], [26]. Um dos objetivos é orientar a concepção de produtos e sistemas, para que sejam interoperáveis.

Aplicações desenvolvidas para utilizar sistemas de suporte para IoT podem integrar e fazer uso de diversos dispositivos, de fabricantes diferentes com recursos heterogêneos e limitações específicas. Estes dispositivos podem oferecer serviços através de interfaces de software específicas e se conectar por diferentes interfaces físicas, como USB, Bluetooth, ZigBee ou Ethernet. Mesmo em um nível mais alto de abstração, cada dispositivo pode utilizar protocolos de comunicação específicos e formatos de dados diferentes [27].

A Figura 3.1 apresenta uma organização geral de elementos para a Internet das Coisas. Uma aplicação pode utilizar serviços de *middleware*, quando conveniente, para

acessar serviços e dispositivos em nível alto de abstração. A comunicação entre os serviços de *middleware* com os dispositivos e serviços pode ser feita diretamente ou através de um *gateway*[1]. Neste contexto, as arquiteturas propostas para IoT incluem Protocolos de Aplicação, orientados à interação entre processos e dispositivos ou entre dispositivos. Estes protocolos podem ser utilizados tanto pelos serviços de *middleware* — que encapsulam o código necessário — como diretamente pelas aplicações — que podem fazer um uso específico dos mesmos.

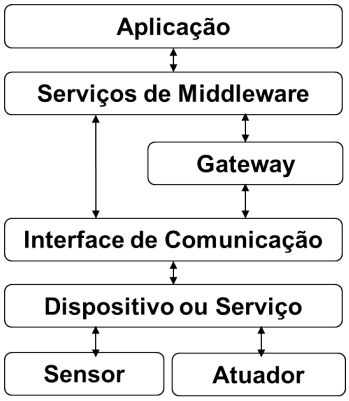


Figura 3.1. Organização de elementos na IoT.

Vários Protocolos de Aplicação estão estabelecidos e alguns padronizados, com destaque para o MQTT [29], CoAP [30], AMQP [31] e XMPP [32]. Cada um tem características indicadas para determinadas aplicações em IoT, dependendo dos recursos e modos de operação dos dispositivos ou serviços a serem integrados (Figura 3.2).

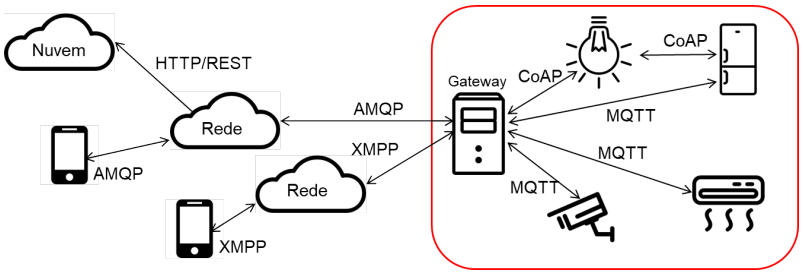


Figura 3.2. Aplicação com uso de múltiplos protocolos de aplicação.

Alguns destes protocolos contemplam aspectos não-funcionais como segurança, autenticação, QoS e escalabilidade (por exemplo, [10], [33], [11]), que podem não ser integralmente aproveitados quando um *framework* é empregado. Além disso, fabricantes podem fornecer dispositivos com algum Protocolo de Aplicação embarcado ou, ainda, um novo dispositivo pode requerer justamente o provisionamento de um Protocolo de Aplicação para realizar a comunicação com o ambiente de IoT não contemplado em um *framework*. Assim, mesmo utilizando (ou até desenvolvendo) um *framework* para IoT é importante dominar os conceitos e mecanismos destes protocolos.

[1] *Gateways* para IoT poderiam ser discutidos num capítulo à parte. Soluções em hardware e software podem ser consultadas em [28].

O objetivo deste capítulo é apresentar os principais Protocolos de Aplicação para IoT atualmente disponíveis — MQTT, CoAP, AMQP e XMPP — mencionados anteriormente, através de uma abordagem prática. Ainda, a tecnologia WebSocket [34] e a abordagem REST sobre HTTP também serão apresentadas como alternativa de suporte para a interação entre aplicações, dispositivos e serviços. Serão discutidos os conceitos principais, as diferenças entre os protocolos — relacionando estas diferenças com as aplicações e cenários para os quais são mais indicados — e apresentados exemplos práticos de programação e seus ambientes de execução. Com esta base será apresentado um estudo de caso mais estruturado com o emprego de um dispositivo NodeMCU.

As seções deste capítulo estão estruturadas da seguinte forma. Na Seção 3.2 são resumidos os estilos de interação *request-response* e *publish-subscribe*, base para os protocolos discutidos. Antes de apresentar os protocolos, a Seção 3.3 apresenta o mecanismo de WebSockets, que viabiliza o uso de vários protocolos de aplicação a partir de um navegador Web. Em seguida, a Seção 3.4 apresenta cada um dos protocolos, destacando suas principais características. Para ilustrar de forma prática o uso de cada protocolo, a Seção 3.5 apresenta um exemplo simples, junto com as bibliotecas e ambientes utilizados para o desenvolvimento e operação do mesmo. A Seção 3.6 traz um estudo de caso mais estruturado, onde se explora a conciliação de dois protocolos. Por último, a Seção 3.7 conclui o capítulo. No Apêndice 3.8 listamos referências complementares para ampliar as opções do leitor nos seus primeiros passos para o uso dos protocolos apresentados.

3.2. Estilos de interação

Sistemas de acionamento e monitoramento normalmente utilizam dois estilos de interação: requisição-resposta (*request-response*) e publica-subscreve (*publish-subscribe*) [35]. Aplicações na Internet das Coisas têm este perfil: ou a aplicação tem interesse em monitorar algum sensor — ou conjunto de sensores, ou interesse em acionar algum dispositivo atuador, ou os dois. Estas aplicações geralmente incorporam uma estrutura de laço com os seguintes passos: monitoramento, avaliação, acionamento, reconfiguração e novo monitoramento. Neste laço, ou de forma concorrente, podem estar incluídas nas atividades da aplicação a interação com o usuário, exibição de informações, etc.

Considerando os passos de monitoramento e acionamento, de uma forma geral, aplicações de IoT podem precisar interagir de forma síncrona com um processo, agente ou dispositivo remoto. Isto é, a aplicação envia uma mensagem com uma requisição e só pode continuar suas atividades depois de receber uma resposta, ou uma recusa.

Outra possibilidade é a aplicação IoT não poder, ou não querer, aguardar uma resposta, mas receber uma notificação de forma assíncrona. Isto é, a aplicação registra sua requisição junto ao seu alvo, e continua sua execução logo em seguida. Num momento posterior, uma notificação pode chegar, e a aplicação deve estar preparada para tratar a mesma. Este estilo de interação é útil quando um evento é gerado, um dado estado é atingido ou uma computação concluída por uma entidade remota e, tão logo ocorra, isso deva ser informado à aplicação.

3.2.1. Direta - Requisição-Resposta (*Request-Response*)

Protocolos de Aplicação que utilizam o estilo *request-response* são estruturados para oferecer interação síncrona e direta entre dois processos (Figura 3.3).

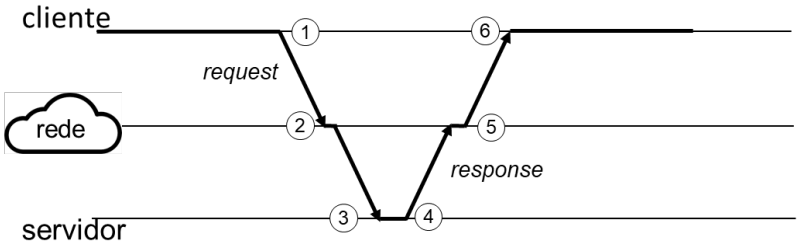


Figura 3.3. Estilo de interação *request-response*.

1. O processo cliente, ou enviante, envia uma mensagem contendo uma requisição para o processo servidor, ou recebedor e em seguida é bloqueado para aguardar a resposta;

2. A mensagem contendo a requisição precisar ser transportada pela rede, passando pelas barreiras de segurança, até que seja entregue ao servidor;

3. O processo servidor, que estava bloqueado, aguardando uma requisição, recebe a mensagem, interpreta o seu conteúdo e realiza o serviço solicitado na requisição;

4. A requisição pode, por exemplo, solicitar algum cálculo/processamento do servidor, a leitura de uma variável de um sensor ou o acionamento de um atuador. Se uma resposta é necessária, uma nova mensagem é preparada contendo esta resposta e retornada para o cliente;

5. A mensagem de resposta também precisa ser transportada pela rede, passando pelas barreiras de segurança no caminho de volta;

6. Finalmente, é recebida pelo processo cliente, requisitante, que se desbloqueia, avalia resposta e prossegue sua execução.

É importante observar que as duas trocas de mensagens ocorrem de forma direta: os dois elementos cliente e servidor precisam estar envolvidos diretamente. A requisição só é recebida pelo servidor depois que o cliente envia a mensagem, e esta é recebida com algum atraso devido à latência da rede e dos sistemas de suporte à troca de mensagens. Se o servidor se prepara para receber a requisição antes da mensagem ser transmitida, este é bloqueado até que a mesma esteja disponível.

Por outro lado, se uma resposta é necessária, os papéis são invertidos. Logo após o envio da requisição o cliente se coloca pronto para receber uma mensagem contendo a resposta e, por isso, é bloqueado. O servidor prepara uma mensagem com a resposta e a envia para o cliente. Este é desbloqueado apenas quando a mensagem é efetivamente entregue.

Uma característica inerente à comunicação direta, o processo cliente precisa conhecer a referência ou endereço do processo servidor, geralmente o número IP, a porta e o protocolo de transporte utilizado (no caso do presente capítulo, UDP ou TCP). O servidor, por sua vez, deve expor uma referência ou endereço conhecido e possível de ser obtido pelo processo cliente. Além disso, elementos de segurança, como *firewall* e NAT, precisam ser configurados para o endereço do servidor.

3.2.2. Indireta - Publica-Subscreve (*Publish-Subscribe*)

O estilo de interação indireta, publica-subscreve (*publish-subscribe*), também chamado de interação baseada em eventos (*event-based*), é apropriado para comunicação assíncrona, quando os elementos que precisam interagir, não podem ou não querem estabelecer uma comunicação direta para trocar mensagens, ou não podem aguardar bloqueados a interação ocorrer. Ainda, também é a opção adequada quando a informação a ser comunicada pode ser produzida à qualquer momento, como por exemplo, uma variável medida por um sensor atinge um valor ou limiar.

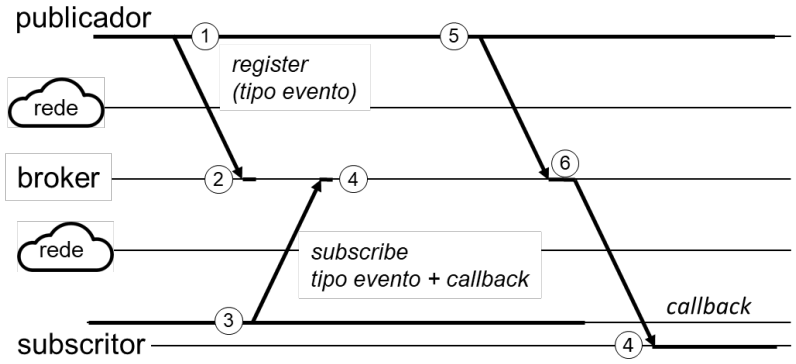


Figura 3.4. Estilo de interação *Publish-Subscribe*.

O suporte para este estilo de interação é mais complexo, se comparado com o estilo *request-response*. Seguindo a Figura 3.4, observamos os seguintes elementos:

- Publicador (*publisher*) ou fonte de evento (*event source*), é o elemento que pode gerar eventos. A aplicação pode interagir com vários publicadores.
- Evento, é uma abstração que pode ser representada por uma estrutura de dados ou objeto. Os eventos podem ter um tipo, identificação ou atributos, que indicam a sua natureza ou origem.
- Subscritor (*subscriber*) ou sorvedor de evento (*event sinc*), é o elemento que está interessado em receber notificações quando eventos de determinado tipo ocorrem. A aplicação pode trabalhar com vários subscritores concorrentemente.
- O subscritor interessado precisa informar uma referência de *callback* ou tratador (*handler*), para onde a notificação deve ser enviada e o evento entregue. Essa referência pode ser padronizada, pode ser um endereço, ou um nome de procedimento/método.

- Sistema de eventos (*event engine*) ou *event broker* ou simplesmente *broker* (termo mais comum nos sistemas de IoT), intermedeia as interações entre publicadores e subscritores, incluindo os pedidos de registro e interesse em eventos, e o encaminhamento das notificações.

Como existem mais elementos e mais trocas de mensagens entre os elementos, os atarsos de comunicação poderiam interferir mais, se comparado ao *request-response*. Por outro lado, observe-se também na Figura 3.4 que as linhas de interação entre os elementos ocorrem de forma independente ao longo do tempo, reforçando a ideia de uma interação assíncrona. Assim, os atrasos da rede podem ter importância menor.

1. O publicador primeiro envia uma mensagem para o *broker* pedindo o registro de um tipo de evento;
2. O *broker* recebe a mensagem e registra o tipo de evento.

   Em alguns casos, esta sequência pode ser feita por configuração administrativa.
3. O processo subscritor envia uma mensagem para o *broker* registrando interesse em um tipo de evento, e passa a referência de *callback*;
4. O *broker* verifica os aspectos de segurança – quando isso fizer parte de sua operação – e, então, adiciona a referência informada pelo subscritor na lista de interessados pelo evento;

   Depois de registrar o interesse em um evento, o processo subscritor pode continuar seu processamento.
5. Em algum momento qualquer, o publicador gera um evento e envia uma mensagem ao *broker* solicitando sua publicação;
6. Recebendo a publicação, o *broker* percorre a lista relacionada ao tipo de evento e envia uma mensagem de notificação para cada subscritor interessado, em sua referência da *callback*;
7. A notificação é recebida na referência de *callback* pela rotina de tratamento, *handler*, sem que a rotina original seja interrompida. A arquitetura de software necessária para o tratamento concorrente das atividades “normais” e de notificações não é aprofundada aqui.

A necessidade de um elemento intermediário, como o *broker*, introduz um ponto de complexidade: temos mais um elemento, que precisa executar como um serviço contínuo, que por sua vez precisa ser mantido/administrado. Entretanto, como ressaltado nas próximas seções, este elemento pode ser responsável por procedimentos de segurança e controle de acesso, o que é positivo. Pode também oferecer opções de enfileiramento, filas com prioridade diferenciada e persistência de eventos, permitindo que um subscritor que estava *offline* receba notificações passadas. Um elemento central como o *broker* também pode facilitar a comunicação de elementos atraz de redes com NAT e *firewall*, sem a necessidade de esforços de configuração nos elementos de borda da rede.

Por último, uma dica que pode ajudar o entendimento do uso dos protocolos. Os mecanismos relacionados ao estilo de interação *publish-subcribe* podem ser usados em uma aplicação de uma maneira um pouco "invertida" no caso de atuadores. Por exemplo, digamos que a aplicação precise acionar um relé controlado por um dispositivo Arduino. Digamos também, que uma aplicação Android permita ao usuário controlar este relé remotamente, enviando um "comando" de liga-desliga. O Arduino controlando o relé poderia ter o papel de subscritor, registrando o interesse em "eventos de liga-desliga". A aplicação Android, por sua vez poderia fazer o papel de fonte de evento do tipo "liga-desliga", enviando um evento quando o usuário assim acionasse, que por sua vez levaria ao *broker* notificar o dispositivo Arduino, que então atuaria ligando ou desligado o relé.

3.3. WebSockets

O WebSocket [34] é um mecanismo similar ao *socket* "tradicional", proposto para oferecer suporte para troca de mensagens bidirecionais, *full-duplex*, sobre TCP, entre um navegador Web e um servidor HTTP remoto, que suportem HTML5 (Figura 3.5). Navegadores como, por exemplo, o Internet Explorer, Mozilla Firefox, Google Chrome, Safari e Opera já possuem ou estão viabilizando suporte ao WebSocket.

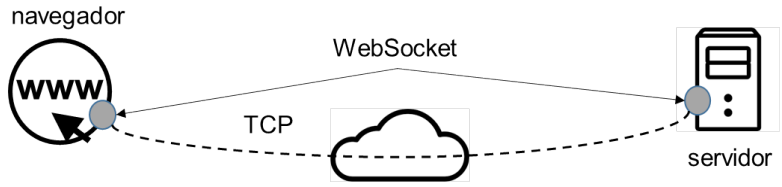


Figura 3.5. Comunicação com WebSocket.

O mecanismo WebSocket cria uma conexão que pode ser mantida aberta por um longo tempo, com pouco impacto no desempenho do navegador. É possível abrir e fechar conexões a qualquer momento. Com isso passa a ser também possível receber notificações, sem a necessidade de uso de "truques" como o *polling*, com vantagens na estrutura dos programas, melhor desempenho e latências menores.

O WebSocket tem características específicas em relação ao "socket" tradicional, dado que o contexto de uso é, em princípio, específico ao ambiente Web. Existe a preocupação de compatibilidade com o HTTP. Por isso, existe um protocolo WebSocket, padronizado pelo IETF [36] na RFC 6455, que gerencia o ciclo de vida de uma conexão WebSocket e permite, também, que uma conexão iniciada por HTTP chaveie para uma conexão WebSocket. Isso é feito através de uma mensagem HTTP *UPGRADE*.

Para o desenvolvimento das aplicações é proposta uma API de programação, padronizada pelo W3C [37]. A API para WebSockets abre a possibilidade para que aplicações para IoT possam também utilizar os navegadores como parte da solução para acionar diretamente um atuador ou obter uma informação de algum sensor. Este é o ponto explorado aqui.

O programador deve utilizar uma linguagem de programação Web, como JavaScript, e bibliotecas de suporte ao mecanismo WebSocket para desenvolver uma aplicação

cliente executando em um navegador. Após abrir uma conexão TCP utilizando WebSocket, o programa pode enviar ou receber mensagens contendo qualquer tipo de informação. É possível criar um protocolo customizado para uma aplicação específica, mas também é possível transportar protocolos padronizados, como os Protocolos de Aplicação discutidos neste capítulo como o MQTT, AMQP e XMPP[2].

A API WebSocket requer que o programador descreva a referência, ou endereço, do servidor remoto através de uma URL, e informe um esquema específico *ws* ou *wss* (para a versão segura), por exemplo: `ws://lcc.uerj.br:8000/`. A versão segura utiliza a camada SSL/TLS como no HTTPS.

O fluxo de interação entre o navegador e o servidor Web segue àquele utilizado por *sockets* TCP (Figura 3.6).

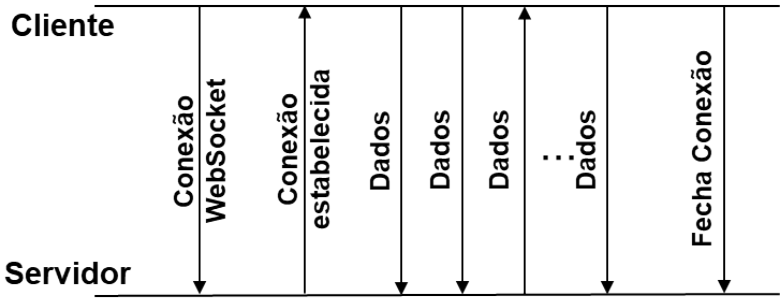


Figura 3.6. Conexão, troca de mensagens, desconexão com WebSocket.

A API do mecanismo WebSockets oferece primitivas para a criação, conexão, envio-recebimento de mensagens e término da conexão. Também oferece alternativas para recebimento de eventos e integração com o HTTP e mecanismos do navegador. Vamos utilizar um exemplo de código para introduzir a API e discutir o fluxo de interação (Código 3.1 e Código 3.2). Tanto o cliente quando o servidor utilizam o Node.js [38], e fazem uso dds mecanismos de notificação disponíveis neste tipo de ambiente.

```
1 const WebSocket = require('ws');
2 const cliWS = new WebSocket('ws://lcc.uerj.br:8080/');
3 cliWS.addEventListener('open', function sndRequest() {
4   var obj = new Object();
5   obj.a = 15;
6   obj.b = 16
7   cliWS.send(JSON.stringify(obj));
8 });
9 cliWS.addEventListener('message', function recResponse(data) {
10  console.log(data);
11  cliWS.close();
12 });
```

Código 3.1. Exemplo de programa cliente Websocket.

[2]O CoAP é baseado em UDP e, em princípio não pode ser transportado por TCP (Seção 3.4.2)

Inicialmente o cliente carrega a biblioteca de suporte (linha 1) e, em seguida, cria o Websocket, passando como parâmetro a URL do servidor (linha 2). O cliente precisa aguardar a conexão ser estabelecida e para isso registra a função *sndRequest()*, que é executada quando a conexão estiver pronta (evento *open* do *cliWS*), linha 3. Quando isso ocorre, um objeto é criado e enviado em uma estrutura *JSON*, linha 7. Por último, a função *recResponse(data)* é registrada para ser executada quando uma mensagem de retorno chegar (linha 9). Quando esta estiver disponível, a informação de resposta é logada e a conexão fechada (linha 11).

A execução do cliente em um navegador pode ser feita diretamente pelo ambiente de Node.js ou simplesmente carregando uma página HTML contendo uma *tag* para executar o código a partir de um arquivo.

```
1 const WebSocket = require('ws');
2 const wsServer = new WebSocket.Server({ port: 8080 });
3 wsSocket.on('connection', function execServ(activeWS) {
4   activeWS.on('message', function recvRequest(msg) {
5     console.log('received: %s', msg);
6     obj = JSON.parse(msg);
7     var resp = new Object(); resp.value = obj.a+obj.b;
8     activeWS.send(JSON.stringify(resp));
9   });
10 });
```

Código 3.2. Exemplo de programa servidor WebSocket.

O programa do servidor segue a mesma linha (Código 3.2). Inicialmente, é instanciado um WebSocket "servidor", passivo, especializado em receber pedidos de conexão, associado à porta 8080 (linha 2). Isso resolve em parte o problema do cliente conhecer a referência do servidor, pois a porta é fixa. Em seguida a função *execServ(activeWS)* é registrada para execução quando um pedido de conexão for aceito (linha 3). Como resultado de uma nova conexão, um Websocket ativo é criado (*activeWS*), semelhante ao que ocorre com o *socket* TCP tradicional. Cada conexão estabelecida recebe uma instância diferente da função *execServ*, responsável por tratar a conexão com o cliente de forma concorrente (detalhes de como isso acontece fogem do escopo do capítulo). A execução do serviço no exemplo é simples: a função *recvRequest* é registrada para ser executada quando a mensagem *msg* for recebida (linha 4). Esta função loga a mensagem, obtém as informações transmitidas (esta seria a requisição), executa a soma (esta seria a resposta) e envia a mesma como uma estrutura *JSON* pelo WebSocket ativo (linha 8). Observa-se que no servidor não fechamos a conexão, pois ele está preparado para receber novas conexões.

O código apresentado para o servidor precisa de algumas adaptações para executar no contexto de um servidor Web, como o Apache, ou de um Servidor de Aplicação, como o GlassFish. Mas o uso da API de WebSocket é essencialmente o mesmo.

Na próxima seção os Protocolos de Aplicação serão apresentados. O WebSocket será utilizado depois, na seção de exemplos, e também como mecanismo de suporte para o uso destes protocolos em navegadores Web.

3.4. Protocolos

Nesta seção apresentamos os Protocolos de Aplicação para IoT. Para cada protocolo serão discutidos os conceitos básicos, mecanismos/estruturas principais e os estilos de interação suportados. Também serão abordados aspectos específicos, que distinguam os protocolos como, por exemplo aspectos de QoS, diferentes esquemas de filas, facilidade para passar por NAT e *firewall*, registro, autenticação e mecanismos de segurança disponíveis. Também importante, serão apresentadas as soluções para identificação de referências, endereços, eventos e filas, quando pertinente.

3.4.1. HTTP/REST

O *Hypertext Transfer Protocol*, HTTP [39], é um protocolo com estilo *request-response*, descrito na RFC 2616, base para a comunicação de dados para a *World Wide Web*, WWW, desde 1990. Requisições do navegador Web e a resposta do servidor são transportadas por HTTP. Como outros protocolos de aplicação da Internet, o HTTP tem as informações de controle transmitidas como cadeias de caractere.

O HTTP é **media independent**, ou seja, qualquer tipo de dado pode ser enviado, desde que o cliente e o servidor saibam como lidar com os conteúdos. A informação efetivamente sendo transportada pode ter um dos tipos previstos no padrão, usando o tipo *MIME*, descrito em um campo chamado *content-type*, como por exemplo, imagens, arquivos compactados, textos em HTML ou XML.

O TCP é utilizado como protocolo de transporte. Nas primeiras versões do HTTP a conexão era desfeita logo após uma resposta ser retornada para o cliente. Na versão 1.1, descrita na RFC 2616 [39], o protocolo passa a manter a conexão aberta entre cliente e servidor, permitindo que sequências de requisições e respostas sejam tratadas, melhorando o desempenho.

Também é possível prover segurança na comunicação entre um cliente e servidor HTTP utilizando a versão segura, HTTP *Secure*, que executa sobre uma camada SSL/TLS.

O HTTP faz uso do Identificador Uniforme de Recursos, *Uniform Resource Identifier*, URI [40], como referência para o servidor com o qual a conexão será estabelecida, e para determinado recurso hospedado neste servidor. No seguinte exemplo, (i) *http* é chamado de esquema, e indica o protocolo ou tecnologia utilizada; (ii) *www.lcc.uerj.br* é chamado de domínio, e será convertido para um número IP após consulta ao DNS; (iii) *8080* é a especificação da porta onde o servidor espera receber pedidos de conexão. Caso não seja informado, o esquema vincula a porta a um padrão (80, no caso do HTTP); (iv) *laboratorio* indica a rota ou caminho para o diretório onde o recurso está disponível; e (v) *recurso* é a referência ao recurso desejado.

```
http://www.lcc.uerj.br:8080/laboratorio/recurso
```

Uma vez estabelecida a conexão, mensagens HTTP de requisição e reposta são transmitidas. São padronizados alguns tipos de mensagem, chamados de métodos no HTTP, que podem provocar reações diferentes do servidor. Nos exemplos deste capítulo utilizaremos:

GET. Recupera um *recurso* do *servidor*, dados fornecidos na URI.

POST. Envia dados para o servidor no corpo da mensagem HTTP, como por exemplo, informações de texto, *upload* de arquivos, ou informações contidas em formulários HTML. O servidor precisa estar configurado e preparado para realizar alguma ação com as informações enviadas. O exemplo "clássico" são os programas acionados pelo mecanismo de CGI, disparados quando o servidor identifica o parâmetro *action* dentro de um formulário HTML.

Outros métodos disponíveis são: *HEAD*, *PUT*, *DELETE*, *CONNECT*, *OPTIONS* e *TRACE*.

Por padrão, as mensagens de resposta contém um código de retorno, entre os quais: (i) 200, indica sucesso; (ii) 404, indica recurso não encontrado e (iii) 500, erro interno do servidor.

O Código 3.3 apresenta trechos das mensagens de requisição e resposta utilizando o método *GET*. Observa-se que as informações de controle aparecem em texto pleno.

```
1 GET / HTTP/1.1
2 Host: www.lcc.uerj.br
3 Connection: keep-alive
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) [...]
5 Accept: text/html,application/xhtml+xml,application/xml; [...]
6 Accept-Encoding: gzip, deflate
7 Accept-Language: pt-BR,pt;q=0.9,en-US;q=0.8,en;q=0.7

8 HTTP/1.1 200 OK
9 Date: Wed, 11 Apr 2018 23:22:40 GMT
10 Server: Apache/2.2.15 (CentOS)
11 Connection: close
12 Content-Type: text/html
```

Código 3.3. Mensagens de requisição e resposta HTTP com método *GET*.

A razão central para considerar o HTTP como suporte à interação entre processos, dispositivos e serviços em uma aplicação para a IoT é utilizar um protocolo de aplicação provadamente funcional, disseminado, que executa sobre uma infraestrutura de comunicação já estabelecida, com segurança geralmente configurada.

Os métodos disponíveis *PUT*, *GET*, *POST* e *DELETE*, e a abordagem de campos e meta-informações do HTTP podem ser relacionados a uma interface CRUD (*Create* (Criar), *Retrieve* (Consultar), *Update* (Atualizar) e *Delete* (Apagar)), amplamente usada em sistemas de informação. Estes poderiam ser utilizados para acionar atuadores ou obter informações de sensores, por exemplo. O problema, em princípio, seria: (i) mudar a interpretação padrão dos servidores Web para estas operações e (ii) prover um mecanismo para executar ações fora do contexto do processo do servidor Web. Por exemplo, o método GET, por padrão, implica em se percorrer o *caminho* até o diretório onde o *recurso* se encontra e em seguida recuperar o arquivo cujo nome é "recurso" e enviar o mesmo como resposta para o cliente. Outro aspecto desta interpretação é que o conteúdo de *recurso* é estático, à menos que seja alterado manualmente. No caso de IoT queremos chamar um

procedimento para acionar um atuador, ler um sensor ou solicitar um serviço, por exemplo. Neste caso, a execução e a resposta podem ser diferentes a cada chamada.

Ao longo dos anos de operação da Web, alguns mecanismos foram propostos para utilizar a sua infraestrutura para executar procedimentos remotos. O *Common Gateway Interface*, CGI, foi um dos primeiros mecanismos propostos [41]. A cada solicitação, um ambiente de execução é criado e um programa carregado e executado. Parâmetros de entrada podem ser passados como variáveis de ambiente para o programa a ser executado. A resposta é, geralmente, um texto em HTML criado dinamicamente pelo programa. Várias evoluções foram propostas.

Os Serviços Web (*Web Services*) são uma outra abordagem, mais eficiente que o CGI, que permite que um processo cliente envie uma requisição para execução remota de procedimento. Para isso, é enviando para o servidor o nome do procedimento e uma lista de parâmetros de entrada. O serviço é realizado e uma resposta é retornada para o cliente. Conceitualmente, funciona como o mecanismo *Remote Procedure Call*, RPC, utilizando também o conceito de interface, descritas em WSDL, e é aderente ao padrão SOA (*Service Oriented Architecture*). Requisições e respostas são representadas em XML. Um esquema XML, chamado SOAP, regulamenta como os elementos das requisições e respostas devem ser estruturados em arranjos e *tags* XML específicos. *Web Services* são amplamente suportados por bibliotecas, servidores HTTP e servidores de aplicação especializados. A abordagem é bem completa e tem um padrão estabelecido [42].

Sistemas e aplicações para IoT podem ser baseadas em *Web Services*. Entretanto, a abordagem tem custo computacional notável. Este custo pode ser atribuído às rotinas de *parsing* das estruturas XML, pela necessidade de ser manter os elementos de suporte para descrição de interfaces em WSDL e pelo custo dos mecanismos que encaminham as informações de uma chamada para a entidade que vai executar a mesma.

REST sobre HTTP. A abordagem *REpresentational State Transfer*, REST, tem se mostrado um alternativa interessante para IoT [43], [44]. Independente de aspectos conceituais discutidos em [45], REST permite o uso da infraestrutura do HTTP para acionar ou obter recursos, apenas dando uma interpretação diferente para os métodos *GET*, *PUT*, *POST* e *DELETE*, e valendo-se da possibilidade de enviar informações adicionais numa mensagem HTTP (que podem ser interpretados como parâmetros de entrada). O conceito em torno de REST discute que a abordagem não é um RPC, que o melhor uso não é fazer chamadas remotas de procedimentos com uma possível resposta como retorno. Entretanto, algumas infraestruturas para IoT utilizam REST para ler e acionar dispositivos transferindo estruturas JSON nos dois sentidos, como por exemplo [13], com o objetivo de se ter um mecanismo de interação distribuído, com custo aceitável executando sobre HTTP.

Todo mecanismo que estende a interpretação padrão dos métodos do HTTP precisa configurar o servidor Web com este objetivo. Isso vale para servidores bem estruturados como o Apache [46] ou servidores customizados, como veremos na seção de exemplos. Sem entrar em detalhes, e apenas considerando o REST, esta configuração pode ser feita confinando as aplicações ou serviços REST em um diretório específico, que por sua vez terá configurado o comportamento desejado[3].

[3]Uma tática simples é utilizar um arquivo como o *.httaccess* com as configurações e transformações

Neste capítulo considera-se, então, a estratégia de combinar o protocolo HTTP e a abordagem REST como opção de Protocolo de Aplicação para integrar processos, dispositivos e serviços em aplicações para IoT. Os serviços ou dispositivos oferecidos como recursos para aplicações IoT precisam aderir ao padrão arquitetural do REST, passando a ser "considerado" RESTful. Como qualquer outro recurso referenciado e recuperado por HTTP, um serviço RESTful, precisa estar localizado num rota alcançável pelo servidor Web no sistema de arquivos local. O servidor Web utiliza, então, as informações enviadas pelo cliente na mensagem HTTP: (i) a *rota*, para localizar o recurso e (ii) o *recurso*, utilizado como nome do serviço RESTful a ser acionado. Por exemplo, o trecho código HTTP seguinte, extraído do exemplo na Seção 3.5.2 mostra o método *PUT* indicando o acionamento do recurso *vermelho*, localizado em */servicos/LEDServer/* no host *teste.lcc.uerj.br*. Como resposta uma mensagem HTTP com código *200 OK* é devolvida.

```
PUT /servicos/LEDServer/vermelho HTTP/1.1
Host: teste.lcc.uerj.br
...
HTTP/1.1 200 OK
```

A URI utilizada seria, por exemplo:

```
http://teste.lcc.uerj.br:8080/servicos/LEDServer/vermelho
```

O mecanismo REST não precisa considerar detalhes da implementação do serviço RESTful. Por outro lado, o mecanismo pode usar as informações de controle do HTTP para complementar a definição de como o serviço será executado. Campos customizados podem ser usados para transferir parâmetros para o serviço. Mesmo não sendo recomendado, o próprio campo com a informação útil sendo transportada pode transferir dados em notação XML ou JSON. A configuração definida no servidor para tratar a requisição REST e a própria implementação do serviço é que vão definir como estas informações são utilizadas e como o serviço será acionado [47].

Em resumo, para desenvolver aplicações HTTP/REST o programador precisa:

- desenvolver o serviço e implantá-lo no servidor, na localização adequada;
- configurar o servidor para acionar corretamente o serviço;
- desenvolver a aplicação cliente montando a URI e informações adicionais a serem inseridas na mensagem HTTP. Bibliotecas e IDEs amplamente disponíveis facilitam a programação (Seção 3.5.2).

Como última observação, o uso de HTTP não é obrigatório para acionamento de serviços com abordagem REST. Um exemplo é a adoção desta abordagem no protocolo CoAP descrito na próxima seção. Na Seção 3.5.2 apresentamos uma aplicação HTTP/REST, utilizando alguns dos aspectos discutidos aqui.

necessárias, no caso de um servidor Apache.

3.4.2. CoAP

O *Constrained Application Protocol*, CoAP [30], é projetado para permitir a troca direta de mensagens entre dispositivos com poucos recursos computacionais, interconectados por redes com baixas taxas de transferência, com a 6LowPAN [48], por exemplo. Os dispositivos considerados geralmente são construídos com microcontroladores de baixo custo, como por exemplo o Arduino, NodeMCU e outros. O CoAP é descrito na RFC 7252 do IETF CoRE (*Constrained RESTful Enviroments*) Working Group.

O estilo de interação oferecido pelo CoAP é o *request-response*, com características semelhantes ao HTTP/REST. Como na *Web*, os dispositivos são referenciados usando o endereço IP e o número da porta. Entretanto, por estratégia de projeto, o CoAP é executado sobre UDP e não TCP. Isso pode dificultar a implantação de uma aplicação cujos elementos estejam distribuídos em redes atrás de NAT e firewall. A configuração, neste caso, deve considerar tanto os clientes como os servidores dado que não existe uma conexão. Assim, o CoAP é mais facilmente usado em uma rede local.

O CoAP não foi desenhado para substituir o HTTP, mas ele implementa um pequeno subconjunto de práticas HTTP amplamente aceitas, e as otimiza para a troca de mensagens entre dispositivos (*machine-to-machine*, M2M) [23].

O acesso aos serviços expostos pelo dispositivo se dá por URIs REST. Também de forma semelhante ao HTTP, são previstos tipos de requisição, por exemplo *GET*, *PUT*, *POST* e *DELETE*, com códigos de resposta, como por exemplo, 404, 500 e a especificação de um tipo de conteúdo usados para transmitir informações.

As mensagens transportadas pelo CoAP tem formato binário, permitindo o transporte de qualquer formato de dados, como por exemplo XML, JSON ou qualquer outro desejado. Possui um cabeçalho fixo de 4 bytes e um *payload* máximo que deve caber em um único datagrama UDP, ou no *frame* IEEE 802.15.4 [48].

O CoAP pode também usar o DTLS como protocolo de transporte para comunicação segura. Além disso, para contornar a falta de garantia de entrega de pacotes do UDP, o CoAP oferece dois níveis de qualidade de serviço: *Confirmado*, que adiciona um pacote *ACK* de confirmação de recebimento, e *Não-Confirmado*, sem mensagem de confirmação.

As opções na troca de mensagens dão flexibilidade ao CoAP. Por isso, discutimos algumas opções a seguir.

Confirmado com *piggbacking*. O cliente envia uma mensagem *CON* para o servidor solicitando a temperatura (Figura 3.7). O tipo da requisição é *GET*, a URL do caminho é "sensores/temperatura". O *ID* da mensagem é um número de 16 bits usado para identificar exclusivamente uma mensagem e ajudar o servidor na detecção de mensagens duplicadas. Assim que o servidor recebe a mensagem *CON*, a temperatura é obtida e uma mensagem de confirmação *ACK* é retornada, com o mesmo *ID*. Junto com a confirmação, a mensagem também leva “de carona” os dados de temperatura solicitada, no caso *30C*. Por fim, a resposta também tem um código de mensagem, neste caso, "2.05 Content". Estes são semelhantes aos códigos de status HTTP.

Se uma confirmação não for necessária, ou se for aceitável para a aplicação eventualmente não receber uma resposta, uma mensagem do tipo *NON* pode ser utilizada.

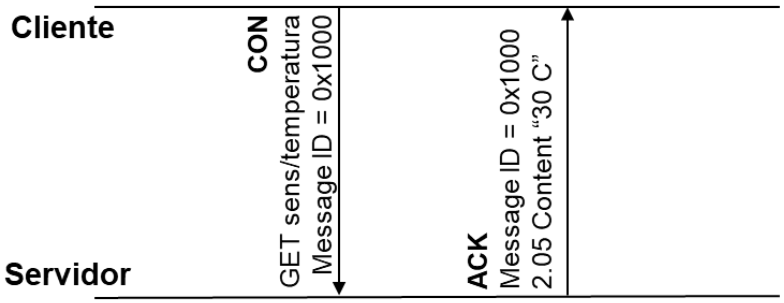


Figura 3.7. Mensagem CoAP confirmada com resposta *piggybacked*.

Confirmado sem *piggybacking* (correlação de mensagens). No caso em que a resposta para uma mensagem confirmada não possa ser levada de " carona", *piggybacked*, o ACK é enviado rapidamente, por exemplo, no caso do servidor precisar de um tempo maior para processar a mensagem. Posteriormente, uma resposta separada é enviada de volta ao cliente.

Como o CoAP é transportado por UDP, não existe uma conexão estabelecida: cada mensagem é um datagrama independente (por isso não pode ser usado com WebSockets). Assim, mensagens de requisição e resposta precisam ser correlacionadas pelo protocolo, dado que a interação deve ser coerente. A solução é o uso de um *token* para identificar o par de mensagens requisição-resposta (Figura 3.8).

Depois que o servidor tiver concluído o serviço, ele enviará uma outra mensagem *CON* contendo a resposta. Esta resposta possui um novo *ID* de mensagem e deverá ser confirmada posteriormente por um ACK correspondente. Entretanto, o valor do *token* desta resposta é o mesmo da requisição, permitindo ao cliente correlacionar as mensagens.

De uma forma geral, o *token* é usado para correlacionar mensagens, simulando uma conexão, mesmo que a mensagem seja não-confirmada e os IDs diferentes.

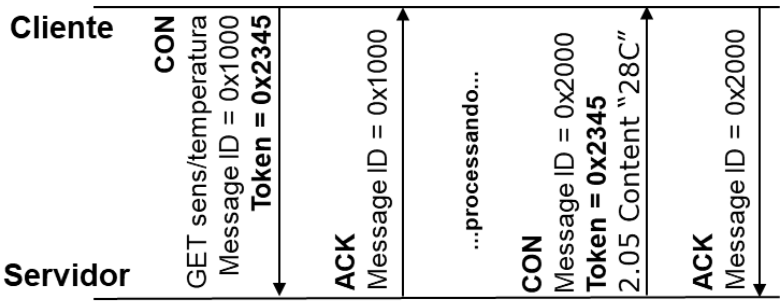


Figura 3.8. Mensagem CoAP confirmada, resposta separada (correlação).

Além da troca de mensagens *request-response*, o CoAP prevê duas outras funcionalidades e opções:

Observador. Com esta opção, a mensagem de requisição do cliente pode conter o registro do interesse por um recurso. Qualquer mudança de estado no recurso, por exemplo, pode ser notificada com o envio de uma mensagem. Neste caso, o *token* é usado para identificar o

evento ou origem da notificação. Com isso simula-se o estilo *publish-subscribe*, chamado de *resource-observe* no CoAP [49].

Descoberta de Recursos. O CoAP também padroniza uma forma para um cliente descobrir os recursos disponíveis em um servidor. Para isso, uma mensagem *GET* deve ser enviada com a URL do servidor finalizada com `/.well-known/core`. Neste caso, o servidor retorna os recursos disponíveis no formato CoRE Link [50].

O CoAP oferece outras opções de configuração e interação, além das discutidas nesta seção. Por exemplo, o conceito *block-wise* que facilita a transmissão de informações longas. A RFC 7252, já mencionada, apresenta estas outras opções.

3.4.3. MQTT

O *Message Queuing Telemetry Transport*, MQTT [29], foi concebido por Andy Stanford-Clark (IBM) e Arlen Nipper (Arcom, agora Cirrus Link) em 1999, como um protocolo leve para troca de mensagens entre dispositivos, que tivesse baixo consumo de bateria e pouco uso de banda de rede. Atualmente o MQTT é um padrão ISO/IEC, 20922:2016 e OASIS [51].

Embora o seu nome inclua "enfileiramento de mensagens" o MQTT não é orientado à fila de mensagens como o AMQP e o XMPP. Essencialmente o MQTT é um sistema gerenciador de eventos, que suporta o estilo de interação *publish-subscribe*, e utiliza o TCP para todas as trocas de mensagens entre publicadores e subscritores com o *broker*. A Figura 3.9 apresenta os elementos principais do MQTT.

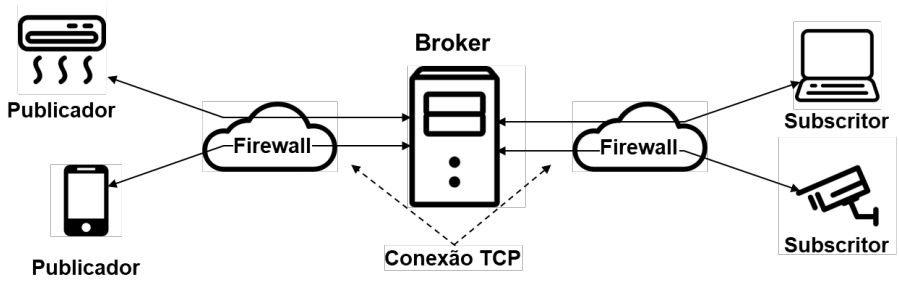


Figura 3.9. Elementos do MQTT.

A arquitetura centralizada do *broker*, facilita a introdução de mecanismos de autenticação e a implementação de características de confiabilidade ou garantia de entrega (chamadas de QoS no MQTT). As credenciais do usuário devem ser enviadas ao se estabelecer uma conexão. Além disso, a comunicação pode fazer uso da camada de segurança TLS/SSL.

No MQTT um tipo de evento é chamado de *tópico*. Um tópico é criado quando uma mensagem *PUBLISH* ou *SUBSCRIBE* é enviada (detalhes adiante). A estratégia para nomear ou referenciar um tópico segue a da formação de nomes de diretórios, que pode ter níveis separados pelo caractere "/" e usar *wildcards*.

Exemplo 1: `minhacasa/terreo/sala/temperatura`. O subscritor recebe o tópico *temperatura* de *sala*.

Exemplo 2: `minhacasa/terreo/$+$/temperatura`. O subscritor recebe o tópico *temperatura* de qualquer "parte" do *terreo*.

Exemplo 3: `minhacasa/terreo/\#`. O subscritor recebe qualquer tópico de qualquer "parte" do *terreo*.

Os clientes MQTT — publicadores e subscritores — e o *broker* comunicam-se através de mensagens de tipos específicos. As principais: *CONNECT*, *SUBSCRIBE*, *UNSUBSCRIBE* e *PUBLISH*, oferecem suporte para a interação no estilo *publish-subscribe*. Existem também mensagens do tipo *ACK* para cada ação, dependendo do nível de QoS selecionado. Para utilizar o MQTT o programador deve enviar e receber mensagens destes tipos, contendo alguns dados inerentes à informação de interesse, como, por exemplo, o tópico, e informações de controle, como, por exemplo, a senha do usuário. Na sequência discutimos as principais.

CONNECT. Enviada pelo cliente quando este deseja conectar-se ao *broker*. Como o MQTT utiliza o TCP, uma conexão é efetivamente estabelecida entre cada cliente e o *broker*. Isso facilita a administração da rede pois, a configuração de *firewall* e NAT precisa apenas ser realizada no nó onde o *broker* executa, uma vez que a conexão TCP permite troca de mensagens bi-direcional. Além disso, a conexão é também reconhecida como sendo o endereço de *callback* para todos os *tópicos* subscritos por cada cliente.

- **clientId**. Identificador único de cada cliente. A aplicação deve gerenciar esta informação.
- **cleanSession**. Especifica se uma conexão vai iniciar uma nova sessão ou vai reestabelecer uma sessão prévia. Se um dado cliente subscrever um tópico e indicar o uso de QoS 1 ou 2, isso fica registrado no *broker*. Caso seja necessário reestabelecer uma conexão e *cleanSession false*, a sessão recupera as subscrições e parâmetros daquele *clientID* e as mensagens possivelmente perdidas são recebidas.
- **username**. Credenciais de autenticação e autorização junto ao broker. Observa-se que o usuário deve registrar-se previamente no *broker*.
- **password**. Credenciais de autenticação e autorização junto ao broker.
- **lastWillTopic**. Quando a conexão for encerrada inesperadamente, o *broker* publicará automaticamente uma mensagem de "último desejo" em um tópico.
- **lastWillMessage**. A própria mensagem de "último desejo"enviada para o *lastWillTopic*.
- **keepAlive**. Intervalo de tempo em que o cliente precisa efetuar *ping* no *broker* para manter a conexão ativa.

SUBSCRIBE. Enviada para informar ao *broker* os tópicos de interesses para recebimento de notificações.

- **tópico**. Referência do tópico para subscrever.

- **qos**. Indica a semântica de entrega e confirmação com que as mensagens neste tópico precisam ser entregues aos clientes, segundo a Tabela 3.1.

Tabela 3.1. Níveis de QoS no MQTT.

Nível	Semântica	Descrição
0	no máximo uma vez (*at most once*)	A mensagem de confirmação não é enviada pelo receptor. É conhecido como "Fire and Forget". Pode ser mais eficiente.
1	pelo menos uma vez (*at least once*)	A mensagem será entregue ao receptor pelo menos uma vez, porém, a tentativa de entrega pode ocorrer mais de uma vez. O publicador precisa guardar a mensagem em um *buffer* local até o recebimento da confirmação pelo receptor.
2	exatamente uma vez (*exactly once*)	A mensagem será recebida, e apenas uma vez. É o nível mais seguro, mas também pode ser menos eficiente. Neste caso, tanto o publicador quanto o receptor, tem a certeza do recebimento da mensagem pelo destinatário.

PUBLISH. Este tipo de mensagem é usado em dois momentos. Primeiro, pelo cliente ao *broker* para a publicação de algum conteúdo em algum tópico. Em seguida, o *broker* realiza as verificações necessárias e retransmite esta mensagens a cada um dos clientes subscritos.

- **topicName**. Referência do tópico no qual a mensagem é publicada. Se o tópico ainda não existe, ele é criado.
- **qos**. Nível de qualidade de serviço da entrega da mensagem. Observe-se que o nível que QoS configurados nas mensagens *PUBLISH* e *SUBSCRIBE* são independentes, pois dizem respeito à comunicação entre o cliente o *broker*.
- **topicMessage**. Informação que dever ser transportada na mensagem. Pode ser uma sequência de caracteres ou um *blob* binário de dados.

Na Seção 3.5.4 um exemplo prático ilustra um ambiente de operação MQTT o uso de biblioteca com API para montar as mensagens do protocolo.

3.4.4. AMQP

O *Advanced Message Queuing Protocol*, AMQP [31], é um sistema de comunicação orientado à filas de mensagens, projetado e desenvolvido pela JPMorgan Chase em 2003. O AMQP é, atualmente, um padrão ISO/IEC (versão 1.0), OASIS. A norma está descrita no documento ISO/IEC 19464:2014 [52].

Como todo sistema orientado a filas, o AMQP tem em sua arquitetura um gerenciador de filas e um roteador de mensagens, além do protocolo de envio, recebimento e enfileiramento de mensagens. Este conjunto de elementos é geralmente chamado de *middleware* orientado a mensagens, MOM (*message oriented middleware*).

Para situar este tipo de infraestrutura, temos a analogia com sistemas de correio eletrônico que utiliza elementos chamados Agentes de Troca de Mensagens (*Message Transfer Agents*, MTA), que poderiam ser baseados no AMQP.

A arquitetura do AMQP versão 0-9-1, utilizada neste capítulo, apresenta os seguintes elementos, coletivamente chamados de *entidades* do *broker* AMQP (Figura 3.10)[4]:

- **Produtores** ou **Publicadores**. Clientes que enviam mensagens a um ou mais destinatários. Ao publicar uma mensagem o produtor pode especificar atributos de mensagens (meta-dados), que podem ser usados pelo *broker*.
- **Consumidores** ou **Subscritores**. Clientes que recebem mensagens de uma fila.
- ***Exchage***. Recebe mensagens dos produtores e as encaminha para uma fila de mensagem. Este encaminhamento pode utilizar políticas parametrizadas por propriedades extraídas da própria mensagem ou de seu conteúdo, ou segundo regras chamadas *Bindings*.
- ***Binding* (ligação)**. Regra que define o relacionamento entre um *Exchage* e uma e Fila de Mensagens, incluindo critérios de roteamento/encaminhamento.
- **Fila de mensagens**. Armazena mensagens até que elas possam ser processadas com segurança por um consumidor ou múltiplos consumidores. Cada fila é criada com um nome, que pode ser usado como chave de roteamento pelo *Exchange*.

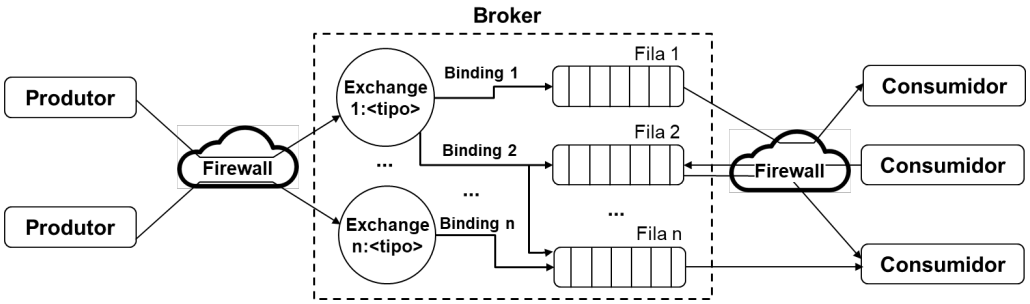


Figura 3.10. Elementos e entidades do *Broker* AMQP.

As mensagens são transportadas de forma binária, e o TCP é utilizado como protocolo de transporte. A camada SSL/TLS pode ser usada para comunicação segura entre os elementos. Para ter acesso ao servidor, aplicações produtoras e consumidoras precisam estabelecer uma conexão e se autenticar no *broker*. A conexão é mantida pela aplicação enquanto houver interesse pelos serviços do AMQP. Esta característica também facilita o acesso ao serviço mesmo que o cliente esteja em redes com NAT e protegida por *firewall*.

O padrão inclui características configuráveis de confiabilidade, segurança e interoperabilidade. O AMQP pode emular o estilo de comunicação *request-response* e também o *publish-subscribe*. Isso é facilitado pelo uso de um elemento central, o *broker* AMQP,

[4]A versão 1.0 apresenta um arquitetura e nomenclatura de elementos um pouco diferentes.

para intermediar a troca de mensagens e, também, pela flexibilidade dos elementos como o *Exchange* e o gerenciamento das Filas de Mensagens. Também é possível oferecer combinações de características não-funcionais como prioridade de filas ou filtros, com o uso de *Bindings*. A confiabilidade da entrega de mensagens, baseada em mensagens de controle *ACK* é um outro ponto positivo do AMQP.

O protocolo para troca de mensagens do AMQP é neutro e aberto, facilitando o envio de qualquer informação e a interoperabilidade entre diferentes plataformas. Além disso, o protocolo é programável, contribuindo para a facilidade de configuração de características oferecidas pelo *broker*. A configuração das entidades, regas e políticas é realizada pela própria aplicação, através de mensagens específicas, e não pelo administrador do servidor. Assim, uma aplicação pode declarar as entidades necessárias, esquemas de roteamento e o comportamento de filas. Uma das configurações possíveis para a aplicação é selecionar o tipo *Exchange* a ser utilizado:

- **Direto, *amq.direct* ou sem nome**, tipo mais simples, padrão para toda fila criada, utilizado para comunicação ponto a ponto ou *unicasting*. Roteia as mensagens segundo a chave de roteamento (na verdade, um nome associado ao *Exchange* e a uma *Fila*).
- **Fanout, *amq.fanout***, utilizado para comunicação *1*-para-*N* ou comunicação *broadcast*. A chave de roteamento é ignorada e uma cópia de mensagem é colocada em todas as filas associadas ao *Exchange*. Este tipo de *Exchange* pode ser usado para aumentar a escalabilidade.
- **Tópico, *amq.topic***, utilizando emular o estilo *publish-subscribe*, ou comunicação *1*-para-*N* ou *N*-para-*N*, ou comunicação *multicasting*. As mensagens são roteadas para uma ou mais filas, com base na chave de roteamento da mensagem e em um padrão usado para associar a *Fila* ao *Exchange*.

Além do tipo, é possível selecionar atributos do *Exchange* como: o nome específico (uma cadeia de caracteres), se serão duráveis ou não duráveis (são restaurados ou não após uma reinicialização do servidor), e se têm metadados associados a eles na criação (detalhes em [53]).

Depois que as mensagens são recebidas pelo *Exchange* e roteadas de acordo com os *Bindings*, estas seguem para suas respectivas filas. As mensagens são armazenadas em uma base de dados interna e gerenciada pelo *broker* para posterior recuperação de clientes interessados. As Filas de Mensagens no AMQP também têm atributos como o nome e durabilidade, pode ser exclusiva ou compartilhada, apagada automaticamente, além de metadados associados na declaração. Selecionando propriedades adequadas, as Filas de Mensagens podem oferecer suporte para a interação entre produtores e consumidores com características específicas:

- **Armazena e encaminha (*Store and forward*)**. Armazena temporariamente as mensagens encaminhadas pelos produtores e as distribui para consumidores registrados de forma circular. Geralmente são criadas para longos períodos e compartilhadas por vários consumidores.

- **Resposta privada.** Recebe as mensagens dos produtores, e as retém até encaminhá-las para um único consumidor. Geralmente são temporárias, com nome atribuído pelo servidor e privativas para um único consumidor.

- **Subscrição privada.** Coleta mensagens de vários produtores, subscritos previamente, e as encaminha para um único consumidor, responsável pelas subscrições.

As possibilidades de configuração e a API do AMQP são amplas. Conforme mencionado, a criação de *Filas*, seleção de *Exchanges* e declaração de *Bindings*, além de várias outras ações, têm suporte da API. O leitor é orientado a consultar as referências para avaliar as opções. Combinações de *Exchanges*, Filas de Mensagens e configurações de roteamento e mensagens de confirmação *ACK* também são possíveis. Na documentação do *broker* RabbitMQ [53], signatário da versão 0-9-1 do AMQP, são apresentadas algumas destas combinações.

Clientes consumidores utilizam a família de métodos *consume* e *deliver* da API do AMQP. Podem também utilizar métodos do tipo *push*. Para usar esta API o consumidor deve se registrar na fila de interesse. Quando uma mensagem for colocada na fila, o *broker* entrega a mesma diretamente para o consumidor. Uma fila pode ter mais de um consumidor interessado. Cada registro ou subscrição tem uma identificação chamada *consumer tag*. Também é possível buscar uma mensagem usando métodos do tipo *pull*. Clientes produtores utilizam a família de métodos *put* e *publish*. Variações de operação para estas primitivas são previstas para que os clientes usufruam das diferentes configurações do *broker*.

A Seção 3.5.5 apresenta uma aplicação simples que ilustra as possibilidades básicas do AMQP e utiliza o RabbitMQ como *broker*.

Para os objetivos deste capítulo, um ponto ainda deve ser discutido. O AMQP, como acontece vários outros MOMs, é flexível e robusto. Para isso, é necessária a execução de muitos elementos de suporte no servidor, e o consumo de recursos computacionais é inevitável. O AMQP pode ser usado de forma federada para oferecer escalabilidade, adicionando mais complexidade ao suporte. Este tipo de de suporte é geralmente necessário em grandes sistemas de informação, orientado à negócios, *enterprise*. O IBM MQ, utilizado em sistemas corporativos e o Java Message Service, JMS, que pode ser integrado a sistemas com suporte do Enterprise Java Beans, são exemplos de MOM.

Então, qual seria a utilidade do AMQP em um contexto de IoT? Provavelmente o AMQP não vai ser utilizado diretamente no acesso à dispositivos com poucos recursos ou com interfaces de comunicação de baixa capacidade, como ocorre com o CoAP ou o MQTT. Bibliotecas AMQP ainda não estão amplamente disponíveis para o Arduino, por exemplo. Por outro lado, a confiabilidade e robustez do AMQP pode ser empregada com vantagens para trazer ou levar requisições, respostas e informações para dispositivos com mais recursos, como *smartphones*, ou ainda para organizar o registro de eventos e demais informações coletadas de sensores em um serviço na nuvem. O cenário da Figura 3.2 ilustra este ponto, e o estudo de caso apresentado na Seção 3.6 apresenta um caso prático.

3.4.5. XMPP

O *Extensible Messanging and Presence Protocol*, XMPP [32] é também um sistema orientado a fila de mensagens. Ele oferece um protocolo para troca de mensagens que usa XML para comunicação, o que inclui mensagens instantâneas, como o AMQP, e também suporte para os conceitos de presença e colaboração.

O XMPP utiliza o TCP como protocolo de transporte e suporta vários estilos de comunicação, sendo os principais o *request-response* e o *publish-subscribe*. A camada SSL/TLS pode ser usada para comunicação segura entre os elementos. Para ter acesso ao servidor, os clientes precisam estabelecer uma conexão e se autenticar.

O projeto original do XMPP era chamado de *Jabber*, com sua primeira versão lançada em 1999. O *Jabber* foi originalmente projetado para uso em aplicativos de mensagens instantâneas, sendo adaptado aos poucos, para utilização em outros cenários. O protocolo original teve seu nome modificado para XMPP. Atualmente, o XMPP tem sido utilizado para atender a diferentes requisitos não funcionais, como funcionamento em navegadores, utilizando extensões WebSocket 3.3, além de aplicações para IoT. É um protocolo aberto e padronizado desde 2011 pela Internet Engineering Task Force [54].

O XMPP é utilizado por aplicativos de Mensagens Instantâneas, *Chat* em Grupo, Jogos, Geolocalização e voz sobre IP. O WhatsApp, por exemplo, utilizava o XMPP em suas primeiras versões. O Google Talk também utiliza o suporte do XMPP.

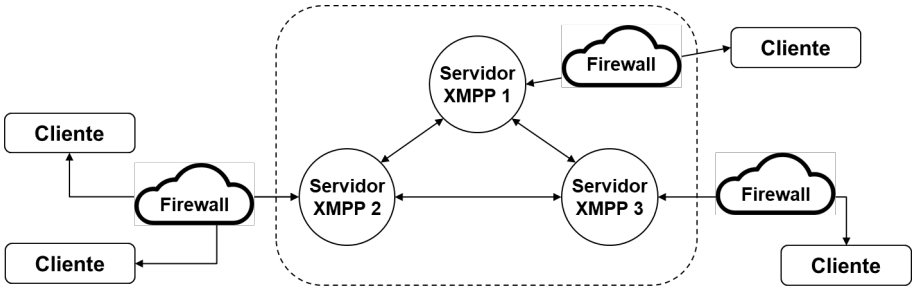


Figura 3.11. Servidores XMPP formando uma federação.

Sendo um sistema de mensagens, o XMPP também requer elementos gerenciadores de filas, chamados servidores de mensagens XMPP. O XMPP pressupõe uma rede federada de servidores, como mediadores no encaminhamento das mensagens. Isso permite que clientes distribuídos em redes protegidos por *firewalls* separados comuniquem-se uns com os outros. Cada servidor controla seu próprio domínio e autentica usuários nesse domínio. Os clientes de um domínio podem se comunicar com clientes em outros domínios através da federação. Os servidores XMPP estabelecem uma malha de conexões, de maneira segura, para trocar mensagens entre seus domínios. A Figura 3.11 ilustra este aspecto.

A autenticação de um cliente é feita usando uma arquitetura baseada em autenticação simples e camada de segurança (SASL) ou TLS/SSL.

A identidade de cada cliente é chamada de endereço XMPP ou *Jabber ID* (JID). Este identificador é semelhante à combinação de um endereço de e-mail e uma URL. Por

exemplo, um usuário com o nome “aluno” executando a aplicação “protocolosIoT” em “lcc.uerj.br” terá o seguinte JID:

`aluno@lcc.uerj.br/protocolosiot`.

Da mesma forma, recursos compartilhados, como salas de discussão, também são representados por JIDs. Por exemplo, “reuniao@lcc.uerj.br” representa uma discussão “reuniao” que está disponível no endereço “lcc.uerj.br”.

A comunicação XMPP consiste de fluxos bidirecionais de fragmentos XML chamados de *stream* e *stanza*, respectivamente. O *stream* é um contêiner para o troca de elementos, ou *stanzas* XML entre duas entidades quaisquer em uma rede. São especificadas três tipos de *stanza*:

- ***Presence***. Utilizado para enviar informações sobre o próprio usuário para outras partes autorizadas e interessadas. Um exemplo é o anúncio para estas entidades sobre sua disponibilidade ou presença (conectado ou desconectado). Este é um diferencial em relação ao modelo de serviços do AMQP.
- ***Message***. Utilizado para enviar mensagens assíncronas (sem aguardar resposta) para um determinado receptor.
- ***IQ* (*info-query*)**. Utilizada para enviar uma requisição que requer uma resposta da outra entidade, interação *request-response*. Existem tipos de *stanza* IQ pré-definidos: *get*, *set*, *result* e *error*. É obrigatório o uso de um campo *id*, para permitir a correlação de requisições e respostas, como ocorre com o *token* do CoAP.

Um fluxo típico de *stanzas* é apresentado na Figura 3.12. Um *stanza* é enviado para um destinatário e este a responde de forma apropriada. O processo de comunicação se dá da seguinte forma: (i) um cliente abre uma conexão com o servidor XMPP e inicia um *stream* XML; (ii) em seguida, existe a negociação de segurança e a autenticação; (iii) *stanzas* do tipo *Message* são, então, trocados e (iv) por último, o *stream* XML e a conexão são fechadas.

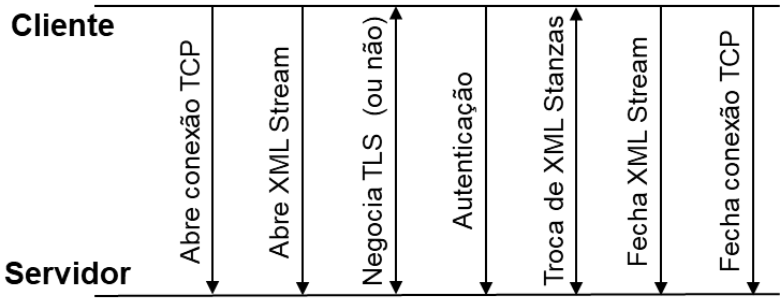


Figura 3.12. Troca de mensagens em um fluxo XMPP.

As bibliotecas de desenvolvimento XMPP, de forma geral, encapsulam a criação e manipulação dos *stanzas* XML, necessários para comunicação entre as entidades. Os *stanzas* gerados são simples. O Código 3.4 apresenta um *stanza* do tipo *Message*, com os elementos necessários.

```
<message xml:lang='en'
  to='aluno@lcc.uerj.br'
  from='alexszt@ime.uerj.br/dicc'
  type='chat'>
  <body>Podemos trabalhar no texto hoje?</body>
</message>
```

Código 3.4. *Stanza* XMPPdo tipo *Message*.

O exemplo no Código 3.5, mostra um *stanza IQ* com uma requisição, e um *stanza IQ* com a resposta retornada pelo servidor XMPP. A requisição solicita ao servidor uma lista de recursos suportados. Esta funcionalidade é útil para o cliente realizar verificações antes de tentar utilizar algum serviço do servidor XMPP.

O *stanza* de requisição tem tipo *get* (linha 5) e precisa de um identificador único (linha 3). O identificador é usado para correlacionar a resposta/confirmação com a requisição. A solicitação específica é descrita em uma *query* qualificada por um *namespace*, neste caso, referenciado por *disco#info*, indicando um pedido de informações ao serviço de descoberta (linha 6). Outros tipos de solicitação e extensões estão disponíveis.

```
1  <!--Requisição-->
2  <iq from="roberto@lcc.uerj.br"
3    id="123456"
4    to="lcc.uerj.br"
5    type="get">
6    <query xmlns="http://jabber.org/protocol/disco#info"/>
7  </iq>
8  <!--Resposta-->
9  <iq from="lcc.uerj.br"
10   id="123456"
11   to="roberto@lcc.uerj.br/6bcfui1lpj"
12   type="result">
13   <query xmlns="http://jabber.org/protocol/disco#info">
14     <identity name="Openfire Server" category="server" type="im"/>
15     <identity category="pubsub" type="pep"/>
16     [...]
17     <feature var="http://jabber.org/protocol/pubsub#subscribe"/>
18   </query>
19 </iq>
```

Código 3.5. *Stanza* XMPP *IQ* dos tipos *get* e *result*.

A resposta é retornada pelo servidor XMPP em um *stanza IQ* do tipo *result* (linha 12), com informações de suas capacidades. O mesmo identificador deve ser usado, linha 10, para permitir que o cliente correlacione a resposta com a requisição, dado que as mensagens em um fluxo XML são independentes. Observamos na resposta que a extensão *publish-subscribe* está disponível no servidor (categoria *pubsub*, linha 15), permitindo ao cliente tomar a decisão de usar este estilo de interação.

O XMPP fornece um protocolo aberto, fácil de usar, flexível e extensível. Isso faz com que a comunidade que utiliza o protocolo, crie uma série de extensões. Existe um fórum chamado XMPP Standards Foundation (XSF), que publica um conjunto de extensões, que são revisadas e discutidas, garantindo a interoperabilidade. Essas extensões são chamadas de XMPP Extension Protocols (XEPs) [55]. O XMPP incentiva e reúne as extensões relacionadas à IoT em `http://www.xmpp-iot.org/`.

O exemplo da Seção 3.5.6 ilustra o uso do XMPP através de chamadas de biblioteca, que tornam mais simples a montagem dos *stanzas*.

3.4.6. Comparação

Para concluir esta Seção, comparamos os protocolos apresentados de forma consolidada e complementamos o conjunto com algumas informações adicionais.

Na Tabela 3.2 reunimos características gerais. Como curiosidade, temos o ano em que cada protocolo foi proposto. Ainda que o uso no contexto de IoT seja recente, as propostas não são tão novas. Todas os protocolos estão atualmente padronizados por algum órgão ou entidade ligada à computação, telecomunicações ou sistemas distribuídos. Também listamos os principais sistemas de suporte para os protocolos. Na seção seguinte utilizaremos alguns deles para construir os exemplos. Destaque para o RabbitMQ, que oferece suporte ao MQTT e ao AMQP. Na última coluna consolidamos os principais estilos de interação suportados por cada protocolo.

Tabela 3.2. Características Gerais.

Protocolo	Ano	Padrão	Bibliotecas	Interação
HTTP	1997	IETF, W3C	Disponíveis para Java, CSharp, Javascript e outras	request-response
CoAP	2010	IETF, Eclipse Foundation	Esp-CoAP, Californium	request-response
MQTT	1999	OASIS, Eclipse Foundation	PubSubClient, RabbitMQ, Eclipse MQTT JS	publish-subscribe
AMQP	2003	OASIS, ISO/IEC	RabbitMQ	request-response, publish-subscribe
XMPP	1997	IETF, W3C	Jaxmpp2, OpenFire, Outras	request-response, publish-subscribe

A Tabela 3.3 reune algumas características ligadas à comunicação. Observa-se que o único protocolo que não utiliza o TCP é o CoAP. O CoAP foi projetado para ter pouco *overhead*, ou seja o menor impacto possível na comunicação. Entretanto por utilizar o UDP, é necessário um esforço maior para a configuração de *firewall* e *NAT*, por isso, a indicação direta do seu uso é em redes locais. Listamos, para rápida referência, as portas usadas por padrão em cada protocolo.

A coluna *Confiabilidade* indica como cada protocolo trata o requisito não-funcional de garantia de entrega das mensagens. Em alguns casos este requisito é tratado como QoS

(o MQTT coloca explicitamente desta forma). Preferimos usar o termo *confiabilidade*, pois o termo *QoS* é geralmente usado para várias características não-funcionais ligadas à garantias de tempo de resposta, latência, prioridade e variação de atraso. O AMQP, por exemplo, pode oferecer tratamento prioritário para filas específicas — e isso poderia ser considerado suporte à QoS. Mas, isso não está incluído na coluna. A confiabilidade de entrega é apoiada em dois elementos: o uso do TCP, que tem características de garantia de entrega, controle de erro e controle de congestionamento, por exemplo, e na transmissão de mensagens de confirmação (tipo *ACK*), que devem ser verificadas pela aplicação.

Tabela 3.3. Características de Comunicação.

Protocolo	Transp.	Porta	Confiabilidade	Autent.	Segur.
HTTP	TCP	80, 443 (TLS/SSL)	Confb. TCP	Sim	TLS/SSL
CoAP	UDP	5683, 5684 (DTLS)	Confirmado, Non-confirmable	Não	DTLS
MQTT	TCP [5]	1883, 8883 (TLS/SSL)	QoS 0 - At most once, QoS 1 - At least once, QoS 2 - Exactly once	Sim	TLS/SSL
AMQP	TCP	5672, 5671 (TLS/SSL)	Confb. TCP, mensagens ACK (versão 0.9.1)	Sim	TLS/SSL
XMPP	TCP	5222 (cliente), 5269 (servidor), 5223 (TLS/SSL)	Confb. TCP, mensagens ACK	Sim	TLS/SSL

Por último, características relacionadas ao controle de acesso e à privacidade. Com exceção do CoAP, os protocolos oferecem alguma forma de autenticação, restringindo o acesso à usuários (ou aplicações) que são autenticados. Os protocolos baseados em um elemento central como o *broker* permitem que serviços de terceiros para autenticação sejam utilizados. Para privacidade, opções de criptografia das mensagens são utilizadas.

3.5. Suporte, Bibliotecas e Exemplos

Nesta seção é apresentado um exemplo prático para cada protocolo discutido na Seção 3.4, considerando também o mecanismo WebSockets apresentado na Seção 3.3. A Figura 3.13 ajuda a entender o cenário usado como base. Nos exemplos, exploramos os usos mais indicados de cada protocolo, *request-response* ou *publish-subscribe*. Um dispositivo NodeMCU é utilizado na maioria dos exemplos [56]. O ambiente de programação é compatível com o Arduino.

Nosso NodeMCU hora atua como "servidor", que pode ligar ou desligar os LEDs vermelho e verde, e hora como publicador de eventos, que periodicamente publica informações. Um dispositivo Java/JavaScript, que pode ser um *smartphone* Android ou um nó Linux (ou Windows) que atuam como clientes e servidores completam o conjunto de elementos.

[5] UDP com MQTT-SN

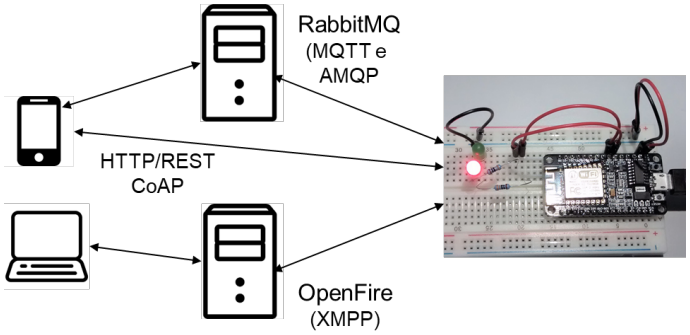


Figura 3.13. Cenário dos exemplos.

Java e o "dialeto" C para Arduino são utilizadas como linguagens de programação no desenvolvimento dos exemplos. Nos exemplos será possível verificar que, devido à limitações do Arduino, são necessárias estruturas de controle para os protocolos, que são opcionais, ou mais flexíveis, em outros ambientes. Por exemplo, é recomendando, por precaução, inserir no *loop* do Arduino instruções para reconexão. Para interação via MQTT sobre WebSockets, a linguagem de programação JavaScript é utilizada.

As ferramentas de desenvolvimento utilizados são comuns à vários ambientes de projeto: Android Studio para aplicações Android, NetBeans para desenvolvimento Java e JavaScript e ArduinoIDE para o desenvolvimento de aplicações para o NodeMCU. Entretanto, todos códigos apresentados nas próximas subseções podem ser editados em qualquer IDE compatível com a linguagem utilizada.

Completando o exemplo de cada protocolo, a biblioteca ou sistema de suporte utilizados são também apresentados. Antecipando, o *broker* RabbitMQ é usado como suporte para o MQTT e para o AMQP. Ele executa em um dispositivo RaspberryPi. Não existe razão específica para esta configuração além de mostrar que é possível fazer isso. O servidor OpenFire, que suporta o XMPP executa em um nó Windows 10. Outras bibliotecas e serviços de suporte são listados complementarmente no Apêndice 3.8.

3.5.1. WebSockets

O exemplo com o WebSocket mostra como esta tecnologia pode ser utilizada por navegadores, em uma comunicação bidirecional, com outros servidores ou clientes. Os estilos de interação apresentados anteriormente, bem como os Protocolos de Aplicação que utilizam TCP, podem utilizar-se do WebSocket como um mecanismo de transporte, similar ao uso do *socket* tradicional.

O exemplo apresentado possui um cliente que envia ao servidor um relatório a cada X unidades de tempo. Além disso, o cliente está preparado para receber, a qualquer momento, uma solicitação do servidor para que esse intervalo X seja alterado. O servidor é preparado para tratar a informação enviada pelo cliente e, a qualquer momento, através de uma mensagem, pode alterar a frequência com que essa informação é enviada.

Para a implementação deste exemplo foi utilizado um NodeMCU como o cliente, que gera relatórios periodicamente. Para implementar o servidor WebSocket foi utilizado a linguagem de programação Java.

Este cenário poderia ser utilizado em um ambiente de monitoramento, por exemplo, onde existem diversos sensores gerando relatórios para um único servidor. É possível ainda que esses papéis sejam invertidos, como por exemplo, em um cenário de Cidade Inteligente esses sensores poderiam ser disponibilizados como *beacons*, fazendo com que o nodeMCU seja o servidor WebSocket, e cada cliente interessado nas informações deveria estabelecer a comunicação.

O Código 3.6 apresenta o servidor Java desenvolvido para o cenário descrito acima. Utilizamos a classe *Server*, da biblioteca *Tyrus* que provê uma infraestrutura geral de um servidor WebSocket. Entretanto, com pequenas modificações é possível integrar este mesmo código com ambientes Web estruturados como o Apache Web Server [46] ou o Glassfish [57].

```
1 @ServerEndpoint(value = "/")
2 public class WebSocketServer {
3   public static void main(String[] args) {
4      Server server = new Server(
5         "0.0.0.0", 8080, "/", null,
6               WebSocketServer.class);
7      try {
8        server.start();
9        while(true){
10         Thread.sleep(10000);
11         for(Session s:list){
12           if(s.isOpen())
13           s.getBasicRemote().sendText(getJson());
14     } } }
15     catch (Exception e) { e.printStackTrace();}
16     finally { server.stop();}
17   }
18   @OnOpen
19   public void onOpen(Session session) throws IOException {
20       list.add(session);
21   }
22   @OnClose
23   public void onClose(Session session) throws IOException {
24       list.remove(session);
25   }
26   @OnMessage
27   public void onMessage(String message, Session sessao){
28     usaInformacao(message);
29 }}
```

Código 3.6. Servidor WebSocket escrito em Java.

O programa começa criando uma variável referente ao servidor WebSocket, indicando também as configurações desejadas. Neste exemplo o serviço deve ser disponibilizado para todas as interfaces disponíveis (0.0.0.0) na porta 8080. Além disso, é indicado também que a própria classe `WebSocketServer` irá responder aos eventos da conexão (linhas 4 a 6). Só então o servidor é iniciado na linha 8.

Conforme discutido na apresentação deste cenário, o servidor pode eventualmente enviar mensagens para os clientes para que eles alterem a periodicidade do envio de relatórios. Para fins de demonstração, o código apresentado envia estas mensagens a cada 10s para todos os clientes atualmente conectados (linha 11). Então na linha 13 é preparado um *JSON* e enviado como texto para os clientes. Para isso, uma lista concorrente com todos os clientes é mantida.

A biblioteca Tyrus [58], utilizada na implementação do servidor, permite que, através de anotações no código, determinados métodos funcionem como *callbacks* para eventos do WebSocket. Neste caso, a linha 20 será executada sempre que um cliente se conecta ao servidor pela primeira vez, então ele é adicionado à lista de clientes atual. De forma análoga a linha 16 é executada sempre que um cliente se desconecta. Por último, a linha 28 é executada sempre que o servidor recebe uma mensagem do cliente.

```
1 #include <WebSocketsClient.h>
2 WebSocketsClient webSocket;
3 void setup(){
4   configuraWifi(ssid, senha);
5   webSocket.begin(IP, PORTA, "/");
6   webSocket.onEvent(webSocketEvent);
7   webSocket.setReconnectInterval(5000);
8 }
9 void loop() {
10   webSocket.loop();
11   webSocket.sendTXT(getReport());
12   delay(intervalo);
13 }
14 void webSocketEvent(WStype_t type, uint8_t * payload, size_t length) {
15   switch(type){
16   case WStype_CONNECTED:
17     webSocket.sendTXT(thisDeviceInfo());
18     break;
19   case WStype_TEXT:
20     Json msg = Json.parseObject(toString(payload,length));
21     if(msg.success())
22       intervalo = msg["intervalo"];
23     break;
24 }}
```

Código 3.7. Código para o nodeMCU operando como cliente WebSocket.

O código 3.7 apresenta o programa do cliente, utilizando a biblioteca Arduino WebSocket [59]. Inicialmente, são configurados os elementos específicos, como por exemplo o Wi-Fi, sensores, atuadores, etc. Na linha 5 é iniciado o cliente WebSocket que deve se conectar ao *IP*, *PORTA* e caminho do servidor. Então na linha 6 é registrada a função responsável por responder aos eventos da conexão. A biblioteca, permite que outros parâmetros da conexão sejam editados, como por exemplo um tempo para se restabelecer a conexão (linha 7) automaticamente.

Com todas as configurações realizadas, o cliente passa a executar a função *loop*,

que realiza tarefas específicas da biblioteca (linha 10), gera um relatório e envia para o servidor (linha 11) e então, dorme por um determinado intervalo de tempo (linha 12).

Por ultimo, é definida a função que trata os eventos gerados pela conexão. Para manter a simplicidade do exemplo iremos apresentar os eventos: `WStype_CONNECTED` e `WStype_TEXT`, que representam, respectivamente, a confirmação de uma nova conexão estabelecida e o recebimento de uma mensagem de texto. A linha 17 é executada no momento que a conexão é estabelecida, neste caso, o dispositivo irá enviar um texto contendo informações sobre o próprio dispositivo para o servidor. Quando o dispositivo recebe uma mensagem de texto do servidor ele tenta interpretar como um JSON (linha 20) e então verifica se a mensagem é válida (linha 21). Caso positivo, o valor recebido será atribuído à variável intervalo (linha 22).

3.5.2. HTTP/REST

Como primeiro exemplo utilizando o estilo *request-response*, será usado o protocolo HTTP sobre REST. No cenário apresentado, um nodeMCU tem o papel de servidor e permite que um LED seja controlado através de um serviço REST.

O Código 3.8 apresenta de forma simplificada este servidor, enquanto o Código 3.9 apresenta um cliente implementado em Java.

Seguindo o Código 3.8, na linha 4 é configurado o servidor HTTP na porta 80. A escolha da porta é arbitrária, mas é comum que a porta 80 seja utilizada para aplicações HTTP. Em seguida, são realizadas as configurações específicas da aplicação para o Wi-Fi e para os LEDs.

Nas linhas 9 e 8 os serviços REST são configurados: o caminho — seguindo o exemplo apresentado na Seção 3.4.1, é utilizado `/servico/LEDService/vermelho` —, e qual função será responsável por tratar as requisições, de acordo com o método HTTP utilizado (*GET* ou *PUT*). Na linha 10 o servidor RESTful é, então, inicializado e fica disponível.

A função de *loop* (linhas 12 a 14) contém uma chamada específica para um procedimento da biblioteca utilizada, `ESP8266WebServer` [60]. Não será detalhada aqui.

A função responsável por tratar as requisições recebidas por mensagens HTTP com método *PUT* foi implementada de forma a utilizar um estrutura `if-else` para dividir as possíveis solicitações. A primeira parte verifica se a requisição é válida e contém todos os cabeçalhos necessários (linha 16). Se for reconhecido algum erro o servidor responde com um código 400 (*Bad Request*) e com um HTML de erro pré-definido (linha 17). Caso contrário, é verificado se a requisição é um pedido para desligar o LED vermelho (linha 21), o LED é desligado (linha 22) e então envia para o cliente um HTML específico (linha 23). Essa estrutura se repete de forma análoga para o caso de uma solicitação para ligar o LED vermelho (linhas 24, 25 e 26). Por último, é tratado as solicitações que possuem os cabeçalhos necessários, mas o conteúdo inválido, neste caso é enviando um HTML de erro (linha 28) com o mesmo código 400.

```
1 #include <ESP8266WiFi.h>
2 #include <ESP8266WebServer.h>
3 #define ledVermelho 13
4 ESP8266WebServer server(80);
5 void setup() {
6   configuraWifi(ssid, senha);
7   configuraLED();
8   server.on("/servicos/LEDServer/vermelho", HTTP_PUT, handlerPut);
9   server.on("/servicos/LEDServer/vermelho", HTTP_GET, handlerGet);
10  server.begin();
11 }
12 void loop() {
13  server.handleClient();
14 }
15 void handlerPut(){
16  if(!server.hasHeader("led")){
17    server.send(400, "text/plain", HTMLPageErro);
18    return;
19  }
20  String valor = String(server.header("led"));
21  if(valor == "0"){
22      digitalWrite(led_vermelho, 0);
23      server.send(200, "text/plain", HTMLPageLedDesligado);
24  }else if(valor == "1"){
25      digitalWrite(led_vermelho, 1);
26      server.send(200, "text/plain", HTMLPageLedLigado);
27  }else{
28      server.send(400, "text/plain", HTMLPageErro);
29 }}
30 void handlerGet(){
31  server.send(200, "text/plain", HTMLPageDefault);
32 }
```

Código 3.8. NodeMCU operando como servidor HTTP.

A função que responde as requisições *GET* e envia um HTML padrão com o código 200 (*OK*) é definida na linha 31.

Observa-se a facilidade para montar mensagens HTTP através de API utilizada, bastando chamar o método *send* e alguns parâmetros.

O Código 3.9 mostra o programa cliente, acessando o servidor e realizando uma solicitação para ligar o LED. Na linha 4 a URL para o recurso é montada e passada para a função que irá realizar a requisição. A linha 7 utiliza essa informação para criar um objeto que será utilizado para estabelecer a conexão com o servidor (linha 8). Com a conexão estabelecida o cliente pode receber dados através de um `InputStream` ou enviar dados através de um `OutputStream`.

Neste exemplo estamos interessados em enviar uma informação para o servidor através dos cabeçalhos do HTTP, então iremos utilizar o método PUT (linha 9) e adicionar o cabeçalho `led` com o valor 1 (linha 10). Para simplificar iremos armazenar a informação recebida em uma `String`(linha 11). A conexão é finalizada pelo cliente e por

último a resposta do servidor é utilizada pelo cliente, neste caso, ele simplesmente exibe a mensagem (linha 13).

```
1 public class HTTPClient {
2 public static void main(String[] args) throws Exception {
3   HTTPClient http = new HTTPClient();
4   http.sendPut("http://domínio/servicos/LEDServer/vermelho");
5 }
6 public void sendPut(String url) throws Exception {
7   URL obj = new URL(url);
8   HttpURLConnection con = (HttpURLConnection) obj.openConnection();
9   con.setRequestMethod("PUT");
10  con.setRequestProperty("led","1");
11  String response = con.getResponseCode();
12  con.disconnect();
13  System.out.println(response);
14 }}
```

Código 3.9. Cliente HTTP/REST escrito em Java.

É importante observar que dadas as características do HTTP/REST o cliente poderia ser simplesmente um navegador acessando este serviço através de uma URL. O Código 3.10 apresenta uma requisição realizada por um navegador e a resposta do servidor nodeMCU.

```
1 PUT /servicos/LEDServer/vermelho HTTP/1.1
2 Host: teste.lcc.uerj.br
3 Connection: keep-alive
4 Content-Length: 0
5 User-Agent: Mozilla/5.0 (X11; Linux x86_64)
6         AppleWebKit/537.36 (KHTML, like Gecko)
7         Chrome/65.0.3325.181 Safari/537.36
8 led: 1
9 Content-type: application/json; charset=UTF-8
10 Accept: */*
11 Accept-Encoding: gzip, deflate
12 Accept-Language: pt-BR,pt;q=0.9,en-US;q=0.8,en;q=0.7

13 HTTP/1.1 200 OK
14 Content-Type: text/plain
15 Content-Length: 22
16 Connection: close
```

Código 3.10. Mensagens de requisição e resposta HTTP com método *PUT*.

Para exemplificar iremos detalhar os cabeçalhos envolvidos na requisição *PUT*, que tem como objetivo alterar o estado do recurso, neste caso ligar o LED. A linha 1 apresenta o método HTTP utilizado (*PUT*), o caminho do recurso (`/servicos/LED/vermelho`)

e qual a versão do HTTP utilizada (1.1). Em seguida, na linha 2, é definido o *Host* para qual essa requisição será entregue. Só então os diversos cabeçalhos utilizados são definidos, entre eles a informação específica desta aplicação, identificada pela chave `led` (linha 8).

A resposta do servidor começa na linha 13 e contem a versão do HTTP utilizada (1.1) e o código de resposta (200). Além disso, é informado também qual o tipo (linha 14) e o tamanho (linha 15) da resposta. Por último, o servidor informa que irá fechar a conexão (linha 16).

3.5.3. CoAP

Para o exemplo do CoAP, será reusado o mesmo cenário do exemplo HTTP: o dispositivo NodeMCU será utilizado como servidor, e o cliente desenvolvido em Java tem como objetivo desligar o LED. Foi utilizada a biblioteca Esp-CoAP [61] para o servidor e para o cliente a biblioteca Californium [62]. Estas mesmas bibliotecas permitem que o NodeMCU seja utilizado como cliente e um dispositivo capaz de executar programas Java como servidor.

O Código 3.11 apresenta o servidor CoAP desenvolvido Nas primeira linhas são realizadas tarefas específicas, como a importação de bibliotecas, configurações do Wi-Fi e a configuração do LED. Na linha 5 é indicada qual função será responsável por responder às requisições e qual o caminho utilizado. Na linha 6 o servidor é, então, iniciado.

```
1 #include <coap_server.h>
2 void setup() {
3   configuraWifi(ssid, senha);
4   configuraLED();
5   coap.server(callbackLed, "ledVermelho");
6   coap.start();
7 }
8 void loop() {
9   coap.loop();
10  delay(500);
11 }
12 void callbackLed(coapPacket &pacote, IPAddress ip, int port, int obs) {
13   String message = char2string(pacote->payload);
14   if (message.equals("0"))    {
15     digitalWrite(led_vermelho, 0);
16     coap.sendResponse(ip,port,RespostaDesligado);
17   }else if (message.equals("1")){
18     digitalWrite(led_vermelho, 1);
19     coap.sendResponse(ip,port,RespostaLigado);
20   }else{
21     coap.sendResponse(ip,port,RespostaDefault);
22 }}
```

Código 3.11. NodeMCU operando como servidor CoAP.

A função *loop* deverá ser modificada de acordo com a biblioteca escolhida, no caso da Esp-CoAP o desenvolvedor deve chamar a biblioteca a cada 500*ms* (linha 9).

Por último a função `callbackLed` é executada sempre que o servidor recebe uma requisição no caminho declarado anteriormente. Na linha 13 o conteúdo da requisição é convertido para *String* a fim de facilitar o tratamento da mensagem. Neste ponto poderiam ser realizadas outras etapas como validação da mensagem, conversões de formato(JSON, XML, etc) ou até mesmo uma verificação de autenticidade através de uma assinatura digital, por exemplo.

De forma similar ao exemplo da Seção 3.5.2, esta função utilizou uma estrutura `if-else` para tratar as requisições. A primeira possibilidade trata uma requisição para desligar o LED, então é verificado o código recebido (linha 14), alterado o estado do LED (linha 15) e então enviado uma resposta para o cliente (linha 16). De maneira análoga as linhas 17, 18 e 19 tratam uma requisição para ligar o LED. Caso o código recebido não seja reconhecido, uma resposta padrão é enviada para o cliente (linha 21).

O Código 3.12 apresenta o cliente CoAP, desenvolvido em Java, que realiza uma solicitação para desligar o LED. Utilizando a URL do servidor, o cliente cria um objeto para intermediar a comunicação (linha 2). Na linha 3 o cliente realiza o procedimento para abrir uma determinada porta para receber a resposta do servidor e caso não aconteça nenhum erro, na linha 9 essa porta é atribuída a conexão.

```
1 public void coapRequest(){
2   CoapClient client = new CoapClient("coap://domínio/ledVermelho");
3   CoapEndpoint endpoint = new CoapEndpoint(PORTA);
4   try{
5     endpoint.start();
6   }catch(Exception ex){
7     trataExcecao(ex);
8   }
9   client.setEndpoint(endpoint);
10  Request request = new Request(CoAP.Code.PUT);
11  request.setPayload("0");
12  request.setType(CoAP.Type.CON);
13  CoapResponse response = client.advanced(request);
14  if (response != null){
15    System.out.println(response.getCode());
16    System.out.println(response.getOptions());
17    System.out.println(response.getResponseText());
18  }else {
19    System.out.println("Request failed");
20 }}
```

Código 3.12. Cliente CoAP escrito em Java.

Com os parâmetros da conexão definidos o cliente monta uma requisição *PUT* (linha 10) com os dados necessários para realizar a desativação do LED (linha 11) e a mensagem é configurada para o tipo *Confirmado*. Só então essa requisição é realizada. Como a requisição é *PUT* e o tipo *CON* foi configurado, uma resposta é aguardada de forma síncrona (linha 13). Após uma verificação da resposta (linha 14) o cliente pode informar ao usuário a resposta ou eventuais erros.

3.5.4. MQTT e WebSockets

No exemplo com MQTT, o módulo NodeMCU faz o papel de um cliente subscritor, através da biblioteca PubSubClient [63]. Um navegador foi usado como subscritor e como publicador, com o objetivo de mostrar o uso do MQTT com o WebSocket como mecanismo de transporte. Neste caso, a biblioteca utilizada foi a Eclipse Paho para Javascript [64]. O RabbitMQ [53], usado como *broker* neste exemplo, foi implantado em um RaspberryPi e disponibilizado para a conexão de clientes através do *número IP* e *porta*.

Neste cenário, o navegador no papel do publicador, pode enviar mensagens para um tópico do *broker* e este repassa esta informações para todos os subscritores interessados. Neste momento, tanto o módulo NodeMCU como o navegador, devem tratar a mensagem recebida. Para este exemplo, será utilizado o tópico `topic/led/vermelho` para controlar um LED no nodeMCU.

O Código 3.13 apresenta o programa em linguagem C para o nodeMCU como subscritor MQTT. As primeiras linhas do programa realizam algumas configurações específicas desse exemplo, como importar a biblioteca, configurar o Wi-Fi, etc. Na linha 6 é configurada a URL e a porta do *broker* ao qual o cliente deve se conectar. Já na linha 7 é indicada qual a função que responderá aos eventos de chegada de mensagens.

```
1 #include <PubSubClient.h>
2 PubSubClient MQTT(conexaoObj);
3 void setup() {
4   configuraWifi(ssid, senha);
5   configuraLED();
6   MQTT.setServer(URL, PORTA);
7   MQTT.setCallback(mqttCallback);
8 }
9 void loop() {
10   while (!MQTT.connected()){
11     if (MQTT.connect(ID,Usuario,Senha)){
12       MQTT.subscribe("topic/led/vermelho");
13     }else{
14       delay(500);
15   }}
16   MQTT.loop();
17 }
18 void mqttCallback(char* topic, byte* payload, unsigned int length) {
19   String message = char2string(payload);
20   if (message == "vermelhoon") {
21     digitalWrite(ledVermelho, 1);
22   else
23     digitalWrite(ledVermelho,0);
24 }}
```

Código 3.13. NodeMCU como subscritor MQTT.

De maneira similar aos exemplos anteriores que utilizaram o nodeMCU, a função `loop` contém as chamadas para a biblioteca utilizada, que precisam executar periodica-

mente. Verifica-se, a cada iteração, se a conexão foi perdida (linha 10) e, se necessário, ela é reestabelecida (linha 11). Na linha 12, o nodeMCU, então, subscreve o tópico `topic/led/vermelho`.

Quando uma mensagem é recebida do *broker*, *mqttCallback* é chamada. Primeiramente, a mensagem é tratada e convertida para `String` (linha 19). Observe-se que, de forma similar ao discutido na Seção 3.5.3, a rotina de tratamento de mensagem pode ser utilizado para outras tarefas além de uma simples conversão de formato. Após essa etapa, o conteúdo da mensagem é verificado (linha 20) e, então, o LED pode ser ligado ou desligado (linhas 21 e 23).

O Código 3.14 apresenta o publicador e subscritor MQTT desenvolvido em JavaScript para ser executado em navegadores. O código faz parte de uma página HTML, carregada pelo navegador, que iria prover campos para o usuário entrar com o texto da mensagem a ser publicada. Também seria provido um campo para visualização das mensagens recebidas, publicadas por outros clientes naquele determinado tópico.

```
1 var client = new Paho.MQTT.Client(URL,PORTA,ID);
2 client.onConnectionLost = function (responseObject) {
3   reconectar();
4 };
5 client.onMessageArrived = function (message) {
6   print(message.payloadString);
7 };
8 var options = {
9   timeout: 3,
10  userName: Usuario,
11  password: Senha,
12  onSuccess: function () {
13    client.subscribe('topic/led/vermelho', {qos: 1});},
14  onFailure: function (message) {
15    erro(message);},
16  useSSL: true
17  };
18 client.connect(options);
19 var send = function(data) {
20    message = new Paho.MQTT.Message(data);
21    message.destinationName = "topic/led/vermelho";
22    client.send(message);
23 };
24 form.submit(function() {
25                     send(input.val());
26                     input.val('');});
```

Código 3.14. Cliente JavaScript Publicador-Subscritor MQTT utilizando WebSocket.

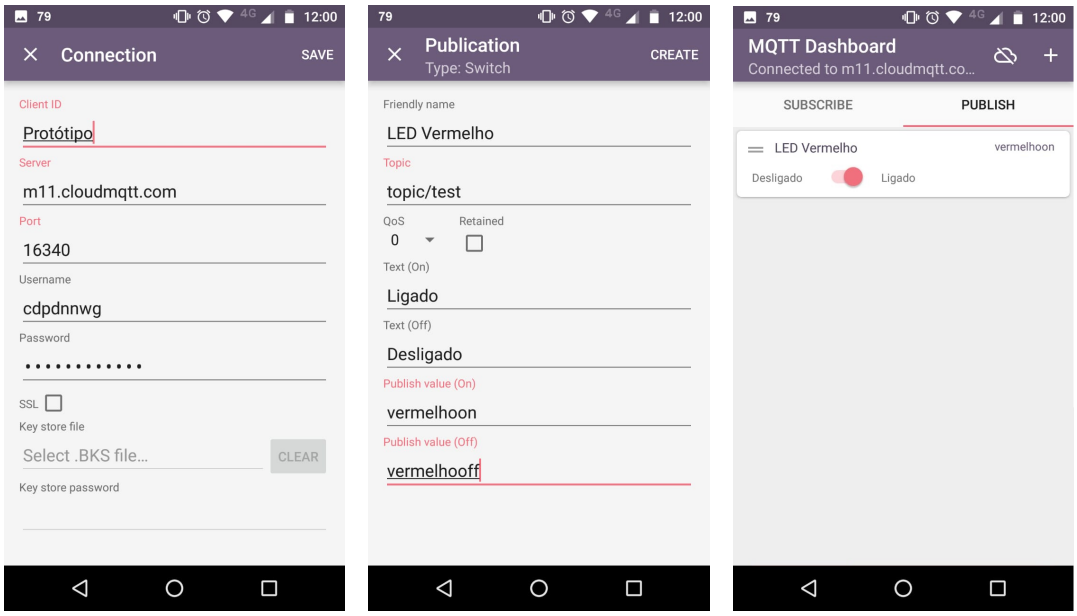
Na linha 1 é configurado a URL e a porta do *broker* e o ID do cliente para essa conexão. Este ID é uma *String* utilizada pelo *broker* para identificar de forma não-ambígua os clientes conectados. Em seguida na linha 3 é incluído um *callback* para a perda de conexão. De forma análoga, na linha 6 é determinado o comportamento do programa ao

receber uma mensagem, neste caso ele ira exibir a informação para o usuário.

Antes da conexão ser estabelecida mais alguns parâmetros são definidos (linha 8), como por exemplo o nome e senha do usuário (linhas 10 e 11), em quais tópicos está interessado como subscritor e seus respectivos níveis de QoS (linha 13), bem como o tratamento de erro 15. Com todos os parâmetros definidos, essas informações são passadas para o cliente criado e a conexão é estabelecida na linha 18.

Até então nosso cliente seria apenas um subscritor. Porém, na linha 19 é definida a função que trata o envio de mensagens para o *broker*. Nesta função é especificado o tópico onde as mensagens serão publicadas (linha 21) e, então, é realizado o envio das informações (linha 22). O conteúdo das mensagens é digitado pelo usuário em um campo de texto de um formulário HTML, que são passadas para a função detalhada acima (linha 24).

Existem ferramentas úteis para prototipação e testes com o MQTT. Estas ferramentas exploram a simplicidade da API e dos tipos de mensagens do protocolo para oferecer ao programador uma forma prática para testar a interação com o *broker*, mesmo sem programas desenvolvidos. Incluímos aqui um exemplo de teste utilizando a ferramenta IoT MQTT Dashboard [65]. Na Figura 3.14, temos a sequência de configurações do servidor (Figura 3.14(a)) e do tópico (Figura 3.14(b)). Em seguida, um teste de publicação no tópico (Figura 3.14(c)).



(a) Configuração do servidor. (b) Configuração do tópico. (c) Publish (on-off).

Figura 3.14. Ferramentas de prototipação e testes para MQTT.

3.5.5. AMQP

Para demonstrar o uso do AMQP, foram desenvolvidos dois clientes: um publicador e um subscritor. Ambos foram desenvolvidos na linguagem de programação Java, fazendo uso do *broker* e da API do RabbitMQ [53]. O suporte do RabbitMQ ao MQTT e ao AMQP facilita a interoperabilidade, como será demonstrado na Seção 3.6, e certamente

facilitou a implementação do exemplo.

Para este exemplo, o Código 3.16 apresenta um publicador que envia uma mensagem de "Hello World"para o *broker*, que então a repassa para os respectivos interessados, neste caso o cliente apresentado no Código 3.17.

Independente do papel do cliente AMQP, publicador ou subscritor, ele primeiro deve se conectar ao *broker*. Como o estabelecimento da conexão independe do papel do cliente, a rotina foi apresentada separadamente no Código 3.15. Na linha 3 é criada uma fábrica de conexões e com ela diversos parâmetros podem ser definidos. Para manter a simplicidade do exemplo, definimos apenas o IP do *broker* (linha 4) e então estabelecemos a conexão (linha 5). Para este cenários os dois clientes vão se conectar na mesma fila, para isso eles devem criar um canal (linha 6), o que permite que a mesma conexão TCP estabelecida com o *broker* seja reutilizada em diversos momentos. Só então, o cliente declara uma fila no *broker* (linha 7).

Por padrão a biblioteca irá criar a fila, caso ela não exista, ou então retornar a referência para a fila existente. Além disso, esta fila criada será, por padrão, "não-exclusiva", não será descartada automaticamente quando não houver mais clientes conectados e não será persistida caso o servidor tenha que ser reiniciado. É importante notar que a biblioteca utilizada oferece esta mesma função com mais parâmetros para que o programador possa customizar a fila de acordo com sua necessidade.

```
1 import com.rabbitmq.client.*;
2 /.../
3 ConnectionFactory fabrica = new ConnectionFactory();
4 fabrica.setHost(IP);
5 Connection conexao = fabrica.newConnection();
6 Channel canal = conexao.createChannel();
7 canal.queueDeclare(FILA);
```

Código 3.15. Cliente AMQP realizando uma conexão com o broker, escrito em Java.

Caso o cliente queira exercer o papel de publicador ele também deve utilizar o Código 3.16. Para isso ele deve apenas criar uma mensagem (linha 1) e então utilizar o canal para publicar (linha 2).

```
1 String mensagem = "Hello World!";
2 canal.basicPublish("", FILA, null, mensagem.getBytes());
```

Código 3.16. Cliente AMQP no papel de Publicador, escrito em Java.

No momento da publicação podem ser definidas diversas propriedades para aquela mensagem. No exemplo apresentado foi utilizado o *Exchange* do tipo direto, representado pela *String* vazia no primeiro parâmetro da função, e não foram criadas propriedades personalizadas, representadas por *NULL* no terceiro parâmetro da função. Essas propriedades

poderiam ser utilizadas, por exemplo, para determinar o tipo da mensagem que esta sendo enviada, uma data de validade, cabeçalhos ou prioridade (Seção 3.4.4).

Caso o cliente esteja interessado nas mensagens de uma fila ele deve utilizar o Código 3.17. Para isso, é preciso declarar um consumidor (linha 1) que irá tratar cada mensagem. Neste caso do exemplo, a informação é exibida (linha 8). Neste ponto, o cliente poderia recuperar todas as propriedades personalizadas definidas anteriormente pelo publicador através do parâmetro `properties` recebido pela função. Uma vez criado o consumidor, este deve ser registrado no canal com a informação da fila (linha 11). No exemplo, o cliente definiu que a biblioteca deve tratar o envio das mensagens de confirmação(*ACK*) através do segundo parâmetro da função.

```
1 Consumer consumidor = new DefaultConsumer(channel) {
2   @Override
3   public void handleDelivery(
4                     String consumerTag,
5                     Envelope envelope,
6                     AMQP.BasicProperties properties,
7                     byte[] body) throws IOException {
8     String mensagem = new String(body, "UTF-8");
9     print(mensagem);
10 }};
11 canal.basicConsume(FILA, true, consumidor);
```

Código 3.17. Cliente AMQP no papel de Subscritor, escrito em Java.

3.5.6. XMPP

Para o exemplo com XMPP, uma aplicação cliente Java foi desenvolvida, utilizando a biblioteca Jaxmpp2 [66]. Como servidor XMPP, foi usada a implementação OpenFire [67], instalada em um ambiente Windows. O exemplo está dividido em três partes: (i) Conexão, (ii) Publicador e (iii) Subscritor.

De forma similar ao apresentado na Seção 3.5.5, o procedimento para estabelecer conexão não sofre modificações entre o publicador e o subscritor e é apresentado separadamente no Código 3.18. Como primeiro passo, é necessário criar um objeto da biblioteca (linha 1), que irá permitir a configuração de diversos parâmetros da conexão como o *Jabber ID* (linha 3), a senha (linha 4) e aspectos de segurança da conexão (linha 5).

Além disso, diversos módulos podem ser utilizados para determinar o comportamento do cliente, como, por exemplo, o módulo de presença, que informa o *status* (online/offline) do usuário para os outros clientes (linhas 6 a 8).

Com todas as configurações realizadas, o módulo de publicador/subscritor é registrado (linha 9) e, então, realizada a conexão (linha 10). Observa-se que as mensagens para registro e notificação *publish-subscribe* são montadas pela biblioteca como *stanzas* IQ de tipo *put*, com uma extensão XMPP já disponível e documentada, utilizando uma *tag* específica chamada *pubsub*.

```
1  Jaxmpp jaxmpp = new Jaxmpp();
2  UserProperties up = jaxmpp.getProperties();
3  up.setUserProperty( SessionObject.USER_BARE_JID, JID );
4  up.setUserProperty( SessionObject.PASSWORD, Senha );
5  jaxmpp.getConnectionConfiguration().setDisableTLS(false);
6  PresenceModule.setPresenceStore(
7         jaxmpp.getSessionObject(),
8         new J2SEPresenceStore());
9  jaxmpp.getModulesManager().register(new PubSubModule());
10 jaxmpp.login();
```

Código 3.18. Exemplo de Conexão XMPP, escrito em Java.

O Código 3.19 apresenta o cliente que exerce o papel de subscritor. Na linha 1 é obtida uma referência direta para o módulo de publicador/subscritor que será utilizada para registrar o interesse deste cliente em um determinado tópico (linha 4). É importante notar que são utilizados dois *Jabber ID* diferentes, um para especificar o domínio de interesse e o outro para identificar o usuário.

Ao se registar em um determinado tópico, o cliente passa, como um dos parâmetros, uma função de *callback* (linha 5) que deve tratar os eventos relacionados ao estouro de temporizador (linha 6), estabelecimento da conexão (linha 9) e possíveis erros (linha 14).

```
1  PubSubModule pubsub = jaxmpp.getModule(PubSubModule.class);
2  BareJID jid = JID-Dominio;
3  JID subJid = JID-Usuario;
4  pubsub.subscribe(jid, Topico, subJid,
5    new PubSubModule.SubscriptionAsyncCallback() {
6    public void onTimeout() throws JaxmppException {
7    }
8    protected void onSubscribe(IQ resp,
9              PubSubModule.SubscriptionElement element) {
10   }
11   protected void onEror(IQ resp,
12                 XMPPException.ErrorCondition e1,
13                 PubSubErrorCondition e2)
14                 throws JaxmppException {
15   }});
```

Código 3.19. Cliente Subscritor XMPP, escrito em Java.

O Código 3.20 apresenta o cliente como publicador. Novamente, ele deve primeiro obter a referência para o módulo publicador/subscritor (linha 1). Só então o cliente cria a mensagem (linha 2) e adiciona o conteúdo (linha 3).

Com a mensagem construída o cliente pode utilizar a referência do módulo, junto com a identificação do tópico e o *Jabber ID* para publicar a mensagem (linha 4). O cliente, então, registra uma função de *callback* que deve tratar eventos de erro (linha 7), sucesso (linha 10) e estouro de temporizador (linha 13).

```
PubSubModule pubsub = jaxmpp.getModule(PubSubModule.class);
Element payload = ElementFactory.create("pubsub", null, XML-Schema);
payload.setValue(Conteudo);
pubsub.publishItem(jid, Topico, null, payload, new AsyncCallback() {
  public void onError(Stanza resp,
                      XMPPException.ErrorCondition e)
                      throws JaxmppException {
  }
  @Override
  public void onSuccess(Stanza resp) throws JaxmppException {
  }
  @Override
  public void onTimeout() throws JaxmppException {
  }});
```

Código 3.20. Cliente Publicador XMPP, escrito em Java.

3.6. Estudo de Caso - Gateway MQTT-AMQP

Nesta seção integramos dois protocolos e servidores apresentados na Seção 3.5 para construir um estudo de caso mais estruturado. A Figura 3.15 apresenta o cenário proposto.

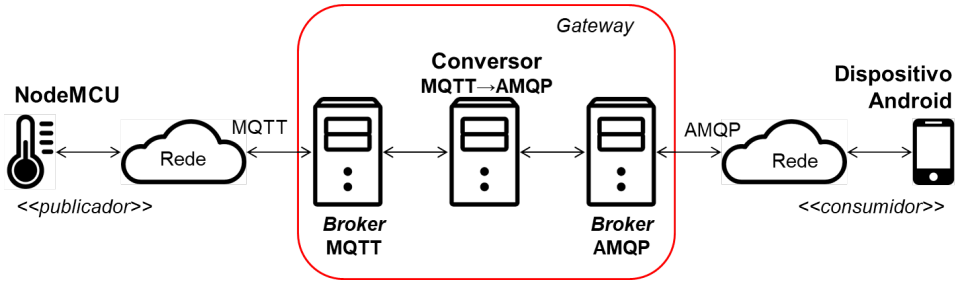


Figura 3.15. Estudo de caso: conversão MQTT para AMQP.

Clientes. Um NodeMCU equipado com um sensor de temperatura *DHT11* periodicamente publica o valor da temperatura em um determinado tópico MQTT. Uma aplicação Java, executando em um *smartphone* Android opera como consumidor AMQP e se inscreve em uma fila para obter o valor da temperatura.

Suporte. As mesmas bibliotecas e *brokers* usados nas Seções 3.5.4 e 3.5.5 serão reusados aqui. Os *brokers* executam em nós independentes, mas podem executar em um mesmo nó.

***Gateway*.** Tanto MQTT e o AMQP, utilizados neste estudo de caso, dão suporte ao estilo *publish-subscribe*, mas têm suas diferenças de funcionamento. O foco do exemplo é conciliar os dois protocolos, fazendo o papel de um *gateway*. Isso será feito com uma aplicação Java que acessa os dois *brokers*, covertendo as chamadas às APIs e adaptando os conteúdos quando necessário.

Observa-se que as aplicações-cliente são similares às utilizadas na seção anterior, deferindo essencialmente no conteúdo sendo publicado/recebido.

Cliente NodeMCU. O Código 3.21 apresenta uma parte da função *loop*, que neste exemplo tem por objetivo obter a temperatura do *DHT11* (linha 4), inserir o valor em uma mensagem de texto *JSON* (linhas 5 a 8) e publicá-la no tópico `uerj/sala01/temperatura` utilizando o protocolo MQTT (linha 9). O passo anterior, de conexão ao *broker*, é similar ao do exemplo da Seção 3.5.4, e por isso é omitido. Na linha 10 o programa *dorme* por cinco segundos, quando então volta a executar o método *loop* novamente.

```
1 char mensagem[tamanho];
2 void loop() {
3   /.../
4   float t = dht.readTemperature();
5   StaticJsonBuffer<200> jsonBuffer;
6   JsonObject& obj = jsonBuffer.createObject();
7   obj["valor"]=t;obj["unidade"]="C";
8   obj.printTo(mensagem);
9   MQTT.publish("uerj/sala01/temperatura",mensagem);
10  delay(5000);
11 }
```

Código 3.21. NodeMCU como publicador de temperatura MQTT.

Conversor MQTT-AMQP. O conversor é implementado como um serviço Java (Código 3.22). O programa pode ser dividido em cinco partes: (i) conexão ao servidor AMQP (linha 2); (ii) conexão ao *broker* MQTT (linha 3); (iii) subscrição ao tópico no *broker* MQTT, `uerj/sala01/temperatura`(linha 4); (iv) recebimento de uma mensagem através do MQTT (linha 6) e sua conversão; e, por último, (v) envio da mensagem convertida para uma fila AMQP (linha 10).

```
1 /.../
2 connectAMQPBroker();
3 connectMQTTBroker();
4 subscribeMQTT("uerj/sala01/temperatura");
5 /.../
6 public void onMessage(String msg, String topico) throws Exception {
7   String newJSON = adicionaDataAoJson(msg,System.currentTimeMillis());
8   publishAMQP(topico.replace('/', '.'),newJSON);
9 }
10 public void publishAMQP(String fila, String message){
11   channel.queueDeclare(fila, false, false, false, null);
12   channel.basicPublish("", fila, null, message.getBytes());
13 }
```

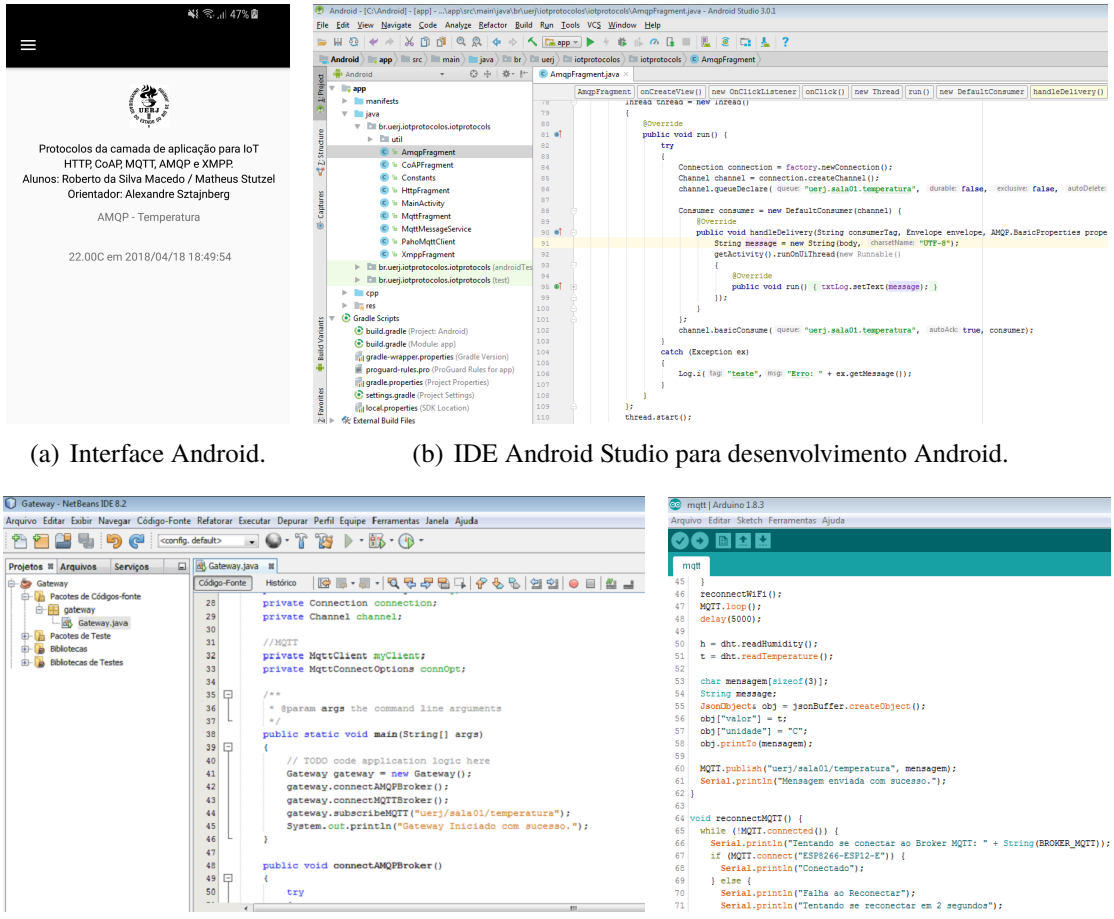
Código 3.22. Gateway realizando a conversão de MQTT para AMQP escrito em Java.

As três primeiras etapas foram apresentadas nas Seções 3.5.5 e 3.5.4, respectivamente, e são essencialmente as mesmas para o estudo de caso. As diferenças estão apenas no nome do tópico e da fila. Assim, destacamos as etapas (iv) e (v), discutindo as rotinas para a conversão entre os protocolos.

A etapa de recebimento e conversão de uma notificação MQTT, é representada no Código 3.22 pelo método de *callback* `onMessage`, linha 6, chamado toda vez que uma mensagem é recebida através do tópico MQTT subscrito por este cliente. Este método recebe como parâmetro a mensagem recebida e o tópico pelo qual ela foi recebida. Neste exemplo adicionamos a informação de data à mensagem recebida e salvamos a mesma novamente em uma estrutura *JSON* (linha 7). Então, a mensagem e o tópico são passados para a etapa (v) do código. Como precisamos conciliar o tópico MQTT com uma fila AMQP, adaptamos o nome original do tópico para o padrão aceito no AMQP. A solução adotada foi substituir o caractere "/" por ".", como mostrado na linha 8.

Por último, a etapa de retransmissão da informação para uma fila AMQP, é apresentada através da função `publishAMQP` (linha 10). Na linha 11 a fila é criada. Como visto anteriormente, caso a fila já exista apenas uma referência é retornada. Então, na linha 12, a mensagem é publicada na fila `uerj.sala01.temperatura` do servidor AMQP.

Cliente Adroid. O cliente executando num dispositivo Android deve se conectar ao *broker* AMQP, criar um canal para a fila desejada e então registrar um consumidor para as mensagens recebidas. Esse procedimento foi discutido na Seção 3.5.5 no Código 3.17.



(a) Interface Android. (b) IDE Android Studio para desenvolvimento Android.

(c) IDE NetBeans para desenvolvimento Java. (d) IDE para desenvolvimento Arduino.

Figura 3.16. Interface Android e IDEs para desenvolvimento.

Concluindo o estudo de caso, para ilustrar o ambiente de desenvolvimento utilizado e a interface desenvolvida, algumas telas foram reproduzidas na Figura 3.16. A interface Android é preparada para exibir a última temperatura e a hora recebida (Figura 3.16(a)), desenvolvida com a IDE Android Studio (Figura 3.16(b)). Os módulos do *gateway*, em Java, foram desenvolvidos com o NetBeans (Figura 3.16(c)). Cabe ao desenvolvedor obter e configurar as bibliotecas, no caso para acesso ao MQTT e AMQP, adequadamente na IDE. O módulo nodeMCU é programado no ambiente oferecido pelo Arduino, que integra o compilador e facilita o *upload* para o dispositivo (Figura 3.16(d)).

3.7. Conclusão

Neste capítulo foram apresentados os principais Protocolos de Aplicação utilizados em aplicações para a Internet das Coisas: CoAP, MQTT, AMQP e o XMPP. Além destes, o mecanismo de WebSocket e o uso de REST sobre HTTP também foram apresentados neste contexto. Todos são padronizados por organizações como o ITU-T, IEEE, e o IETF ou por organizações como o OASIS e o W3C.

Estes protocolos são suportados atualmente por bibliotecas disponíveis para, praticamente, todas as plataformas, incluindo dispositivos com poucos recursos. O CoAP e o MQTT foram concebidos considerando dispositivos e redes com recursos limitados.

O AMQP e XMPP são protocolos baseados em filas de mensagens, adequados para uso nos principais sistemas operacionais e plataformas. Não foram projetados inicialmente para IoT, mas tornaram-se parte integrante da solução, provendo a integração entre redes com dispositivos IoT, sistemas executando na nuvem e dispositivos móveis, permitindo a transferência de dados de forma síncrona e assíncrona entre estes elementos.

Observa-se que as soluções baseadas em um *broker* e em TCP também facilitam a comunicação elementos implantados em redes protegidas por NAT e *firewall*. A rede onde o *broker* executa precisa ser configurada com critério, mas as redes onde executam os clientes, em princípio, não precisam. Com isso, um ambiente inteligente implantado dentro de uma residência pode interagir com elementos em outras redes ou pode ser controlado externamente, desde que consiga manter uma conexão com um *broker*. O MQTT é um exemplo.

Os protocolos e mecanismos apresentados oferecem suporte à interação entre aplicações, dispositivos e serviços nos estilos *request-response* e *publish-subscribe*, além de variações destas, sobre UDP e TCP. As discussões e os exemplos desenvolvidos, envolvendo os vários mecanismos e bibliotecas disponíveis, permitem ao leitor decidir a direção a seguir em um próximo passo para experimentações com os protocolos apresentados.

Referências

[1] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things Journal*, vol. 1, pp. 22–32, Feb 2014.

[2] M. Díaz, C. Martín, and B. Rubio, "State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing," *Journal of Network and Computer Applications*, vol. 67, pp. 99 – 117, 2016.

[3] P. Bellavista, G. Cardone, A. Corradi, and L. Foschini, "Convergence of manet and wsn in iot urban scenarios," *IEEE Sensors Jrnl.*, vol. 13, pp. 3558–3567, Oct 2013.

[4] F. Kon and E. Zambom, "Cidades inteligentes: Tecnologias, aplicações, iniciativas e desafios," in *CSBC 2016. JAI 1. Porto Alegre, RS.*, 2016.

[5] B. P. Santos, L. Silva, C. Celes, J. Borges, B. Peres, M. Vieira, L. F. Vieira, and A. A. F. Loureiro, "Internet das coisas: da teoria à prática," in *SBRC 2016. Minicurso 1 (MC-1). Salvador, BA.*, pp. 256–315, 2016.

[6] F. Kon and E. F. Z. Santana, "Computação aplicada a cidades inteligentes: Como dados, serviços e aplicações podem melhorar a qualidade de vida nas cidades," in *CSBC 2017. JAI 4. São Paulo, SP.*, p. 2536, 2017.

[7] H. Sundmaeker, P. Guillemin, P. Friess, and S. Woelfflé, "Vision and challenges for realising the internet of things," Tech. Rep. 2, European Commission Information Society and Media, The address of the publisher, 3 2010.

[8] T. Teixeira, S. Hachem, V. Issarny, and N. Georgantas, *Service Oriented Middleware for the Internet of Things: A Perspective*, pp. 220–229. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[9] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys Tutorials*, vol. 17, pp. 2347–2376, Fourthquarter 2015.

[10] N. Naik, "Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http," in *2017 IEEE International Systems Engineering Symposium (ISSE)*, pp. 1–7, Oct 2017.

[11] L. Nastase, "Security in the internet of things: A survey on application layer protocols," in *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, pp. 659–666, May 2017.

[12] J. Ramirez and C. Pedraza, "Performance analysis of communication protocols for internet of things platforms," in *2017 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pp. 1–7, Aug 2017.

[13] CPSE Labs, "Fiware." Web page, 2017. Europ. Union for the development and global deployment of appl. for Future Internet, https://www.fiware.org/.

[14] cpse-labs.eu, "Sofia - smart objects for intelligent applications." Web pg., 2012. ARTEMIS project, http://www.cpse-labs.eu/sp_sofia.php.

[15] J. Soldatos, N. Kefalakis, M. Hauswirth, M. Serrano, J.-P. Calbimonte, M. Riahi, K. Aberer, P. P. Jayaraman, A. Zaslavsky, I. P. Žarko, L. Skorin-Kapov, and R. Herzog, *OpenIoT: Open Source Internet-of-Things in the Cloud*, pp. 13–25. Cham: Springer International Publishing, 2015.

[16] Particle, "The particle iot platform." Web page, 2017. https://www.particle.io/.

[17] P. S. Filho and R. Nascimento, "Knot: Integrando plataformas e aplicações de internet das coisas." X Escola Potiguar de Computação e suas Aplicações. Tutorial 2., 2017. Recife-PE. https://www.knot.cesar.org.br/.

[18] Eclipse IoT Working Group, "Iot standards." Web page, 2012. https://iot.eclipse.org/standards/.

[19] C. Pereira, A. Pinto, A. Aguiar, P. Rocha, F. Santiago, and J. Sousa, "Iot interoperability for actuating applications through standardised m2m communications," in *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pp. 1–6, June 2016.

[20] oneM2M, "Standards for m2m and the internet of things." Web page, 2017. http://www.onem2m.org/.

[21] I. P2413, "Standard for an architectural framework for the internet of things (iot)." http://grouper.ieee.org/groups/2413/, 2016. IEEE-SA Project. Chair: Oleg Logvinov (STMicroelectronics), Manager: Brenda Mancuso.

[22] OASIS, "Organization for the advancement of structured information standards." Web page, 2017. https://www.oasis-open.org/.

[23] ITU-T, "M2m service layer: Apis and protocols overview." ITU-T Focus Group on M2M Service Layer, 4 2014. https://www.itu.int/dms_pub/itu-t/opb/fg/T-FG-M2M-2014-D3.1-PDF-E.pdf.

[24] ITU-T, "Framework of constrained device networking in the iot environments." Global Information InfraStructure, Internet Protocol Aspects, Next-generation Networks, Internet of Things and Smart Cities, 9 2016.

[25] IETF, "The internet of things." Web page, 2017. https://ietf.org/topics/iot/.

[26] IEEE, "The internet of things." Web page, 2017. https://iot.ieee.org/.

[27] H. A. B. Pötter and Alexandre, "Adapting heterogeneous devices into an iot context-aware infrastructure," in *SEAMS '16*, (New York, NY, USA), pp. 64–74, ACM, 2016.

[28] Postscapes, "Iot gateways." Web page, 2017. http://www.postscapes.com/iot-gateways/.

[29] Mqtt.org, "Message queuing telemetry transport." Web page, 2017. http://mqtt.org.

[30] Z. Shelby, K. Harthe, and C. Bormann, "Rfc 7252 constrained application protocol (coap)." Web page, 2017. http://coap.technology/.

[31] AMQP.org, "Advanced Message Queuing Protocol (AMQP)." Web Page, 2014. https://www.amqp.org/.

[32] XMPP Standards Foundation (XSF), "Extensible messaging and presence protocol (xmpp)." Web page., 2011. https://xmpp.org/.

[33] P. M. Krishnapur, S. M, and C. A. Y., "Fast realtime data transfer using xmpp," in *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, pp. 477–479, July 2016.

[34] websocket.org, "About html5 websocket." Web page, 2017. https://websocket.org/aboutwebsocket.html.

[35] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*. USA: Addison-Wesley Publishing Company, 5th ed., 2011.

[36] IETF, "The WebSocket Protocol." IETF RFC6455., 2017. Web page. https://tools.ietf.org/html/rfc6455.

[37] W3C, "The websocket api," 2015. https://www.w3.org/TR/websockets.

[38] Node.js Foundation, "Node.js." Web page, 2018. https://nodejs.org/en/.

[39] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – http/1.1." RFC 2616. IETF. Web page, 6 1999. https://tools.ietf.org/html/rfc2616.

[40] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform resource identifier (uri): Generic syntax." RFC 3986. IETF. Web page, 1 2005. https://tools.ietf.org/html/rfc6690.

[41] D. Robinson and K. Coar, "The common gateway interface ver. 1.1." RFC 3876. IETF, 10 2004. https://tools.ietf.org/html/rfc3875.

[42] W3C, "Web services activity," 2011. https://www.w3.org/2002/ws.

[43] H. G. C. Ferreira, E. D. Canedo, and R. T. de Sousa, "Iot architecture to enable intercommunication through rest api and upnp using ip, zigbee and arduino," in *2013 IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 53–60, Oct 2013.

[44] F. Vásquez, "Prototype for monitoring patients at rest with wearable and iot technology (june 2017)," in *2017 IEEE Central America and Panama Student Conference (CONESCAPAN)*, pp. 1–6, Sept 2017.

[45] IETF, "RESTful Design for Internet of Things Systems." Web page. https://tools.ietf.org/id/draft-keranen-t2trg-rest-iot-04.html, 2017.

[46] The Apache Software Foundation, "Apache server." Web page, 2017. Apache Server, https://httpd.apache.org/.

[47] G. G. R. Gomes and P. N. M. Sampaio, "A specification and tool for the configuration of rest applications," in *2009 International Conference on Advanced Information Networking and Applications Workshops*, pp. 500–505, May 2009.

[48] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of ipv6 packets over ieee 802.15.4 networks." RFC 4944. IETF. Web page, 7 2007. https://datatracker.ietf.org/doc/rfc4944/.

[49] S. Bandyopadhyay, "Lightweight internet protocols for web enablement of sensors using constrained gateway devices," in *2013 International Conference on Computing, Networking and Communications, Workshops Cyber Physical System*, Jan 2013.

[50] IETF, "Constrained restful environments (core) link format," 8 2012. https://tools.ietf.org/html/rfc6690.

[51] ISO, "Message Queuing Telemetry Transport (MQTT) v3.1.1." OASIS Standard. ISO/IEC 20922:2016, 2016. Web page. Edited by Andrew Banks and Rahul Gupta. https://www.iso.org/standard/69466.html.

[52] ISO/IEC JTC 1 Information technology, "Advanced message queuing protocol (amqp) v0-9-1 specification." ISO/IEC 19464:2014, 2014. https://www.iso.org/standard/64955.html.

[53] Pivotal Software, Inc., "Rabbit MQ." Web page., 2017. Rabbit MQ Messaging Server, https://www.rabbitmq.com/.

[54] IETF, "Extensible messaging and presence protocol (xmpp): Core." Web page, 2011. https://tools.ietf.org/html/rfc6120.

[55] XMPP Extensions, "Extensible messaging and presence protocol (xmpp)." Web page. https://xmpp.org/extensions/, 2011. XMPP Extensions.

[56] NodeMcu, "Connect things easy," 2014. http://www.nodemcu.com.

[57] Oracle, "Oracle glassfish server." Web page, 2017. http://www.oracle.com/technetwork/middleware/glassfish/overview/index.html.

[58] Eclipse Foundation, "Eclipse tyrus." Web page, 2018. https://projects.eclipse.org/projects/ee4j.tyrus.

[59] https://github.com/Links2004, "Websocket server and client for arduino." Web page, 2018. https://github.com/Links2004/arduinoWebSockets.

[60] Arduino team, "Arduino core for esp8266 wifi chip." Web page, 2018. https://github.com/esp8266/Arduino.

[61] thingTronics Innovations, "Esp-coap server/client library for arduino." Web page, 2018. https://github.com/automote/ESP-CoAP.

[62] The Eclipse Foundation, "Californium." Web page, 2017. https://www.eclipse.org/californium/.

[63] N. O'Leary, "Arduino client for mqtt." Web page, 2017. https://pubsubclient.knolleary.net/.

[64] Eclipse Paho, "Eclipse paho javascript client." Web page, 2018. https://www.eclipse.org/paho/clients/js/.

[65] Nghia TH, "Iot mqtt dashboard." Google Play, 2018. https://play.google.com/store/apps/details?id=com.thn.iotmqttdashboard.

[66] Tigase, Inc., "Tigase JaXMPP Client Library." Web page, 2017. https://tigase.tech/projects/jaxmpp2.

[67] Ignite Realtime, "Openfire XMPP Server." Web page, 2017. https://www.igniterealtime.org/projects/openfire/.

3.8. Anexo

Aqui estão reunidas algumas referências para projetos e bibliotecas não utilizados nas seções anteriores, mas relacionados aos assuntos tratados.

Além do RabbitMQ, outros *brokers* e bibliotecas, estão disponíveis, como o Mosquito (https://mosquitto.org/), da Eclipse Foundation; ou o HIVEMQ (https://www.hivemq.com);

Existem instâncias de *broker* MQTT disponíveis na Internet. Nos testes e prototipação com o Dashboard utilizamos o servidor cloudmqtt.com;

ApacheMQ, sistema de mensagens Apache (http://activemq.apache.org/);

STOMP (Simple (or Streaming) Text Orientated Messaging Prot.). https://stomp.github.io;

Algumas extensões do XMPP são de destaque para uso na Internet das Coisas, como por exemplo, XEP-0060 - Publish-Subscribe, XEP-0030 - serviço de descoberta, XEP-0072 - SOAP Over XMPP, XEP-0239 - Binary XMPP;

Lightweight M2M (LWM2M). https://www.omaspecworks.org. Protocolo de gerenciamento de dispositivos e outras especificações;

CoAP.net (CoAP para .NET). https://github.com/smeshlink/CoAP.NET;

Padrão DDS (Data Distribution Service Spec., da OMG). https://www.omg.org/spec/DDS;

Frameworks para o desenvolvimento de aplicações de ambientes inteligentes. *SmartHome* (https://www.eclipse.org/smarthome/) e OpenHAB, baseado no Smart Home (https://www.openhab.org/) usando MQTT;

DOJOT - Com base no FIWARE, este é uma plataforma brasileira de desenvolvimento de soluções IoT para cidades inteligentes. http://www.dojot.com.br/sobre-a-dojot-iot/;

JSXC (JavaScript XMPP Client), cliente XMPP em JavaScript (https://www.jsxc.org/). GetKaiwa, outra opção de um cliente, baseado em Web (http://www.getKaiwa.com). Clientes XMPP para outras linguagens de programação, podem ser encontrados em: https://xmpp.org/software/clients.html;

XMPP-IoT, página Web dedicada a promover a utilização do XMPP em soluções IoT. http://www.xmpp-iot.org/.

Capítulo 4

Redes Corporais Sem Fio e Suas Aplicações em Saúde

Vinicius C. Ferreira, Egberto Caballero, Robson Lima, Helga Balbi, Flávio L. Seixas, Célio Albuquerque e Débora C. Muchaluat-Saade

Resumo

Avanços na eletrônica permitiram o desenvolvimento de sensores biomédicos miniaturizados e inteligentes, que podem ser utilizados para monitorar o funcionamento do corpo humano. O uso da comunicação sem fio se mostrou uma alternativa adequada, que proporciona menor incômodo aos pacientes e maior custo-benefício. A fim de explorar plenamente os benefícios das tecnologias sem fio na telemedicina, um novo tipo de rede sem fio emergiu: as redes corporais sem fio (Wireless Body Area Networks) - WBANs. No entanto, desafios técnicos e sociais devem ser tratados para permitir sua adoção prática. Alguns desses desafios já são conhecidos de outros cenários, como os requisitos de operação para uma rede sem fio, a eficiência energética, os poucos recursos computacionais e a composição heterogênea da rede. Porém, alguns fatores como o uso do corpo humano como meio de propagação, os efeitos da radiação no tecido humano e variações na movimentação do corpo fazem das redes corporais sem fio um novo paradigma de redes de comunicação sem fio. O principal objetivo deste capítulo é a discussão sobre os principais conceitos, desafios e perspectivas para redes corporais sem fio, área de grande oportunidade para pesquisa e desenvolvimento.

Abstract

Advances in electronics have enabled the development of tiny yet and intelligent biomedical sensors to monitor the human body. The use of wireless technology has proved to be adequate for communication, providing less inconvenience to patients and being more cost-effective. In order to fully exploit the benefits of wireless technologies in telemedicine, a new type of wireless network has emerged: Wireless Body Area Networks (WBANs). However, technical and social challenges must be addressed to enable its practical adoption. Some of these challenges are widely known in other contexts, such as

wireless networks operating requirements, energy efficiency, low computational resources and a heterogeneous network composition. However, some factors such as the use of the human body as a propagation media, the effects of radiation on human tissue and variations in body movement make WBANs a new paradigm for wireless communication networks. The main goal of this chapter is to discuss the main concepts, challenges and perspectives for WBANs, a field of great opportunity for research and development.

4.1. Introdução

Redes corporais sem fio, ou WBANs (*Wireless Body Area Networks*), são redes que comumente englobam a utilização de uma coleção de dispositivos que possuem como características principais o baixo consumo energético, pequeno tamanho e peso e capacidade de comunicação sem fio nas proximidades do corpo. Tais dispositivos podem ser implantados sobre, dentro ou nos arredores do corpo humano. Estes dispositivos comumente são sensores e atuadores que podem monitorar funções do corpo e características do ambiente ao seu redor.

Uma das principais motivações para o surgimento de WBANs é o desenvolvimento de aplicações voltadas para a saúde. Tendo em vista o aumento da população idosa mundial em conjunto com o aumento de gastos relacionados à saúde e a necessidade de tratá-la de forma preventiva, o desenvolvimento de novas tecnologias escaláveis capazes de prover monitoramento de funções do corpo humano a baixo custo se torna interessante. Porém, a tecnologia não se restringe somente a aplicações médicas. Além de aplicações médicas, outros campos como entretenimento, esportes, jogos, aplicações militares, etc. podem se beneficiar desta nova tecnologia. No entanto, a fim de apoiar essas novas aplicações e suas demandas, os desafios técnicos e sociais devem ser superados. Dentre os desafios técnicos, podemos destacar o desenvolvimento de um sistema flexível de acordo com a demanda da aplicação em termos de atraso, vazão, tempo de vida da rede e consumo de energia. Dentre os desafios sociais, facilidade de uso, segurança e privacidade de dados, interoperabilidade, custo, segurança física e bem-estar são preocupações importantes.

As principais questões envolvidas na criação de uma WBAN são o impacto do dispositivo sem fio no corpo humano, o tempo de vida da bateria do dispositivo e a coexistência com outras tecnologias sem fio. A presença do corpo humano afeta a propagação do sinal, criando um canal de comunicação específico que deve ser considerado no desenvolvimento dos protocolos de comunicação. Tendo em vista que WBANs são comumente implementadas na banda ISM (*Industrial, Scientific and Medical*) de 2.4GHz, a utilização concorrente do canal por outras tecnologias de comunicação sem fio também deve ser considerada para que não haja falha devido a colisões e ruído. Para o prolongamento da vida útil da bateria e para evitar o aquecimento do tecido humano, mecanismos de economia energética devem ser implementados.

Estes sensores e atuadores são capazes de coletar amostras, monitorar, processar e comunicar diferentes sinais vitais e parâmetros fisiológicos ao médico responsável, provendo informações em tempo real sem ocasionar desconforto ao usuário. Com o uso da informação coletada em um grande período de tempo, o conhecimento sobre o estado de uma pessoa torna-se mais amplo.

Alguns padrões que são considerados para a implementação de WBANs são o IEEE 802.15.4 [IEEE Std 802.15.4 2006a], o IEEE 802.15.6 [IEEE Std 802.15.6 2012] e o *Bluetooth Low Energy* [SIG 2010]. O padrão IEEE 802.15.4, publicado em 2006, especifica as camadas física e de controle de acesso ao meio para comunicações sem fio de curto alcance com baixo consumo energético, baixo custo e baixas taxas de transmissão. O *Bluetooth Low Energy* (BTLE), publicado em 2010, é uma versão de baixo consumo energético do *Bluetooth* original, com o objetivo de operar em dispositivos alimentados por pequenas baterias, como sensores.

O padrão mais recente é o IEEE 802.15.6, publicado em 2012, que foi especificamente desenvolvido para comunicação sem fio dentro, sobre ou nos arredores do corpo humano. Suas características principais são baixo consumo de energia, alta confiabilidade, e alcance na área ao redor do corpo humano suportando diferentes taxas de transmissão que são destinadas a diferentes tipos de aplicações.

É previsto que redes WBAN interajam com outras tecnologias de comunicação, como *ZigBee*, redes de sensores, *Bluetooth*, redes Wi-Fi, redes celulares e sistemas de vigilância. Isto abre campo para a criação de novas aplicações, serviços e dispositivos que trarão maior qualidade de vida. Essas aplicações devem seguir o padrão ISO/IEEE 11073 [IEEE Std 11073-00103 2012]. Este padrão define normas para aplicações de Dispositivos Pessoais de Saúde (DPS), isto é, dispositivos que são utilizados pelos próprios usuários em suas casas, como por exemplo, termômetros, balanças e bombas de insulinas. Também nessa família de normas, é especificado um padrão de comunicação entre DPS e dispositivos gerenciadores, como, *smartphones*, *desktops* e *notebooks*.

Este capítulo aborda os principais conceitos, desafios e perspectivas para WBANs, tema que ainda carece de pesquisa e desenvolvimento. O restante deste texto é organizado da seguinte maneira. A Seção 4.2 apresenta uma visão geral de WBANs, tipos de sensores e arquitetura de rede. Na Seção 4.3, são descritos os protocolos de comunicação WBAN. A Seção 4.4 discute o padrão IEEE 11073, que estabelece normas para a implementação de dispositivos pessoais de saúde. A Seção 4.5 discute desafios e perspectivas futuras no cenário de pesquisa em WBANs. Finalmente, a Seção 4.6 traz as considerações finais.

4.2. *Wireless Body Area Networks*

As WBANs consistem de vários dispositivos heterogêneos que se comunicam entre si por meio de uma rede sem fio nos arredores do corpo humano. Esses dispositivos têm potencial para se comunicar com aplicativos em sistemas vestíveis para monitoramento da saúde humana [Rodgers et al. 2015]. Os tipos de dispositivos WBANs são:

1. *Nós sensores sem fio* respondem a estímulos físicos, coletam dados, processam e relatam essas informações usando comunicação sem fio.

2. *Nós atuadores sem fio* agem de acordo com dados recebidos de nós controladores, advindas de comandos externos ou interação do usuário.

3. *Nós controladores* agem como agregadores das informações coletadas pelos nós sensores, assim como enviam comandos aos nós atuadores.

A próxima seção descreve características de dispositivos médicos e seus requisitos de comunicação.

4.2.1. Dispositivos Médicos

Os dispositivos médicos são implementados como sensores ou atuadores, portáteis ou implantáveis, pequenos e flexíveis. Os sensores são capazes de medir sinais biológicos e transmitir dados através de comunicação sem fio para outros nós internos ou externos em uma rede de comunicação[Pantelopoulos and Bourbakis 2010, Latré et al. 2011].

Os principais componentes de um sensor médico são transceptor de rádio com uma antena para comunicação sem fio, microprocessador, memória, sensores bioquímicos ou analógicos (por exemplo, ECG - eletrocardiograma, glicose, temperatura) e uma bateria para alimentação elétrica. Esses nós normalmente usam um sistema operacional com capacidade de processamento limitada, poucos requisitos de memória e baixa sobrecarga do sistema. Um exemplo de sistema operacional usado em alguns sensores é o TinyOS de código aberto [Levis et al. 2005].

Um dispositivo médico pode ser colocado no corpo humano com um pequeno adesivo ou implante, ou pode ser colocado sob a roupa, permitindo atividade ubíqua, medições fisiológicas e ambientais no ambiente natural durante um longo período de tempo. Existem vários tipos de dispositivos médicos fisiológicos:

1. *Comprimidos* que contêm um transceptor sem fio e sensores que podem detectar enzimas, ácidos nucleicos, contrações do músculo intestinal, acidez intestinal, pressão e outros parâmetros, permitem monitoramento de doenças gastrointestinais de maneira não invasiva [Hao and Foster 2008].

2. *Dispositivos vestíveis* são sensores portáteis com um transceptor sem fio montado na superfície do corpo humano, como por exemplo, citam-se os *smart watches* ou mesmo sensores em forma de anel usados no dedo para monitorar a frequência cardíaca.

3. *Dispositivos fisiológicos implantáveis* podem ser implantados no paciente para operar dentro do corpo humano, como por exemplo, sensores de nível de glicose ou atuadores para injeção de insulina.

A Tabela 4.1 fornece uma lista de várias tecnologias de sensoriamento, junto com seus dados medidos correspondentes, que podem ser integradas como parte de um sistema vestível de monitoramento de saúde. A interface de um sensor pode ser uma entrada local e/ou uma rede de comunicação. Para entrada local, sensores multitoque flexíveis foram desenvolvidos e exibidos através de uma variedade de tecnologias, desde dispositivos OLED (*Organic Light Emitting Diodes*) até *displays* eletrônicos. A Figura 4.1 representa alguns desses sensores no corpo humano [Latré et al. 2011] e a Figura 4.2 mostra um sensor de pele flexível e elástico desenvolvido pela Universidade de Illinois [Xu et al. 2014].

Tabela 4.1. Sensores biológicos

Bio-sinal	Tipo do sensor	Dado medido
Eletrocardiograma (ECG)	Eletrodos de pele / tórax	Atividade do coração.
Pressão sanguínea	Monitor baseado em uma bolsa de ar posicionada no braço	Refere-se à força exercida pela circulação de sangue nas paredes dos vasos sanguíneos.
Temperatura do corpo	Sensor de temperatura	Medida da capacidade do corpo de liberar calor.
Taxa de respiração	Sensor piezoelétrico	Número de movimentos indicativos de inspiração e expiração por unidade de tempo.
Saturação de oxigênio	Oxímetro de pulso	Indica a oxigenação do sangue do paciente.
Frequência cardíaca	Resposta elétrica da pele	Frequência do ciclo cardíaco.
Transpiração	Resposta Galvânica da Pele	A condutividade elétrica da pele está associada à atividade das glândulas sudoríparas.
Sons cardíacos	Fonocardiografia	Registro de sons cardíacos.
Glicose no sangue	Medidores de glicose à base de tira	Medição da quantidade de glicose no sangue.
Eletromiograma (EMG)	Eletrodos colocados sobre o músculo	Atividade elétrica dos músculos esqueléticos.
Eletroencefalograma (EEG)	Eletrodos colocados no couro cabeludo	Medição da atividade elétrica cerebral involuntária e outros potenciais cerebrais.
Movimentos Corporais	Acelerômetro	Medição de forças de aceleração no espaço 3D.

4.2.2. Taxonomia e Requisitos

A comunicação das WBANs pode ser classificada em três classes diferentes da seguinte maneira:

- *Classe 1* - Comunicação intra-WBAN: a comunicação entre os dispositivos que compõem a WBAN.
- *Classe 2* - Comunicação entre WBANs: a comunicação entre dispositivos de diferentes WBANs através de seus respectivos dispositivos pessoais.
- *Classe 3* - Comunicação além-WBAN: a comunicação entre o dispositivo pessoal e redes externas (e.g. WiFi, celular.).

A Figura 4.3 mostra essas classes de comunicação em um sistema baseado em componentes. Os dispositivos estão espalhados por todo o corpo humano em uma visualização de topologia de rede centralizada, em que a localização exata de um dispositivo é específica do aplicativo [Latré et al. 2011, Movassaghi et al. 2014]. Os próximos parágrafos concentram-se na Classe 1, que inclui sensores, atuadores e outros dispositivos. Tecnologias aplicadas às Classes 2 e 3 são citadas.

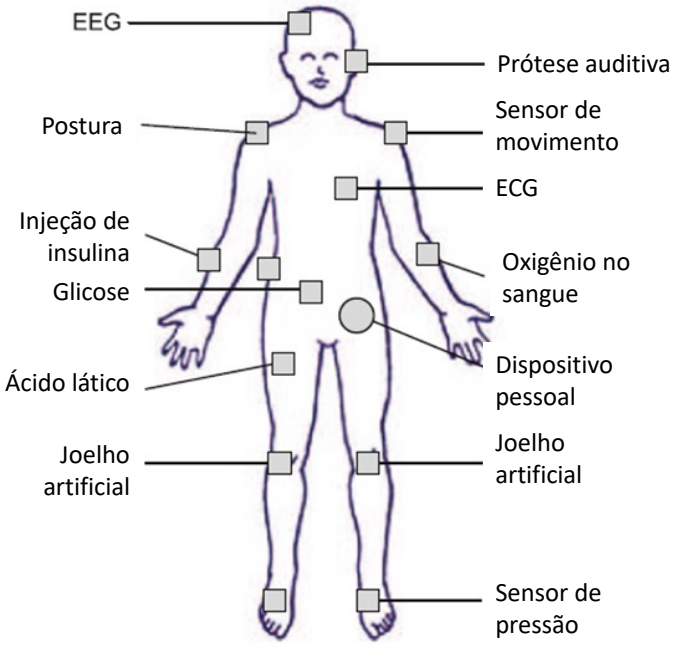


Figura 4.1. Sensores posicionados no corpo humano [Latré et al. 2011].

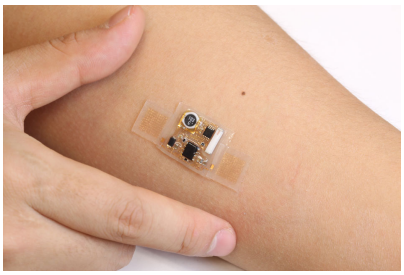


Figura 4.2. Sensor médico desenvolvido pela Universidade de Illinois [Xu et al. 2014]

A maioria das WBANs na literatura utiliza técnicas baseadas em RF (radiofrequência), e são classificadas de acordo com a faixa de operação [Cavallari et al. 2014]. A maioria das tecnologias WBAN opera na banda ISM. Portanto, para evitar problemas de interferência causados pela coexistência com outras redes sem fio, a FCC (*Federal Communications Commission*) forneceu banda de 40 MHz alocada em uma faixa de 2,36 GHz a 2,4 GHz exclusivamente para aplicações médicas, denominada MBAN (*Medical Body Area Network*) [Latré et al. 2011].

Existem outras tecnologias de RFs alocadas para aplicativos WBAN. MICS (*Medical Implant Communications Service*) ou Serviço de Comunicações de Implantes Médicos [Savci et al. 2005] é usado por dispositivos implantáveis e aplicativos que exigem uma taxa de bits acima de 1 Mbps. Pode-se citar, como exemplo, um fluxo de vídeo gerado por uma pequena câmera conectada a uma cápsula ingerível. WMTS (*Wireless Medical Telemetry System*) ou Sistema de Telemetria Médica Sem Fio [Yuce and Ho 2008] fornece uma taxa de bits de cerca de 400 kbps e cobre uma distância máxima de 2 metros.

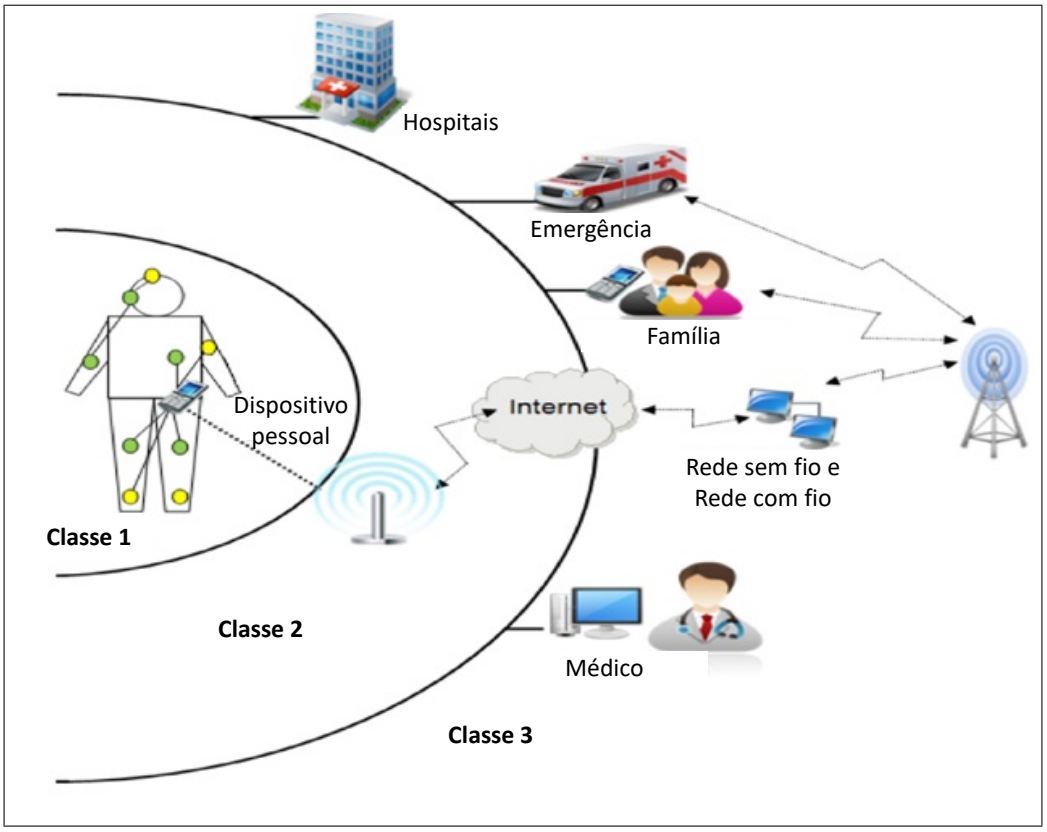


Figura 4.3. Arquitetura WBAN [Movassaghi et al. 2014]

Esta banda alocada é usada pelos sinais gerados por desfibriladores implantados e por neuroestimuladores.

UWB (*Ultra-Wide Band*) é descrito por IEEE 802.15.3 [Rhee et al. 2004], um padrão para WPANs (*Wireless Personal Area Networks*) de alta taxa de dados, que oferece uma faixa de 500 MHz e restrições de densidade espectral de até 41,25 dBm / MHz. Os receptores convencionais possui uma sensibilidade 30 dB maior que os receptores UWB. Isso representa uma proteção adicional em relação a interferência de outros dispositivos WBANs convencionais, já que os receptores UWB não teriam sensibilidade suficiente para reconhecer os sinais emitidos por outros sensores. Além disso, outro recurso que torna a tecnologia UWB uma boa candidata à comunicação WBAN é baixa suscetibilidade a interferência de múltiplos caminhos [Allen et al. 2005].

Uma nova tecnologia, chamada HBC (*Human Body Communication*) [Seyedi et al. 2013], usa o corpo humano como meio de comunicação. A propagação do sinal através do corpo humano é possível devido ao seu acoplamento capacitivo, e o acoplamento galvânico quando uma corrente alternada atravessa o corpo humano [Bae et al. 2012]. A HBC consome menos energia quando comparado ao UWB e oferece uma taxa de bits de 10 Mbps. Além disso, a coexistência de redes HBC é facilmente alcançável, uma vez que o sinal é limitado ao corpo humano. Em HBC, o movimento molecular (também chamado de movimento browniano) afeta a modelagem de camadas física (PHY) e camada de acesso ao meio (MAC).

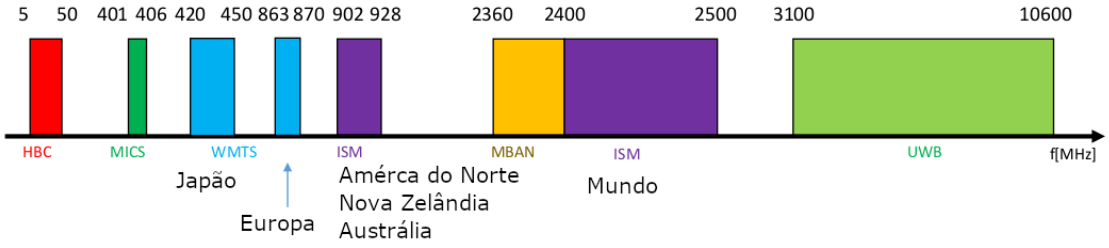


Figura 4.4. Faixa de frequências usadas pelas WBANs. Adaptado de [Cavallari et al. 2014]

Tabela 4.2. Requisitos de taxa de dados para aplicações WBANs

Aplicação	Taxa de dados	Atraso	Taxa de erro de bits (BER)
Monitor de Nível de Glicose	$< 1\ kbps$	$< 250\ ms$	$< 10^{-10}$
Voz	$50 - 100\ kbps$	$< 100\ ms$	$< 10^{-3}$
EEG	$86,4\ kbps$	$< 250\ ms$	$< 10^{-10}$
ECG	$192\ kbps$	$< 250\ ms$	$< 10^{-10}$
Estimulação cerebral profunda	$< 320\ kbps$	$< 250\ ms$	$< 10^{-10}$
Endoscopia através da cápsula	$1\ Mbps$	$< 250\ ms$	$< 10^{-10}$
Fluxo de áudio	$1\ Mbps$	$< 20\ ms$	$< 10^{-5}$
Fluxo de vídeo	$< 10\ Mbps$	$< 100\ ms$	$< 10^{-3}$

Em relação aos dispositivos implantáveis, a comunicação ultrassônica pode representar uma abordagem interessante. Ondas ultrassônicas têm menor nível de atenuação do que tecnologias baseadas em RF, principalmente dentro do corpo humano. O corpo humano é constituído predominantemente por água, que oferece menor resistência à propagação de ondas ultrassônicas [Galluccio et al. 2012].

4.2.3. Requisitos de Comunicação

A diversidade de aplicativos impõe vários requisitos para o desenvolvimento de WBANs. Os principais requisitos são recomendados pelo IEEE TG6 [Zhen et al. 2008], como mostrado a seguir.

1. *Taxa de dados e Qualidade de Serviço (QoS)*: a taxa de bits WBAN pode variar de 1 kbps (por exemplo, monitoramento da temperatura corporal) a 10 Mbps (por exemplo, streaming de vídeo). Uma WBAN pode conter um único ou vários links, sempre que mais de um dispositivo envia dados para um dispositivo pessoal, simultaneamente. No caso de vários dispositivos, mecanismos para Qualidade de Serviço (QoS) devem garantir a priorização do tráfego. As camadas física e de acesso ao meio oferecem métodos para correção de erros e prevenção de interferência, a fim de reduzir o BER (*Bit Error Rate*). Outros parâmetros de transmissão importantes são a latência e o *jitter*. Também é esperado que a WBAN forneça uma resposta rápida e confiável em situações de emergência. A Tabela 4.2 lista alguns requisitos importantes em termos de taxa de bits e QoS para WBANs.

2. *Alcance:* a distância entre os dispositivos WBAN não deve exceder seis metros. Além disso, a topologia em estrela é a configuração de topologia mais típica para as WBANs. O corpo humano representa um obstáculo natural à propagação de RF, principalmente quando se refere a sensores implantados. Uma rede com múltiplos saltos sem fio (*multi-hop*) poderia atenuar esse problema. Neste tipo de rede, os dados são transmitidos de um nó sensor para um dispositivo pessoal através de seus vizinhos mais próximos. Cada WBAN contém vários sensores que podem variar por exemplo de 2 a 10 nós. Uma WBAN geralmente implementa um mecanismo confiável de associação, permitindo que um novo nó seja agregado ou separado de outros em um grupo de nós preexistente, conforme exigido pelo usuário.

3. *Segurança:* a segurança de dados é um requisito primário para WBANs em aplicações médicas e militares. A segurança é tratada em termos de privacidade, confidencialidade, métodos de autorização e integridade de dados. No entanto, os mecanismos atuais de criptografia de dados não são apropriados para WBANs, uma vez que seus nós possuem recursos limitados de processamento, memória e consumo de energia. Existem alguns mecanismos que podem melhorar a segurança da WBAN, como a identificação biométrica.

4. *Antena e canal de rádio:* minúsculas dimensões dos nós WBAN levam a um projeto de antena altamente eficiente. Além disso, o corpo humano afeta o perfil da radiação da antena. Uma caracterização apropriada do canal de rádio é importante para projetar antenas com propriedades de radiação adequadas.

5. *Consumo de energia:* dispositivos WBAN geralmente têm energia da bateria e seu consumo de energia varia de acordo com a aplicação. Espera-se que a vida da bateria seja a de maior tempo possível, especialmente para dispositivos implantados. Isso é obtido usando transceptores de RF de baixa potência e colocando dispositivos WBAN no modo de espera sempre que não estiverem sendo usados. Existem também estudos que consideram o calor [Hoang et al. 2009] e movimento corporal [Von Buren et al. 2006] como fonte de suprimento de energia.

6. *Coexistência:* como mencionado anteriormente, a maioria dos WBANs é projetada para operar na banda ISM. Esta faixa de frequência é usada pela maioria dos dispositivos sem fio. Os principais padrões de transmissão sem fio que operam na banda de frequência ISM são: Wi-Fi (IEEE 802.11) [Bianchi 2000], *Bluetooth* (IEEE 802.15.1) e IEEE 802.15.4/ZigBee [Kirti 2016]. Quando um número de redes sem fio opera na mesma faixa de frequência, a interferência entre elas aumenta a perda de pacotes e diminui o desempenho de WBANs. Isso é um problema especialmente quando ocorre durante eventos de risco à saúde, como por exemplo, um ataque cardíaco. Algumas soluções são tratadas na camada física para mitigar efeitos prejudiciais [Hayajneh et al. 2014]. Existem pesquisas que consideram atividades de outras redes para reforçar os sinais WBAN, ou mesmo retransmitir seus dados de sensores [Heaney et al. 2011].

7. *Projeto do Hardware:* existem restrições significativas de tamanho para dispositivos WBAN. Os aspectos mais críticos são como incorporar a antena e a bateria em um dispositivo minúsculo, oferecendo boa radiação de antena e vida útil da bateria.

Alguns nós WBAN são projetados para serem acoplados a uma peça de roupa, flexível e confortável para o usuário, especialmente durante atividades esportivas ou campanhas militares.

8. *Processamento de sinais:* o consumo de energia pode ser o requisito mais restritivo para dispositivos WBAN. No entanto, técnicas de processamento de sinal mais eficientes podem melhorar o controle sobre o consumo de energia relacionado à aquisição e análise de sinais biológicos. Uma técnica chamada *Compressed Sensing* (CS) permite a amostragem de sinais a uma taxa sub-Nyquist de economia de energia, sem perda de informação [Donoho 2006]. Isso é usado em muitos cenários de WBANs, como EEG (eletroencefalograma), ECG (eletrocardiograma) e EMG (eletromiograma) [Dixon et al. 2012].

4.2.4. Segurança para o Corpo Humano

Radiação é o processo pelo qual a energia é emitida como partículas ou ondas. A radiação eletromagnética inclui desde ondas de rádio a ondas gama. A radiação pode ser classificada como ionizante ou não ionizante, com base em se tem energia suficiente para eliminar os elétrons dos átomos com os quais interage, além de ser capaz de causar danos de menor energia, como quebrar as ligações químicas nas moléculas.

A radiação ionizante, que tem uma frequência mais alta do que a radiação não ionizante, representa uma ameaça à saúde humana. Dependendo dos níveis de exposição, a radiação ionizante pode causar queimaduras, câncer e danos genéticos. Na região do espectro acima de 10^{16}Hz (ultravioleta), a radiação pode ser tratada como ionizante, embora a faixa de frequências não seja muito bem definida.

A maioria das radiações não ionizantes, como a energia de rádio e microondas, é considerada prejudicial à saúde humana para uma determinada quantidade de energia térmica. A energia das partículas de radiação não ionizante é baixa, e ao invés de produzir íons alterados ao passar pela matéria, a radiação não ionizante tem energia suficiente para alterar a configuração de valência vibracional ou eletrônica de moléculas e átomos, produzindo efeitos térmicos. Possíveis efeitos não térmicos de radiação não ionizante em tecidos vivos foram estudados apenas recentemente. Uma dificuldade é que há controvérsias de que as frequências superiores de radiação não ionizante (radiação de microondas e radiação de rádio) são capazes de fato de causar danos biológicos não térmicos. A Agência Internacional de Pesquisa sobre o Câncer sugere que pode haver algum risco de radiação não ionizante para os seres humanos [Baan et al. 2011].

Organizações governamentais e não-governamentais declararam alguns limites à exposição relacionada à radiofrequência ou campos eletromagnéticos de uma forma geral. A ICNIRP (Comissão Internacional de Proteção contra Radiação Não-Ionizante) recomenda algumas restrições de tempo de exposição do corpo humano a campos eletromagnéticos não ionizantes. Tais restrições são tipicamente definidas em termos da Taxa de Absorção Específica (SAR - *Specific Absorption Rate*). A SAR é definida como a taxa de absorção de energia eletromagnética por tecido corporal específico, medida por 6 minutos, e sua unidade é de W/kg. Considerando todo o corpo humano, a SAR estimada pelos dispositivos WBAN é insignificante. No entanto, deve ser dada alguma atenção à SAR local, em outros termos, a SAR medida em uma parte específica exposta do corpo

Tabela 4.3. Limites básicos para a exposição do corpo humano aos campos eletromagnéticos estabelecidos pela ANATEL

Categoria de exposição	SAR do corpo inteiro	SAR local (cabeça e tronco)	SAR local (braços)
Ocupacional	0.4	10	20
Público geral	0.08	2	4

humano aos campos eletromagnéticos. A SAR local depende da condutividade elétrica e densidade do tecido sob exposição. Portanto, os dispositivos WBAN devem minimizar a SAR local e obedecer a regulamentos internacionais e padrões regionais [ICNIRP 2009].

Outras organizações preocupadas em estabelecer limites para a exposição aos campos eletromagnéticos ao tecido corporal são o NCRP (Conselho Nacional de Proteções de Radiação), ANSI e IEEE. No Brasil, a ANATEL (Agência Nacional de Telecomunicações) estabeleceu alguns limites relacionados aos efeitos térmicos causados pela exposição desses campos. Os limites são mostrados na Tabela4.3.

A transmissão de dados de nós sensores sem fio, implantados ou sobre a pele, pode aumentar a temperatura do tecido local e causar efeitos indesejáveis no corpo humano, devido a transmissões mais longas [Movassaghi et al. 2013a]. Os principais efeitos térmicos no corpo humano são:

- Redução do fluxo sanguíneo local.
- Danos térmicos aos órgãos mais sensíveis.
- Alguns tipos de crescimento bacteriano.
- Efeitos sobre os relacionamentos enzimáticos.

A OMS (Organização Mundial da Saúde) estabelece os limites de exposição definidos pela ICNIRP, que são seguidos pela ANATEL. No entanto, a OMS, assim como a ANATEL, consideraram apenas efeitos térmicos. Os efeitos não térmicos do EMERF, prejudiciais à saúde humana, ainda estão em fase de pesquisa. Assim, os riscos biológicos devidos à exposição prolongada ao EMERF devem ser melhor controlados e compreendidos, para que os benefícios dos sensores sem fio no WBAN para a saúde sejam cada vez mais evidentes.

4.2.5. Casos de Uso Relacionados a WBANs

Esta seção descreve três casos de uso relacionados ao uso de WBANs para assistência médica. Esses casos de uso são baseados em padrões de protocolo de transmissão abertos e de código aberto. A Tabela 4.4 lista alguns protocolos de comunicação utilizados nessas aplicações [Al-Fuqaha et al. 2015, Latré et al. 2011, Movassaghi et al. 2014].

1. *Sistema de monitoramento do paciente em casa:* este caso de uso tem como objetivo coletar e medir os sinais fisiológicos de pacientes, entregando-os a um ou vários postos de enfermagem, e identificando situações de risco de vida, considerando o

Tabela 4.4. Alguns protocolos de comunicação usados em WBANs

Protocolo	Tipo	Descrição
IPv6	Endereçamento	IPv6 é a versão mais recente do protocolo da Internet usado pela maioria das WBANs.
RPL	Roteamento	RPL significa *Routing Protocol for Low Power and Lossy Network*. Este protocolo de roteamento é documentado na RFC6550 para baixa capacidade de processamento e dispositivos com restrição de consumo de energia.
mDNS	Serviço de tradução	mDNS significa *Multicast Domain Naming Service*. É um protocolo de aplicação que traduz uma URL em um endereço IP correspondente, conforme documentado na RFC 6762.
HTTP REST	Protocolo de aplicação	REST significa *Representational Transfer of State*. É um protocolo de transferência de mensagens entre elementos da web. Cada um dos elementos da web é identificado por uma URI (*Unified Resource Identification*). Este protocolo estabelece um conjunto básico de operações de dados, como POST, GET, PUT, DELETE e usa uma arquitetura cliente / servidor tradicional.

contexto clínico do paciente. Nesse caso, os pacientes são monitorados na área da saúde, na residência da própria pessoa, em vez da internação. Os sinais fisiológicos do paciente são coletados por sensores e enviados via Wi-Fi ou outro padrão de comunicação sem fio, como o IEEE 802.15.4. O MQTT (*Messaging Queuing Telemetry Transport*) [Hunkeler et al. 2008] é um protocolo de mensagens para dispositivos de baixo processamento otimizados para redes TCP / IP não confiáveis e de alta latência. O MQTT foi inicialmente proposto pela IBM para sistemas de supervisão e aquisição de dados em redes elétricas, também conhecido como SCADA (*Supervision Control and Data Acquisition*). A troca de mensagens usa o padrão de mensagem de publicação / assinatura (*publish-subscribe*), em que os remetentes de mensagens, chamados publicadores (*publishers*), não enviam mensagens diretamente para destinatários específicos, chamados assinantes (*subscribers*), mas são publicadas em classes hospedadas em um servidor, chamado *broker*. Da mesma forma, os assinantes que manifestarem interesse em uma ou mais classes podem enviar uma mensagem de assinatura para o intermediário. A Figura 4.5 mostra o caso de uso. O MQTT está disponível em formato de código aberto e implementado em vários softwares gratuitos [Torres et al. 2016]. Assim, um aplicativo calcula e consolida como informações as atividades vitais de pacientes monitorados em um painel (*dashboard*). Esse painel pode ser exibido em postos de enfermagem ou em aplicações médicas.

2. *Monitorização e mitigação de perturbações relacionadas com a alimentação:* esta aplicação visa ajudar pacientes com problemas motores (por exemplo, doença de Parkinson) na atividade alimentar diária. O objetivo é compensar os tremores das mãos usando MEMS (*MicroElectroMechanical Systems*). O paciente usa luvas equipadas com acelerômetros e pequenos MEMS, que reagem aos movimentos da mão. A comunicação entre acelerômetros e atuadores MEMS deve ser com o me-

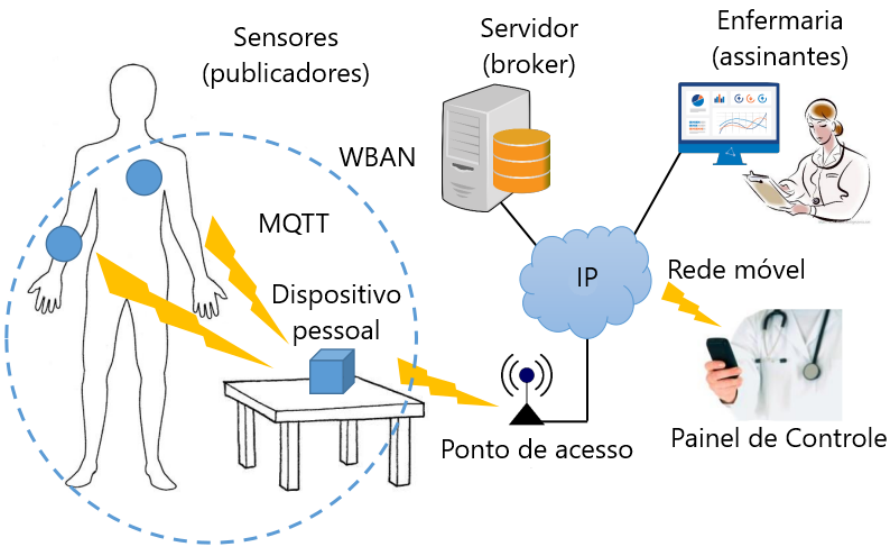


Figura 4.5. Estudo de caso do WBAN: sistema de monitoramento remoto do paciente

nor atraso possível. Assim, o protocolo usado é o DDS [David et al. 2013], que fornece uma ligação direta entre acelerômetros e MEMS. Também é possível transferir alguns dados para um corretor através do protocolo MQTT com o objetivo de compartilhar informações sobre as atividades do paciente, a fim de apoiar alguma análise clínica subsequente.

3. *Sistema de navegação para pessoas com deficiência visual:* pode ser considerado como uma extensão do caso de uso mencionado em (1). Alguns transceptores (por exemplo, DecaWave ou Nanotron) fornecem serviços de localização geográfica em tempo real, também conhecidos como RTLS (*Real-Time Locating Services*). É possível usar o serviço mDNS [Cheshire and Krochmal 2013] para acessar um serviço RTLS local. Então, pode-se mapear as posições dos quartos e móveis em casa. Uma pessoa com deficiência visual poderia ser guiada neste local, evitando obstáculos.

4.2.6. Métricas de Avaliação para WBANs

O desempenho das WBANs está estritamente relacionado com as soluções para enfrentar os desafios presentes nas WBANs. Na literatura para avaliar as diferentes soluções propostas para WBAN, entre elas os protocolos de roteamento, utilizam-se diferentes métricas, as quais são listadas a seguir.

1. *Vida útil da rede.* Define o tempo de operação total da rede até o último nó ficar inativo.

2. *Energia residual.* É a diferença entre a energia inicial e a energia utilizada durante a operação da rede.

3. *Número de nós ativos e inativos.* É o número de nós que depois de um tempo, predefinido, de operação da rede, ainda têm energia ou consumiram toda sua energia respectivamente.

4. *Período de estabilidade.* É o tempo antes de o primeiro nó ficar inativo.

5. *Perda de caminho.* É a diferença entre a potência do sinal transmitido pelo nó transmissor e a potência recebida no nó receptor.

6. *Atraso fim-a-fim (latência).* É o tempo médio tomado por um pacote de dados para alcançar o nó concentrador da WBAN desde o nó origem.

7. *Taxa de entrega de pacotes.* Determinada pelo número de pacotes recebidos no nó concentrador, divididos pelo número de pacotes enviados desde o nó origem.

8. *Taxa de perda de pacotes.* Determinada pelo número de pacotes perdidos na transmissão, é o número de pacotes perdidos divididos pelo número de pacotes enviados desde o nó origem.

9. *Aumento da temperatura.* Geralmente é uma métrica utilizada para avaliar protocolos que têm como principal objetivo evitar o aquecimento. É uma medida de quanto um nó esquenta durante sua operação.

10. *Vazão (throughput).* É definida como a taxa real de transmissão de dados através de uma rede. Normalmente é medida em bits por segundo e será sempre menor do que a taxa de transmissão nominal da interface de rede (*bit rate*).

11. *Pacotes reenviados*. É o número de pacotes que são reenviados como causa de um descarte anterior por alguma causa como por exemplo congestionamento.

Das métricas supracitadas, as quatros primeiras estão relacionadas à eficiência energética. As métricas 6, 7, 8 estão associadas à qualidade de serviço (QoS). A métrica 9 está associada à absorção de radiação e aquecimento dos nós. A métrica 10 oferece uma ideia geral de desempenho da rede, pois vários aspectos como: topologia de rede, particionamento topológico, perdas do caminho e limitação de recursos influenciam em seu desempenho.

Considerando a relação entre as diferentes métricas e os desafios impostos por WBANs, conclui-se que as métricas de desempenho mais importantes para avaliar a implementação de WBANs são: vida útil da rede, energia residual, latência, taxa de entrega de pacotes e vazão. Obter melhores valores dessas métricas são objetivos comuns de diversos trabalhos encontrados na literatura, independentemente de suas diferenças operacionais.

4.3. Comunicação em WBANs

A comunicação interna ou nas proximidades do corpo humano é um desafio para o projeto de protocolos para WBANs que sejam adaptáveis, dinâmicos e flexíveis. Portanto, baixo atraso, alta confiabilidade, baixo consumo de energia, baixa interferência eletromagnética no corpo humano e comunicação efetiva são extremamente importantes em WBANs [Bhandari and Moh 2016].

Os objetivos mais comuns em uma WBAN, usualmente solucionados por um protocolo da camada MAC, são atingir o máximo de vazão, o mínimo de atraso e maximizar

o tempo de vida da rede, controlando as principais fontes de desperdício de energia como a colisão de quadros, através da escuta inativa do canal (*overhearing*) e redução da sobrecarga de controle (*overhead*) [Ullah et al. 2009].

Vários protocolos MAC foram estudados para fins específicos, mas foram adotados com algumas modificações para atender aos requisitos inerentes das WBANs. Esta seção apresenta uma visão geral de alguns dos principais protocolos propostos para o cenário das WBANs.

4.3.1. IEEE 802.15.6

O padrão IEEE 802.15.6 [IEEE Std 802.15.6 2012] foi proposto para comunicação sem fio de curto alcance na vizinhança ou mesmo dentro do corpo humano (mas não limitado a humanos). Esta norma utiliza as bandas de radiofrequência científica, industrial e médica (*Industrial, Scientific and Medical* - ISM) existentes, bem como faixas de frequências aprovadas por associações médicas e autoridades reguladoras locais.

Os requisitos padrão incluem suporte para qualidade de serviço (QoS), baixa potência de transmissão, taxas de transmissão de dados de até 10 Mbps e conformidade com as diretrizes de não interferência.

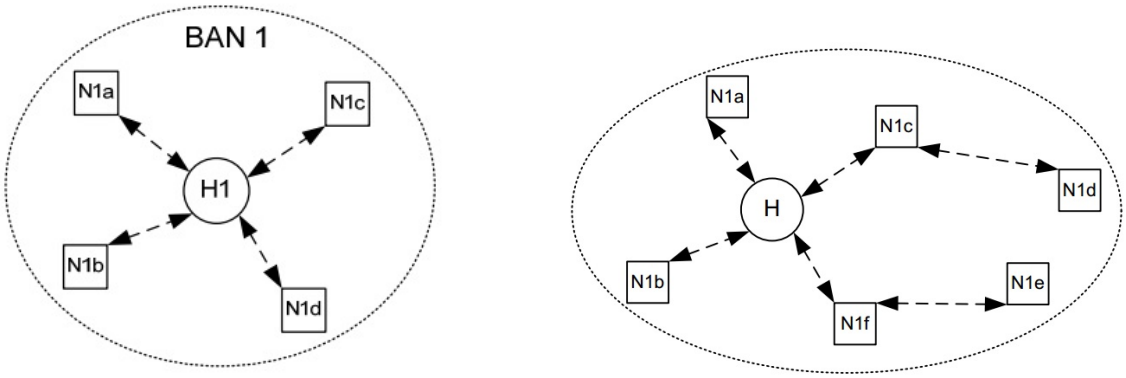
A fim de minimizar a Taxa de Absorção Específica (SAR) do corpo e considerar adequadamente as mudanças nas características do canal de comunicação devido aos movimentos do usuário, as antenas portáteis e os padrões de radiação são modelados para considerar cada tipo de corpo humano (homem, mulher, magro, obeso, etc).

A rede é composta de nós e concentradores (*hubs*), que são organizados em conjuntos lógicos, referidos como BANs (*Body Area Networks*). Os nós são coordenados por seus respectivos *hubs* para acesso ao meio e gerenciamento de energia. Haverá um e apenas um *hub* em uma BAN e vários nós, compondo uma topologia em estrela de um salto, como na Figura 4.6a.

Opcionalmente, um nó com funcionalidade de retransmissão pode ser usado para estender a topologia de estrela para uma topologia de dois saltos, como na Figura 4.6b. As trocas de quadros devem ocorrer diretamente entre os nós e o *hub* da BAN ou, opcionalmente, através do nó com capacidade de retransmissão.

Todos os nós e *hubs* seguem o modelo de referência IEEE 802, tendo uma camada física (PHY) e em uma subcamada de controle de acesso ao meio (MAC). O modelo de referência dentro de um nó ou um hub é mostrado na Figura 4.7. No modelo de referência, a subcamada MAC fornece seu serviço ao cliente MAC (camada superior) por meio do ponto de acesso de serviço (*Service Access Point* - SAP) para a camada MAC, denominado MAC SAP. Enquanto isso, a camada PHY fornece seu serviço para a subcamada MAC através do PHY SAP. O cliente MAC passa unidades de dados de serviços MAC (MSDUs) para a subcamada MAC via MAC SAP, e a subcamada MAC passa quadros MAC, como unidades de dados de protocolo MAC ou MPDUs, para a camada PHY via PHY SAP.

A entidade de gerenciamento de nó (*Node Management Entity* - NME) e a entidade de gerenciamento de *hub* (Hub Management Entity - HME) são interfaces de informações de gerenciamento de rede lógicas para trocar informações entre camadas. Elas não são



(a) Topologia em estrela de um salto (b) Topologia em estrela de dois saltos

Figura 4.6. Topologia das WBANs [IEEE Std 802.15.6 2012]

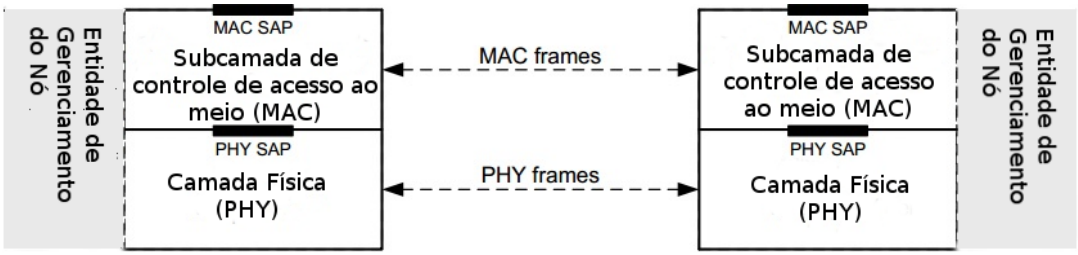


Figura 4.7. Modelo de Referência IEEE 802.15.6 [IEEE Std 802.15.6 2012]

obrigatórias, nem seu comportamento é especificado pelo padrão.

4.3.1.1. Camada Física (PHY)

A camada PHY é responsável pelas seguintes tarefas: ativação e desativação do transceptor de rádio, verificação de uso do canal sem fio (*Clear Channel Assessment* - CCA), transmissão de dados e recepção.

A camada PHY fornece um procedimento para transformar uma unidade de dados de serviço da camada física (PSDU) em uma unidade de dados de protocolo da camada física (PPDU). O IEEE 802.15.6 especificou três diferentes camadas físicas: *Narrow Band* (NB), *Ultra-Wide Band* (UWB) e *Human Body Communication* (HBC).

Na PHY NB, o PSDU é pré-anexado com um preâmbulo de camada física e um cabeçalho de camada física para criar a PPDU, como na Figura 4.8. Esses componentes são:

- Preâmbulo da camada física: o protocolo de convergência da camada física (PLCP) é usado para auxiliar o receptor durante a sincronização para recepção e a recuperação do deslocamento da portadora.
- Cabeçalho da camada física: o cabeçalho PLCP transmite as informações necessárias sobre os parâmetros PHY para auxiliar na decodificação da PSDU no recep-

tor decomposta em um campo RATE, um campo LENGTH, um campo BURST MODE, um campo SCRAMBLER SEED, bits reservados, uma sequência de verificação de cabeçalho (HCS) e bits de paridade BCH. O cabeçalho PLCP deve ser transmitido usando a taxa de dados de cabeçalho especificada na faixa de frequência em operação.

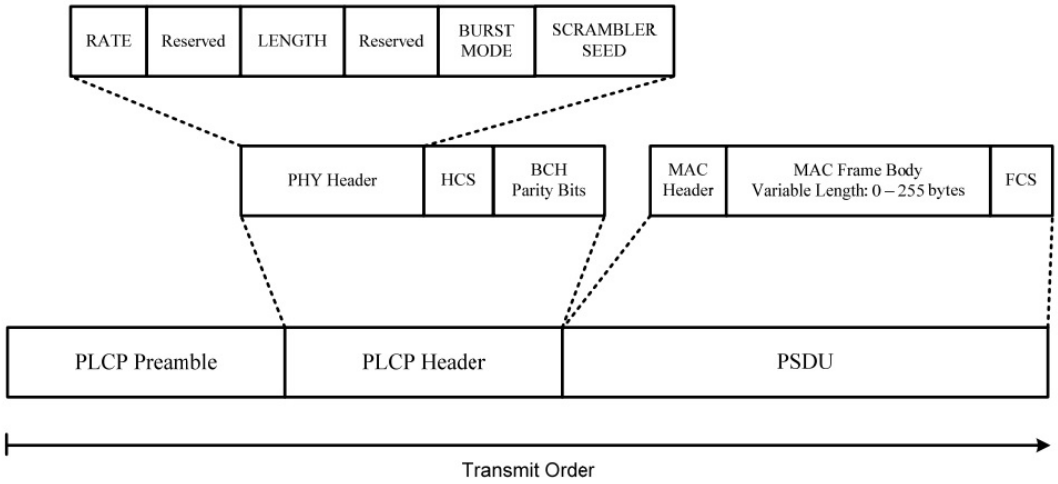


Figura 4.8. Estrutura da unidade de dados da camada física (PPDU) [IEEE Std 802.15.6 2012]

Um dispositivo compatível com PHY NB deve ser capaz de suportar transmissão e recepção em pelo menos uma das seguintes bandas de frequências: 402 MHz a 405 MHz, 420 MHz a 450 MHz, 863 MHz a 870 MHz, 902 MHz a 928 MHz, 950 MHz a 958 MHz, 2360 MHz a 2400 MHz e 2400 MHz a 2483,5 MHz.

A PHY NB usa as técnicas de modulação diferencial por deslocamento de fase DBPSK, DQPSK e D8PSK, exceto em 420-450 MHz onde usa GMSK (*Gaussian Minimum Shift Keying*). A taxa de dados da informação pode ser de até 971.4 kbps na PHY NB.

A especificação PHY de banda ultralarga (UWB) foi projetada para oferecer um desempenho robusto para as BANs e fornecer um grande escopo para oportunidades de implementação de alto desempenho, robustez, baixa complexidade e operação em baixa potência. O interesse em UWB reside no fato de que os níveis de potência do sinal estão na ordem daqueles usados na banda do Serviço de Comunicação de Implante Médico (*Medical Implant Communication Service* - MICS), portanto, fornece níveis seguros de energia para o corpo humano e baixa interferência em outros dispositivos.

Existem dois tipos diferentes de tecnologias UWB utilizadas pelo padrão: por impulsos de rádio (IR-UWB) e modulação de frequência (FM-UWB). Em ambas as tecnologias, o PLCP constrói a unidade de dados de protocolo de camada PHY (PPDU) concatenando o cabeçalho de sincronização (SHR), cabeçalho de camada física (PHR) e unidade de dados de serviço de camada física (PSDU), respectivamente. Além disso, os bits PPDU são convertidos em sinais de RF para transmissão no meio sem fio.

O campo PHR contém informações sobre a taxa de dados do PSDU, comprimento

do quadro MAC, formato do pulso, modo burst, HARQ e a semente do embaralhamento. O cabeçalho de sincronização (SHR) deve ser dividido em duas partes. A primeira parte é o preâmbulo, destinado à sincronização de temporização, detecção de pacotes e recuperação de deslocamento de frequência da portadora. A segunda parte é o delimitador de início de quadro (SFD) para sincronização de quadros. O UWB pode operar em dois grupos de bandas de frequência, banda baixa e banda alta, existem várias frequências opcionais e as seguintes frequências obrigatórias: 3993,6 MHz em banda baixa e 7987,2 MHz em banda alta.

A camada física de comunicação no corpo humano (PHY HBC) usa a tecnologia de comunicação de campo elétrico (EFC). Espera-se que ela tenha uma ampla gama de aplicações, como controle de entrada de salas, segurança de escritório, assistência médica, logística, serviços pessoais avançados e entretenimento. No campo de segurança de escritório, por exemplo, um usuário poderia especificar o documento que deseja imprimir apenas tocando nele.

O pacote HBC é composto por preâmbulo PLCP, delimitador de início de quadro (SFD), cabeçalho PLCP e carga PSH (PSDU). No PLCP, uma sequência de preâmbulo é transmitida quatro vezes para conseguir a sincronização de pacotes. O campo SFD / RI é usado como delimitador de início de quadro (SFD) para o pacote não-*burst* ou é usado como um indicador de taxa (RI) para o pacote *burst* (rajada). O cabeçalho do PLCP contém informações sobre taxa de dados, informações do sinal piloto, um sinalizador do modo rajada, comprimento do quadro MAC e um campo CRC8. Um dispositivo compatível deve ser capaz de suportar transmissão e recepção na faixa de 21 MHz.

4.3.1.2. Camada MAC

Entre as principais razões para o desperdício de energia em redes sem fio estão: (1) colisão de quadros, que ocorre quando mais de um quadro é transmitido ao mesmo tempo, ocorrendo assim perdas de quadros por colisão, e aumentando o consumo de energia com as retransmissões desses quadros perdidos; (2) *idle listening* (escuta ociosa), que ocorre quando um nó escuta um canal ocioso para receber dados; (3) *over hearing*, que ocorre quando um nó escuta o canal para receber quadros que são destinados a outros nós; (4) *packet overhead*, refere-se a transmissão de pacotes e informações de controle adicionadas aos cabeçalhos. O número de pacotes de controle usados para realizar o processo de comunicação de dados também influencia o consumo de energia [Sruthi 2016].

Todos esses aspectos estão estritamente relacionados com as funções da camada MAC, que incluem o controle de acesso ao canal, agendamento da transmissão, empacotamento dos dados e delimitação de quadros (*data framing*), manipulação de erros e gerenciamento de energia. Portanto, nesta camada, um protocolo com um mecanismo de acesso ao meio eficiente é de muita importância para ajudar na eficiência energética.

Os protocolos MAC geralmente usam o TDMA (*Time Division Multiple Access*) ou CSMA/CA (*Carrier Sense Multiple Access with Collision Avoidance*) para o acesso justo ao meio compartilhado. Outras soluções como FDMA (*Frequency Division Multiple Access*) e CDMA (*Code Division Multiple Access*) não são adequados para WBAN, devido à complexidade de hardware e alto poder computacional que eles precisam. Porém

CSMA/CA foi projetado para redes dinâmicas e presume-se que as WBANs não são tão dinâmicas, além disso tem um consumo adicional de energia associado a evitar colisões. Por outro lado os protocolos MAC baseados em TDMA requerem um consumo de energia extra para a sincronização [Javaid et al. 2013].

Com base nessas considerações, no padrão IEEE 802.15.6, foi definido um protocolo MAC específico para WBAN, projetado para operar dentro e ao redor do corpo humano. Este protocolo é apresentado a seguir.

Um quadro MAC, de acordo com o padrão IEEE 802.15.16 ilustrado na Figura 4.9, consiste em um cabeçalho MAC de tamanho fixo, um corpo de quadro MAC de comprimento variável e um campo FCS (*Frame Check Sequence*) de comprimento fixo. O cabeçalho MAC contém as informações de controle do quadro, como versão do protocolo, política de reconhecimento, nível de segurança, retransmissão, etc.. Além disso, o cabeçalho informa o endereço MAC do destinatário, endereço MAC do remetente e identificador da BAN. O corpo do quadro MAC possui dois campos opcionais para fins de segurança, o número de sequência de segurança de baixa ordem e o código de integridade de mensagem (MIC) e a carga útil. O campo FCS utiliza CRC de 16 bits para detecção de erro.

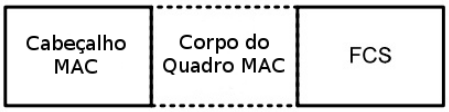


Figura 4.9. Formato do Quadro MAC [IEEE Std 802.15.6 2012]

Existem três modos diferentes de operação em relação a diferentes estratégias de modos de acesso: o modo *beacon* com períodos de *beacon* (superquadro), modo sem *beacons* com superquadros e modo sem *beacons* sem superquadro.

No modo *beacon* com superquadros (*superframes*), o *hub* deve organizar as fases de acesso aplicáveis em cada período ativo (superquadro). Essas fases de acesso são fase de acesso exclusivo (EAP), fase de acesso aleatório (RAP), fase de acesso gerenciado (MAP) e uma fase de acesso com contenção (CAP). Nos intervalos de alocação de EAP, RAP e CAP, um nó pode obter e iniciar transações de quadros, usando o acesso aleatório baseado nos protocolos *Aloha* ou CSMA/CA. No MAP, o *hub* deve organizar os intervalos de alocação e agendar as transmissões. Em um modo não-*beacon* com superquadros, há apenas um período de acesso MAP.

No modo sem *beacons* sem superquadros, um *hub* pode fornecer intervalos de alocação não programados. Um nó pode tratar qualquer intervalo de tempo como uma porção de EAP1 ou RAP1 e empregar acesso aleatório baseado em CSMA/CA para obter uma alocação com contenção.

Funções MAC como QoS e Gerenciamento de Energia são executadas da seguinte maneira:

- A QoS é executada usando diferentes prioridades de usuário (UPs) com limites diferentes de janela de contenção (CW) para o CSMA/CA e limiares de probabilidade de contenção (CP) para acesso *Slotted Aloha*.

- Um nó com pouca energia armazenada em bateria pode hibernar, isto é, estar em estado inativo ao longo dos seus períodos de *beacons* (superquadros). Um campo de Capacidade MAC é usado para este propósito e um nó pode configurar um contador de períodos de inatividade, ou seja, por quantos superquadros ele ficará inativo até que acorde. Quando acordar, deverá realizar suas transmissões e recepções e, ao término do superquadro, retorna a hibernar e reinicia o contador. Além disso, durante o período MAP dentro de um superquadro, o nó pode permanecer inativo, caso não tenha sido agendada nenhuma transmissão ou recepção para o mesmo.

4.3.2. Outros Protocolos MAC Propostos

Vários outros protocolos foram usados para cenários WBAN ou foram propostos para este fim. O padrão IEEE 802.15.4 [IEEE Std 802.15.4 2006b] tem sido um dos focos principais de muitas pesquisas durante os últimos anos. Algumas das principais razões para selecionar o IEEE 802.15.4 para uma WBAN são a comunicação de baixa potência e o suporte para aplicativos WBAN de baixa taxa de dados. Mas como o IEEE 802.15.4 funciona em uma banda não licenciada, problemas de conectividade em tempo real e corrupção de dados são inevitáveis. Além disso, não há mecanismo de QoS no protocolo, que é um recurso fundamental em um cenário WBAN.

Um outro protocolo proposto é o H-MAC [Li and Tan 2010]. É um protocolo MAC que usa o batimento cardíaco para realizar a sincronização entre os nós e reduzir custos extras de energia. Embora o H-MAC reduza o custo extra de energia necessário para sincronização, ele não suporta eventos esporádicos e o ritmo dos batimentos cardíacos depende da condição do paciente.

O protocolo PMAC [Ullah et al. 2014] é um protocolo baseado no padrão IEEE 802.15.6 e BodyMAC [Marinkovic et al. 2009a]. Ele usa dois períodos de acesso, um com contenção (CAP - *Contention-Access Period*) para acomodar tráfego normal e vital, como no IEEE 802.15.6, e um período livre de contenção (CFP - *Contention-Free Period*) para acomodar uma grande quantidade de quadros de dados, como no BodyMAC. No PMAC, o período CFP consiste em vários *slots* TDMA e é usado para uma grande quantidade de dados, incluindo dados de *streaming*.

Em [Sruthi 2016] e [Javaid et al. 2013], apresenta-se um estudo de protocolos MAC eficientes em termos de energia propostos para WBAN. Este estudo inclui, além do protocolo MAC definido no padrão WBAN, outros como: *Battery-Aware TDMA Protocol* [Su and Zhang 2009], *Priority-Guaranteed MAC Protocol* [Zhang and Dolmans 2009], *Energy-efficient Low Duty Cycle* (ELDC) [Marinkovic et al. 2009b], *Power-efficient MAC Protocol* [Al Ameen et al. 2011], *Energy-efficient Medium Access Protocol* (EMAP) [Omeni et al. 2008] e *Adaptive Energy-Efficient MAC* [Van Dam and Langendoen 2003], entre outros. Desses protocolos, os que oferecem melhor desempenho para WBAN são o definido no padrão IEEE 802.15.6 MAC e o protocolo *Adaptive Energy-Efficient MAC*, também conhecido como T-MAC (*Time-out MAC*).

A eficiência energética é o principal desafio encarado pelo protocolo T-MAC [Van Dam and Langendoen 2003], embora também atinja a escuta ociosa. Este usa ciclos de trabalho (*duty cycles*) flexíveis para aumentar a eficiência energética. Cada nó periodicamente acorda para se comunicar com seus vizinhos e depois de passado o pe-

ríodo de atividade, vai dormir novamente até o próximo quadro. Para reduzir a escuta ociosa, transmite todas as mensagens em rajadas de comprimento variável e dorme entre rajadas. Durante o período de inatividade do nó, as novas mensagens são colocadas numa fila, e são enviadas quando o nó acorda. Os nós se comunicam entre si usando o esquema *Request-To-Send* (RTS), *ClearTo-Send* (CTS) e reconhecimento de dados (ACK), evitando colisão e fornecendo transmissão confiável.

O protocolo T-MAC foi implementado na camada MAC do simulador Castalia [Boulis et al. 2011]. Este simulador foi desenvolvido especificamente para a simulação de WBANs. E a implementação de T-MAC em Castalia foi feita em 2010, quando ainda não se tinha disponível o padrão IEEE 802.15.6. Em [Tselishchev et al. 2010], os responsáveis por essa implementação expõem que T-MAC foi escolhido porque era um protocolo MAC popular e bem-sucedido, conforme trabalhos encontrados na literatura [Sruthi 2016] e [Javaid et al. 2013].

4.3.3. Protocolos de Roteamento

Dado que os aspectos relativos ao roteamento em WBAN não estão definidos no padrão IEEE 802.15.6, e também motivados pela busca de soluções para enfrentar os desafios de WBAN, a maioria dos trabalhos de pesquisa em WBANs nos últimos anos estão focados no desenvolvimento de novos protocolos de roteamento.

Na atualidade, existe um número considerável de protocolos desenhados para WBAN já desenvolvidos, e também foram feitos vários estudos focados somente nesses protocolos. Em [Bhanumathi and Sangeetha 2017], apresentam-se os resultados obtidos nesta área de pesquisa de WBANs, mostrando as principais características de mais de 50 protocolos de roteamento desenvolvidos até 2017.

Os protocolos de roteamento propostos para as WBANs foram inicialmente classificados em cinco tipos, de acordo com os seus objetivos [Movassaghi et al. 2013b], [Effatparvar et al. 2016] e estão estritamente relacionados com os desafios identificados em WBANs. Estes grupos são categorizados como: algoritmos de roteamento baseados em cluster, baseados na qualidade de serviços (QoS), baseados em movimentos corporais, cientes da temperatura e protocolos *cross-layered*.

A seguir apresenta-se cada uma destas categorias de protocolos propostos para WBAN. Em cada uma dessas categorias, alguns protocolos encontrados na literatura são comentados.

4.3.3.1. Protocolos de Roteamento Baseados em Cluster

Os protocolos de roteamento baseados em cluster (*Clustered*) são protocolos que tentam agrupar os nós da rede em clusters diferentes e atribuir a um nó, chamado cabeça de cluster (*cluster head*), a missão de encaminhar dados dos sensores de seu cluster, para a o nó coletor (*sink*) ou estação base. Esta transmissão pode ser direta desde o nó cabeça de cluster até o *sink* ou através de outros nós cabeça de cluster. Estes protocolos visam minimizar o número de transmissões diretas de nós sensores para o nó coletor. No entanto, a sobrecarga de número de saltos e atrasos exigidos para a seleção de clusters são as

principais desvantagens desses protocolos.

O protocolo AnyBody foi apresentado em [Watteyne et al. 2007], como um protocolo de auto-organização em que os sensores ligados a uma pessoa são agrupados em clusters. Esse processo é executado em cinco etapas. Primeiro, um nó descobre quais outros nós podem se comunicar diretamente, trocando mensagens HELLO (etapa 1). Então, com base nessas informações, é calculado o parâmetro densidade como a razão entre o número de links e o número de nós dentro da vizinhança a 2 saltos (2-hop), logo depois, cada nó envia uma mensagem HELLO contendo sua densidade e recebe a densidade de seus vizinhos. Baseado nessa densidade, os nós são agrupados em clusters (etapa 2) e é selecionado um cabeça de cluster para cada cluster (etapa 3). A seguir, os clusters são então interconectados (etapa 4) e os caminhos de roteamento são configurados em direção ao nó coletor (etapa 5).

O protocolo HIT proposto em [Culpepper et al. 2004] foi projetado para coletar dados em redes de microssensores sem fio. Este protocolo é baseado em uma arquitetura híbrida que consiste em um ou mais clusters, e cada um baseado em múltiplas transmissões indiretas de múltiplos saltos. Foi desenhado para minimizar o consumo de energia e o atraso da rede, e para atingir esses objetivos são usadas transmissões paralelas tanto na comunicação entre clusters quanto intra clusters. Isso é possível porque cada sensor calcula de forma independente um escalonamento TDMA para o controle do acesso ao meio.

O protocolo HIT consiste nas seguintes fases: (1) Eleição de Cabeças de Clusters: um ou mais cabeças de cluster são eleitos; (2) Anúncio dos Cabeças de Clusters (*Cluster-Head Advertisement*): os cabeças de cluster transmitem seu status por toda a rede para formar um ou mais clusters; (3) Configuração do Cluster (*Cluster Setup*): são formados os clusters e os relacionamentos upstream e downstream de cada cluster, pois várias rotas são descobertas dentro de cada cluster, desde os nós sensores até o cabeça de cluster; (4) Determinação do conjunto de bloqueio (*Blocking Set Computation*): cada nó calcula seu conjunto de bloqueio, que é uma lista de nós que não podem se comunicar ao mesmo tempo com esse nó; (5) Configuração da rota: os sensores dentro de um cluster formam rotas de vários saltos até o cabeça do cluster; (6) Criação de escalonamento TDMA: um escalonamento de TDMA é calculado para permitir transmissões paralelas; (7) Transmissão de dados: é uma fase de estado estacionário longo em que os dados detectados são enviados para a estação base.

De forma geral os protocolos baseados em cluster estão focados principalmente ao melhorar a eficiência energética da rede sem dar atenção aos outros desafios de WBAN. Por isso se considera que estes protocolos não são os mais convenientes para ser utilizados nas WBANs. Também a análise de desempenho desses protocolos, nas propostas dos seus autores, é feita comparando com outros protocolos não projetados para WBANs como o LEACH proposto em [Heinzelman et al. 2000] e PEGASIS proposto em [Lindsey and Raghavendra 2002].

4.3.3.2. Protocolos de Roteamento Baseados em Qualidade de Serviço (QoS)

Os protocolos baseados em QoS fornecem principalmente módulos separados para diferentes métricas de QoS que operam de maneira coordenada. Assim, eles oferecem maior confiabilidade, menor atraso fim-a-fim e maior taxa de entrega de pacotes. Esses protocolos sofrem principalmente de alta complexidade devido aos vários módulos e diferentes métricas de QoS que possuem e operam simultaneamente.

Um exemplo é o protocolo TLQoS (*ThermaL-aware QoS routing protocol*) proposto em [Monowar and Bajaber 2015]. Este é um protocolo de roteamento baseado em QoS consciente da temperatura, que permite ao sistema alcançar a QoS desejada em termos de atraso e confiabilidade para diversos tipos de tráfego, ao mesmo tempo que evita o aquecimento dos nós. Com o fim de fornecer QoS para diversos tipos de tráfego, considerando atraso e confiabilidade como a métrica de QoS, classifica-se o tráfego em quatro tipos: crítico (*Critical traffic*), com restrição de atraso (*Delay constrained traffic*), com restrição de confiabilidade (*Reliability constrained traffic*), regular (*Regular traffic*). Este protocolo utiliza uma abordagem completamente modular para lidar com o tráfego de acordo com suas respectivas demandas de QoS com o menor aumento de temperatura. Para isso, tem um módulo de atraso, um de confiabilidade e um de temperatura. Também conta com um classificador de pacotes com reconhecimento de QoS, que classifica o pacote de acordo com suas demandas de QoS e o envia para o respectivo módulo para o processamento. O módulo de temperatura lida com o pacote regular (sem restrições de atraso e confiabilidade) e garante que o pacote alcance o coletor através de uma rota de temperatura mais baixa. Os outros tipos de pacotes são processados pelo módulo correspondente de acordo com suas demandas de QoS.

Para enfrentar os desafios associados às mudanças do ambiente sem fio no interior do corpo humano, causadas pela variabilidade na perda de caminho (*path loss*) que impõem os diferentes tipos de tecidos do corpo, no protocolo TLQoS se implementa o roteamento localizado baseado em métricas que requerem apenas informações da vizinhança local, para isso define vários potenciais de roteamento, que consistem em funções matemáticas, com base nas métricas de QoS e na temperatura do nó. Para evitar a formação de *loops* de roteamento, e para rotear o pacote em direção ao coletor, reduzindo o número de salto, são introduzidos os potenciais híbridos que consideram a união dos diferentes potenciais de roteamento, e um mecanismo de evitar *loops* de roteamento.

Em [Bhanumathi and Sangeetha 2017], são apresentados 32 protocolos baseados em QoS. Destaca-se o protocolo DMQoS (*Data-centric Multi objective QoS-aware routing protocol*) proposto em [Razzaque et al. 2011]. Foi apresentado como o primeiro projeto completo de um protocolo de roteamento multi-objetivo centrado em QoS para WBANs, que possui clara diferenciação na seleção de rotas entre múltiplos tipos de tráfego em relação aos seus requisitos de QoS. Ele também reduz o custo de energia e a sobrecarga de operação do protocolo, melhorando o desempenho da rede. O DMQoS utiliza arquitetura modular e utiliza localizações geográficas para implementar o roteamento localizado. Realiza o roteamento com reconhecimento de QoS fim-a-fim com decisões locais em cada nó intermediário sem descoberta e manutenção de caminho fim-a-fim. Essa propriedade é importante para a escalabilidade de redes de sensores, auto-adaptabilidade à dinâmica de rede e adequação a várias classes de fluxos de tráfego.

Em DMQoS, os pacotes de dados são divididos em quatro classes: pacotes de dados ordinários (OD - *Ordinary Data Packets*), pacotes de dados controlados por confiabilidade (RP - *Reliability-Driven Data Packets*), pacotes de dados controlados por atraso (DP - *Delay-Driven Data Packets*) e pacotes de dados críticos (CP - *Critical Data Packets*).

A arquitetura de roteamento do DMQoS é constituída por cinco módulos: (1) Classificador de pacote dinâmico: recebe os pacotes de dados do nó vizinho ou das camadas superiores e classifica-os em uma das quatro categorias supracitadas e os encaminha para seus respectivos módulos em uma base FCFS (*First-Come-First-Serve*); (2) Módulo de encaminhamento geográfico com reconhecimento de energia: decide o nó do próximo salto com menor distância e energia residual relativamente alta usando a Otimização Lexicográfica (LO) multi-objetivo. A LO multi-objetivo é utilizada para gerenciar a compensação entre a informação geográfica e a energia residual para garantir uma taxa de consumo de energia homogênea para todos os nós; (3) Módulo de controle de confiabilidade: determina o próximo salto com maior confiabilidade. (4) Módulo de controle de atraso: localiza o próximo salto com menos atraso. (5) Módulo de enfileiramento ciente de QoS: é responsável por encaminhar o pacote de dados recebido para uma das quatro filas de classes com base nas prioridades atribuídas.

Baseado nessa arquitetura, os nós sensores enviam os dados detectados para o coordenador, o qual é um nó central que atua como uma cabeça de cluster e possui menos restrições em termos de energia e capacidade de computação em comparação com os nós sensores.

Em [Khan et al. 2012a], propõe-se o protocolo EPR (*Energy–aware Peering Routing protocol*), que embora esteja dentro do grupo dos baseados em QoS, também foi projetado com o fim de melhorar a confiabilidade e reduzir o tráfego de rede e o consumo de energia. Os autores propõem uma nova arquitetura de rede BAN para ambientes hospitalares internos e um novo mecanismo de descoberta de pares com construção de tabelas de roteamento que ajuda a reduzir a carga de tráfego de rede, o consumo de energia e melhora a confiabilidade da BAN, com base em abordagens centralizadas e distribuídas.

Esta arquitetura considera três tipos de dispositivos de comunicação: (1) Coordenador de monitor médico (MDC - *Medical Display Coordinator*): são dispositivos para exibir os dados do paciente, com fontes de alimentação substituíveis; (2) Coordenador de rede de área corporal (BANC - *Body Area Network Coordinator*): tem energia limitada e é responsável por coletar os dados dos nós sensores e encaminhá-los para os MDCs correspondentes; (3) Coordenador de estação de enfermagem (NSC - *Nursing Station Coordinator*): é um dispositivo centralizado com fonte de alimentação contínua, que mantém o *peering* e o tipo de informação de comunicação de todos os BANCs. Como existem muitos MDCs no hospital, para exibir em tempo real os dados da BAN no MDC dedicado ao paciente correspondente, é proposto um método de *peering* híbrido. Nesse caso, a comunicação BAN vai ter dois modos: centralizado e distribuído. No modo centralizado, a BAN irá se conectar ao NSC para obter as informações de *peering* e no modo distribuído a BAN descobrirá e enviará dados para seus pares.

Nesta proposta, na hora de encaminhar os dados, para selecionar o nó do próximo salto levando em consideração os requisitos de QoS dos dados é usado o protocolo

de roteamento DMQoS. Como diferencial incluem-se três novos aspectos: (1) Troca de mensagens HELLO: esta mensagem entre outras informações contém a energia residual e distância desde o nó emissor até o destino; (2) Tabela de vizinhos: onde cada nó tem a informação referente a cada um dos seus vizinhos; (3) Tabela de rota: já que na tabela de vizinhos podem existir vários registros para uma mesma entrada, considera-se um novo algoritmo de construção de tabela de roteamento para filtrar a tabela de vizinhos e escolher apenas a entrada com o menor custo de comunicação.

O protocolo QPRD (*QoS-aware Peering Routing protocol for Delay sensitive data*) proposto em [Khan et al. 2012b] tenta melhorar o já citado EPR e tem como principal objetivo diminuir o atraso fim-a-fim, classificando os pacotes de dados dos pacientes em duas categorias: Pacotes Ordinários (OP - *Ordinary Packets*) e Pacotes Sensíveis ao Atraso (DSP - *Delay Sensitive Packets*). Para QPRD, diferente de DMQoS, a arquitetura de roteamento é dividida em sete módulos: (1) Receptor MAC: recebe os pacotes de dados dos outros nós; (2) Classificador de pacotes: classifica os pacotes recebidos como pacotes HELLO ou pacotes de dados; (3) Módulo de atraso: monitora os diferentes tipos de atrasos e encaminha os resultados para a camada de rede para descobrir o atraso do nó; (4) Módulo de protocolo HELLO: é o responsável por enviar e receber pacotes HELLO; (5) Módulo de serviço de roteamento; recebe os pacotes de dados das camadas superiores e o classificador de pacotes, os categoriza como OP ou DSP e escolhe o melhor caminho para cada categoria; (6) Módulo de filas com reconhecimento de QoS: encaminha os pacotes de dados recebidos para sua fila correspondente; (7) Transmissor MAC: armazena os pacotes de dados e os pacotes HELLO recebidos em uma fila em modo *First-Come-First-Serve* (FCFS) e os transmite usando o CSMA/CA.

Outro protocolo projetado para melhorar o EPR é o protocolo QPRR (*QoS-aware peering Routing protocol for Reliability sensitive data*) proposto em [Khan et al. 2013]. Particularmente, este protocolo foca-se em melhorar a confiabilidade fim-a-fim, para satisfazer a necessidade dos dados sensíveis à confiabilidade. Para isso, classifica os dados como Pacotes Ordinários (OP) e Pacotes Sensíveis à Confiabilidade (RSP - *Reliability Sensitive Packets*). A arquitetura de roteamento é a mesma que a de QPRD, com a diferença de que o módulo de atraso é substituído por um módulo de confiabilidade, o qual que é responsável por monitorar e calcular a confiabilidade do link entre dois nós.

O protocolo DMQoS é um dos mais conhecidos dos protocolos cientes de QoS, porque pode diminuir o atraso para informações sensíveis ao atraso e, da mesma forma, pode fornecer roteamento confiável para informações confidenciais. Porém, os protocolos EPR, QPRD e QPRR têm menor consumo de energia quando comparados a outros protocolos de seu tipo.

4.3.3.3. Protocolos de Roteamento Baseados em Movimentos Corporais

Características de WBANs como baixa potência de transmissão para evitar o aquecimento dos tecidos e poupar energia, unidas aos movimentos do corpo humano, criam um cenário onde podem acontecer perdas dos enlaces e mudanças na topologia da rede. Os protocolos baseados nos movimentos corporais tentam enfrentar esse problema de particionamento topológico ou desconexão dos enlaces, causados pelo movimento corporal. Em [Bhanu-

mathi and Sangeetha 2017], são abordados 6 protocolos projetados para este fim, deles os dois com melhor desempenho nos cenários WBAN são o protocolo de roteamento oportunista (*Opportunistic routing*) proposto em [Maskooki et al. 2011] e o protocolo ETPA (*Energy efficient thermal and power aware routing*) apresentado em [Movassaghi et al. 2012].

O protocolo de roteamento oportunista (*Opportunistic routing*) para redes corporais sem fio, proposto em [Maskooki et al. 2011], foi projetado para aumentar a vida útil da rede a partir do movimento das partes do corpo. Como nas WBANs o nó coletor é o dispositivo de maior consumo energético, seria conveniente que seja posicionado em uma parte do corpo de forma não invasiva, permitindo que a bateria possa ser trocada facilmente. Neste protocolo é proposto um modelo de rede, no qual o nó coletor é alocado no punho, por exemplo podendo ser incorporado em um relógio de pulso. Outra vantagem dessa alocação do nó coletor é evitar altas atenuações do sinal RF (Rádio Frequência), que em algumas direções seriam totalmente mascaradas pelo corpo. Por outro lado, o nó sensor está localizado no tórax e mede alguns dados do corpo periodicamente e os envia para uma rede externa através do nó coletor. Também utiliza um nó retransmissor (*relay*) na lateral da cintura, de forma que nessa posição tem linha de visão (LOS - *Line of Sight*) com o sensor de tórax e o nó coletor.

Quando o nó sensor deseja enviar um pacote de dados ao coletor, em primeiro lugar, enviará um quadro RTS (*Request to Send*). O sinal RTS é enviado com o nível de potência que apenas nós da linha de visão podem receber. Se o nó da mão (coletor) estiver na posição LOS, ele enviará um quadro ACK (*Acknowledge*) de volta ao nó sensor dentro de um intervalo de tempo limite especificado. Então, o nó sensor enviará seu pacote diretamente para o coletor. No entanto, se não existe LOS entre o sensor e o coletor, o nó coletor não receberá o quadro RTS e, subsequentemente, o nó sensor não receberá o quadro ACK do coletor no intervalo de tempo limite. Depois do tempo limite, o nó sensor enviará um sinal de ativação para o nó *relay* que só tem o receptor ligado. Quando o nó *relay* estiver pronto, ele enviará um sinal para o nó sensor e o nó coletor para iniciar a comunicação e, em seguida, retransmite os dados do sensor no peito para o coletor da mão. No final da comunicação, o nó coletor enviará uma confirmação de recebimento (RAck - *Receive Acknowledge*) para o nó sensor. Se nenhum RAck for recebido, o procedimento acima será repetido até que ocorra uma comunicação com sucesso. Nesse esquema, se assume que os tempos de associação com o coletor e de envio de pacotes, são muito menores do que os movimentos da mão, portanto, as variações do canal não são consideráveis. Este protocolo tem o consumo energético mais baixo comparado com os de seu tipo.

O protocolo ETPA (*Energy efficient Thermal and Power Aware routing*) apresentado em [Movassaghi et al. 2012] foi projetado principalmente com o fim de reduzir a temperatura do nó e evitar a formação de pontos quentes. Embora considere os níveis de energia e a temperatura dos nós no cálculo da função de custo, é uma abordagem baseada na postura e movimento do corpo humano, já que é baseado no esquema de roteamento definido no protocolo PRPLC (*Probabilistic Routing with Postural Link Costs*) apresentado em [Quwaider and Biswas 2009]. No PRPLC, o impacto da mobilidade da postura humana no particionamento da rede é considerado a partir do Fator de Verossimilhança do Enlace (LLF - *Link Likelihood Factor*), o qual é a probabilidade de que qualquer enlace entre dois nós " i " e " j " estejam conectados por um intervalo de tempo discreto " t ".

A proposta considera WBAN com sete nós colocados no corpo (dois nós nas coxas, dois nos tornozelos, dois na parte superior dos braços e um na cintura). Para evitar a escuta ociosa e diminuir a interferência, os quadros são divididos em intervalos de tempo usando um esquema TDMA (*Time Division Multiple Access*). Durante cada ciclo, todos os nós transmitem, em seu intervalo de tempo alocado, uma mensagem HELLO para todos os seus vizinhos contendo a temperatura e a energia residual dele. Então, cada nó calcula a energia recebida dos nós vizinhos. e utiliza essa temperatura, energia residual e transmissão de energia do nó para calcular a função de custo. Quando um nó precisa empacotar para enviar, ele procura por uma rota eficiente com custo mínimo. Se encontrar nós com uma rota eficiente, ele encaminha o pacote, caso contrário, ele armazena o pacote em um buffer. Caso não se encontre uma rota antes de percorrer um tempo igual à duração de dois quadros, ou seja dois ciclos TDMA, o pacote armazenado no buffer é descartado. Também para diminuir o atraso, cada pacote só pode passar por um número predefinido de saltos (*max_hop_count*), caso contrário, ele será descartado.

4.3.3.4. Protocolos de Roteamento Baseados na Temperatura

Os protocolos baseados em temperatura foram projetados com o objetivo principal de minimizar o aumento da temperatura local ou geral do sistema. De fato, a ideia subjacente a esses protocolos é rotear dados em diferentes rotas para evitar um aumento dramático da temperatura em alguns nós, os quais podem causar danos ao tecido humano. Uma gama de protocolos, abordados em [Movassaghi et al. 2013b] e [Oey and Moh 2013], foram propostos seguindo esta abordagem. No entanto, esses protocolos não incluem métricas de qualidade de rede nem as utilizam de forma alguma. Portanto, eles sofrem com a complexidade do sistema e a sobrecarga do uso de recursos de rede, o que aumenta drasticamente com mais nós. Atualmente, esta abordagem está presente apenas de uma forma combinada com outras abordagens. Dos dez protocolos de roteamento cientes da temperatura abordados em [Bhanumathi and Sangeetha 2017], os que têm melhores resultados para enfrentar os desafios de WBAN são o protocolo M2E2 (*Multi-Mode Energy-Efficient Multihop Protocol*) proposto em [Rafatkhah and Lighvan 2014] e o algoritmo de roteamento HPR (*Hotspot Preventing Routing*) proposto em [Bag and Bassiouni 2007].

O protocolo M2E2 (*Multi-Mode Energy-Efficient Multihop Protocol*) foi proposto em [Rafatkhah and Lighvan 2014]. Além de reduzir os pontos quentes em WBAN heterogêneas, também reduz o consumo de energia e aumenta o tempo de vida da rede. Neste protocolo se considera um protótipo de rede no qual um nó coletor está localizado no centro do corpo humano e o outro está localizado no domicílio. Para enfrentar os inconvenientes de WBANs heterogêneas, os nós sensores são organizados no corpo humano com base em sua taxa de dados. Os nós sensores com taxas de dados altas são denominados nós sensores pai e estão alocados nas partes menos móveis do corpo humano, e estão ligados diretamente ao coletor no corpo. Os outros sensores de baixas taxas, chamados nós filhos, alocados nas partes do corpo com maior movimento, podem se comunicar diretamente com o coletor ou através de outros nós filhos, em um caminho multi-salto. Por outro lado os nós alocados em casa são capazes de enviar sinais chamados *Home-Signal* e os nós colocados no corpo humano são capazes de recebê-lo. Para o gerenciamento de energia, utiliza uma comunicação combinada de salto único e múltiplos saltos. Este

protocolo apresenta o melhor desempenho quando é comparado com os demais de sua classificação.

M2E2 consiste em quatro fases principais: (1) Fase de inicialização: onde se transmitem mensagens "HELLO"para todos os nós com o objetivo de informar a vizinhança, a posição do nó coletor e todas as rotas possíveis até ele. Os nós sensores atualizam sua tabela de roteamento enquanto trocam as mensagens HELLO. No caso de receber o *Home-Signal*, o processo de roteamento será formado através de nós sensores localizados em casa; caso contrário, o processo de roteamento será formado através de nós sensores no corpo humano; (2) Fase de roteamento: nessa fase, se o *Home-Signal* for recebido, um dos nós sensores no corpo humano é vinculado à tabela de roteamento dos nós fixos em casa. Neste caso, utiliza-se um salto único para o envio de todos os dados. Se o *Home-Signal* não for recebido, a tabela de roteamento é formada nos nós do corpo. Neste outro caso, a fim de reduzir a perda de energia, é usada a comunicação multi-salto para dados normais, mas para dados de emergência, todos os nós sensores implantados no corpo enviarão esses dados diretamente para a estação base; (3) Fase de agendamento: após a seleção da rota na fase de roteamento, o nó coletor programa um intervalo de tempo para a comunicação entre o coletor e os nós sensores com base no TDMA. (4) Fase de transmissão de dados: os nós sensores enviam seus dados para o nó coletor no intervalo de tempo designado.

Em [Bag and Bassiouni 2007], é proposto o algoritmo de roteamento HPR (*Hotspot Preventing Routing*). Este é uma versão melhorada dos protocolos cientes da temperatura LTR (*Least Temperature Routing*) e ALTR (*Adaptive Least Temperature Routing*), abordados em [Oey and Moh 2013]. HPR tem como objetivo evitar a formação de pontos quentes e reduzir o atraso médio na entrega de pacotes e foi implementado em duas fases: (1) Fase de configuração: onde todos os nós trocam as informações do caminho mais curto e da temperatura inicial. Com base nessas informações, cada nó cria sua própria tabela de roteamento; (2) Fase de roteamento: cada nó encaminha os pacotes para o destino usando a rota de menor número de saltos. Um contador de saltos, associado a cada pacote, é incrementado cada vez que um nó reenvia o pacote, e quando o contador exceder o valor limite se descarta o pacote. Se o nó de destino for um dos nós vizinhos, o pacote será encaminhado diretamente. Caso contrário, ele é encaminhado para o nó de próximo salto no caminho mais curto até o destino com temperatura menor ou igual ao valor limite. A temperatura limite de um nó é derivada da temperatura média dos nós vizinhos e da própria temperatura do nó. Se a temperatura do próximo salto no caminho mais curto até o destino for maior que a soma da temperatura do nó de origem e a temperatura limite, o nó o identifica como um ponto quente. Então o pacote é enviado a outro nó vizinho que não tenha sido visitado pelo pacote (para evitar *loops* de roteamento), com menor temperatura.

4.3.3.5. Protocolos *Cross-layered*

Os protocolos de roteamento *Cross-layered* abordam os desafios das camadas de rede e MAC ao mesmo tempo para melhorar o desempenho geral da rede WBAN. Embora esses protocolos tenham alta produtividade, baixo consumo de energia e um atraso fim-a-fim

relativamente fixo, segundo [Movassaghi et al. 2013b], eles não podem fornecer alto desempenho em casos de movimento do corpo e devido às perdas de caminho associadas aos tecidos do corpo humano em alguns cenários. Dos quatros protocolos projetados para esse fim descritos em [Bhanumathi and Sangeetha 2017], os que têm melhor desempenho são o protocolo CICADA (*Cascading Information retrieval by Controlling Access with Distributed slot Assignment*) proposto em [Latré et al. 2007] e o protocolo TICOSS (*TImezone COordinated Sleep Scheduling*) proposto em [Ruzzelli et al. 2007].

O protocolo CICADA (*Cascading Information retrieval by Controlling Access with Distributed slot Assignment*)[Latré et al. 2007] foi proposto com o fim de introduzir menor atraso e baixo consumo de energia, sendo o protocolo que oferece melhor desempenho nessas métricas, dentre os de sua classificação. Neste protocolo, configura-se uma árvore de maneira distribuída e essa estrutura de árvore é usada subsequentemente para garantir um acesso ao meio livre de colisão e para encaminhar dados para o coletor. O uso de energia é baixo, pois é projetado com base em escalonamento TDMA, e os nós podem dormir nos *slots* de tempo onde não estão transmitindo ou recebendo.

A alocação de *slots* de tempo é feita enviando um esquema de um nó pai para um nó filho. Então, um nó calcula seu próprio esquema com base no esquema que recebeu de seu pai. Cada ciclo é dividido em duas partes: o subciclo de controle e o subciclo de dados. Cada subciclo tem seu próprio esquema para alocação de *slots*, o esquema de controle e o esquema de dados. Esses esquemas são enviados no subciclo de controle e são usados para propagar os esquemas dos pais para os filhos. Quando todos os nós recebem seu esquema, o ciclo de controle é finalizado e inicia-se o ciclo de dados. O esquema de dados consiste em 2 partes também, um período de dados e um período de espera. No período de espera, o nó deve permanecer em silêncio e deve desligar seu rádio. No período de dados, o nó recebe dados de seus filhos e envia dados para seu pai. Cada nó pai constrói uma tabela de seus nós filhos que contém o número de *slots* necessários para transmitir os dados ao nó pai e o número de *slots* necessários para receber os dados de seus nós filhos. Cada subciclo de dados tem um *slot* para que novos nós possam entrar na árvore, para isso cada novo nó filho tem permissão para enviar uma mensagem de requisição de entrada na árvore (JOIN-REQUEST) nesse *slot* depois de ouvir o esquema de dados do nó pai desejado.

O protocolo TICOSS (*TImezone COordinated Sleep Scheduling*), proposto em [Ruzzelli et al. 2007], procura melhorar o padrão 802.15.4 através da divisão da rede em zonas de tempo (*timezones*). Para otimizar o padrão 802.15.4, neste protocolo os nós usam períodos alternados de atividade e inatividade para reduzir o consumo de energia e reduzir as colisões de pacotes ocorridas devido aos nós ocultos. Os nós encaminham os pacotes de dados para o nó coordenador usando o roteamento de caminho mais curto. A ideia de dividir a rede em zonas de tempo é adotada por meio de uma tabela conhecida como tabela V, para a programação de transmissão, e a implementação de três *buffers* FIFO para: (1) pacotes *upstream* destinados ao *gateway*; (2) pacotes *downstream*, destinados à rede; (3) pacotes para transmissão local. A principal tarefa do agendamento da tabela V é dividir o tempo em *slots* que são usados para transmitir os pacotes dos *buffers*.

4.3.3.6. Protocolos MANET em WBANs

Características de WBANs como baixas potências de transmissão unido aos movimentos do corpo humano, criam um cenário com mobilidade e restrições de recursos, que também é característico de redes ad-hoc móveis. Porém, em trabalhos como [Latré et al. 2011], [Ullah et al. 2012], [Movassaghi et al. 2013b], é reiterada afirmação de que as restrições de WBAN não podem ser atendidas pelos protocolos de roteamento das redes de sensores sem fio (WSN) e das redes ad-hoc móveis, isso apenas baseado em afirmações conceituais. Por isso, as pesquisas relativas aos protocolos de encaminhamento para WBAN têm sido focadas mais no desenvolvimento de novos protocolos para enfrentar algum dos desafios definidos para WBAN, sem considerar protocolos já desenvolvidos para redes ad-hoc como AODV [Perkins et al. 2003], DSR [Johnson et al. 2007], DSDV [Perkins and Bhagwat 1994], entre outros.

Por outro lado, considerando que o AODV fornece um bom desempenho em cenários com mobilidade e restrições de recursos, em [Ferreira et al. 2017b] se faz uma análise do uso do AODV para o cenário de WBAN. Após a avaliação feita, neste trabalho se mostrou que o AODV é uma alternativa para o roteamento em WBAN, embora precise ser optimizado para um melhor desempenho frente aos desafios de WBANs. De forma similar, outros trabalhos avaliam o desempenho de diferentes protocolos MANETs em cenários WBANs, conforme comentado a seguir.

Em [Ferreira et al. 2017a], apresenta-se um estudo de estabilidade de rotas em cenários WBAN, baseado em dados reais e o protocolo de roteamento AODV. Os resultados experimentais demonstram que o critério de rota com enlaces de melhor qualidade esperado não foi respeitado sempre. Devido principalmente a que o protocolo AODV escolhe a rota encontrada mais rapidamente e não considera uma métrica de qualidade no roteamento. Portanto, afirma-se que a utilização de uma métrica de qualidade do enlace para formação de rotas permitiria evitar as rotas com menor taxa de entrega.

Em [Asogwa et al. 2012], é feita uma análise de desempenho dos protocolos de roteamento AODV, DSR e DSDV, em um cenário baseado nas principais características de WBANs e usando o protocolo IEEE 802.15.4. A análise é feita com a finalidade de examinar a possibilidade de utilizar protocolos MANET em redes WBANs e qual deles seria o melhor. Utilizou-se como métrica de avaliação, a confiabilidade e o atraso. Análises experimentais mostraram que os protocolos reativos AODV e DSR têm boa confiabilidade, enquanto o protocolo proativo DSDV teve perdas de pacotes superiores a 90%.

De forma similar, o trabalho apresentado em [He et al. 2015] realiza uma comparação do desempenho dos protocolos de roteamento AODV e DSDV para cenários WBAN considerando: atraso fim-a-fim médio, taxa de transferência média fim-a-fim, e taxa média de perda de pacotes. Os resultados mostram que AODV tem melhor desempenho.

Em [Tiwari et al. 2014], realiza-se uma comparação e avaliação do desempenho dos protocolos reativos AODV e DSR em cenários WBAN. Os resultados do estudo comparativo, baseado no atraso fim-a-fim e na vazão, mostram que o protocolo de roteamento AODV supera o DSR. No caso do AODV, o atraso é minimizado e a vazão é maximizada em comparação com o protocolo de roteamento DSR. Também mostra que o protocolo de roteamento AODV diminui significativamente o problema de sobrecarga de controle

encontrado no protocolo DSR, já que o RREQ (*Route REQuest*) em DSR transporta a informação da rota completa, enquanto em AODV, RREQ carrega só o endereço de destino. Finalmente, se conclui que o protocolo de roteamento AODV tem uma taxa de sucesso mais alta e uma resposta mais rápida às alterações da topologia da rede.

Por outro lado, em [Kumari and Nand 2016], são comparados os protocolos de roteamento AODV, DSDV, DSR e AOMDV, tanto em cenários WSN quanto WBAN. Como métricas, na avaliação, se consideraram: taxa de entrega de pacotes (PDR - *Packet Delivery Ratio*), atraso fim-a-fim (E2Edelay - *End-to-End delay*) e vazão. O artigo mostra que AODV e sua variante melhorada AOMDV (*An Optimized Ad-hoc On-demand Multipath Distance Vector*) tem o melhor desempenho em termos de PDR, atraso e vazão, tanto para cenários WSN quanto WBAN. Também se mostra claramente que o desempenho desses protocolos é reduzido no ambiente WBAN em comparação a WSN.

Do estudo da literatura, conclui-se que embora o desempenho dos protocolos MANET seja diminuído pelas características das redes corporais, é viável o uso desses protocolos em WBANs, mesmo que ainda exista a necessidade de adaptá-los para enfrentar melhor os desafios de WBAN.

4.4. ISO/IEEE 11073

Com o grande avanço de tecnologias para o monitoramento de pacientes e o crescente interesse por tecnologias digitais aplicadas à saúde, desencadeou-se o desenvolvimento de Dispositivos Pessoais de Saúde (DPS) com interfaces de comunicação embutidas, tais como USB, *Bluetooth* e *ZigBee*. Em cenários que fazem uso de DPS, o paciente também é responsável pelo monitoramento de sua própria saúde e bem-estar [Martins et al. 2014].

Devido ao surgimento desses novos dispositivos, um grupo de trabalho do IEEE definiu a família de normas denominadas ISO/IEEE 11073 [IEEE Std 11073-00103 2012]. Esta família de normas começou a ser especificada na década de 1990, e tinha como finalidade conectar dispositivos médicos em unidades de saúde. Monitores de sinais vitais e monitores de pressão arterial são alguns exemplos desses dispositivos. Seu uso era feito principalmente por profissionais da saúde (médicos, enfermeiras, etc). Contudo, dispositivos de saúde e equipamentos para exercícios físicos (*fitness*) alcançaram o mercado e seu uso doméstico vem crescendo a cada ano. O termo DPS envolve tanto dispositivos médicos, quanto dispositivos de saúde e *fitness* usados por usuários "leigos" em suas residências [IEEE Std 11073-00103 2012]. É comum a venda desses dispositivos juntamente com produtos eletrônicos de consumo geral.

A família de normas ISO/IEEE 11073 é divida em três grupos, a primeira e mais antiga parte é a ISO/IEEE 11073 *Lower Layer*, que especifica protocolos e serviços de comunicação orientados a conexão, utilizando camadas físicas como infravermelho, tecnologia RF sem fio ou Ethernet [IEEE Std 11073-20101 2004]. Já a parte ISO/IEEE 11073 *Point-of-Care-Devices* especifica normas de comunicação para dispositivos que são usados exclusivamente em unidades de saúde. Por fim, a ISO/IEEE 11073 *Personal Health Devices* (PHD), ou Dispositivo Pessoal de Saúde (DPS), define normas para dispositivos usados pelos usuários em suas casas. Portanto, o foco desta seção será apenas no ISO/IEEE 11073 *Personal Health Devices*. Por questões de simplificação, esta norma será referenciada apenas como IEEE 11073 neste texto.

A norma IEEE 11073 define dois tipos de dispositivos: Agentes e Gerenciadores. Os agentes são tipicamente sensores ou atuadores, de baixa potência e com pouco poder de processamento, enquanto os gerenciadores são dispositivos com um poder de processamento maior, na qual, podem ou não, estar conectados a uma fonte de energia. Apesar de a norma 11073 não comentar explicitamente, dispositivos agentes poderiam ser nós de uma WBAN.

Um dispositivo médico típico é geralmente utilizado em hospitais e manuseado por especialistas. Os dados gerados por esses dispositivos são utilizados para criar diagnósticos e para tratamentos. Dispositivos similares são muitos comuns no dia-a-dia com propósitos de melhorar a saúde, bem-estar, estado físico, etc. Esses aparelhos coletam dados que podem se tornar úteis se compartilhados e reutilizados de forma inovadora. Como por exemplo, uma balança pode enviar leituras para um *personal trainer* específico, ou um médico pode receber dados de leitura de um monitor de pressão sanguínea de seu paciente.

Assim, antes de os dados chegarem em seu destino final (para um especialista da saúde por exemplo), eles devem antes passar por vários outros processos e transformações. Desta forma, o principal objetivo da norma IEEE 11073 é prover um protocolo de comunicação de dados que torne possível implementar DPS com pouca capacidade de processamento.

Os dados gerados por DPS agentes, são transmitidos, primeiramente, para os gerenciadores, que, posteriormente enviam os dados para processamento em centros de telessaúde, profissionais da saúde, amigos, parentes, *personal trainers*, entre outros. O padrão IEEE 11073, portanto, cobre apenas a primeira fase dessa jornada, ou seja, a transmissão dos dados entre os agentes e gerenciadores. *Smartphones*, *desktops* e *tablets* são alguns tipos típicos de dispositivos gerenciadores. Agentes e gerenciadores são conectados, geralmente, através de uma rede local (LAN) ou pessoal (PAN). A Figura 4.10 mostra a comunicação entre DPS e outros componentes de um sistema de saúde.

A fim de oferecer um conjunto completo de padrões, que cubra não apenas a comunicação entre os agentes e os gerenciadores, mas sim, todo o caminho que os dados fazem até chegaram ao seu destino final, A *Personal Connected Health Alliance* (PCHAlliance) publica e promove o *Continua Design Guidelines* (CDG), que define um framework para conexão fim-a-fim de DPS, usando padrões abertos, para criar uma conexão segura e interoperável entre esses dispositivos.

O CDG oferece um conjunto claro de interfaces que permite o fluxo seguro de dados entre os sensores, *gateways* e centros de telessaúde. Além disso, esse guia remove ambiguidade dos padrões para assegurar um ecossistema consistente e interoperável para DPS [PCHAlliance 2017].

Em [Martins et al. 2014], é proposta uma nova arquitetura baseada em dispositivos UPnP (*Plug and Play*) para *Personal Mobile Health*, onde diferentes tipos de dispositivos eletrônicos podem fazer a troca de informações sobre saúde através de tecnologias como XML (eXtended Markup Language).

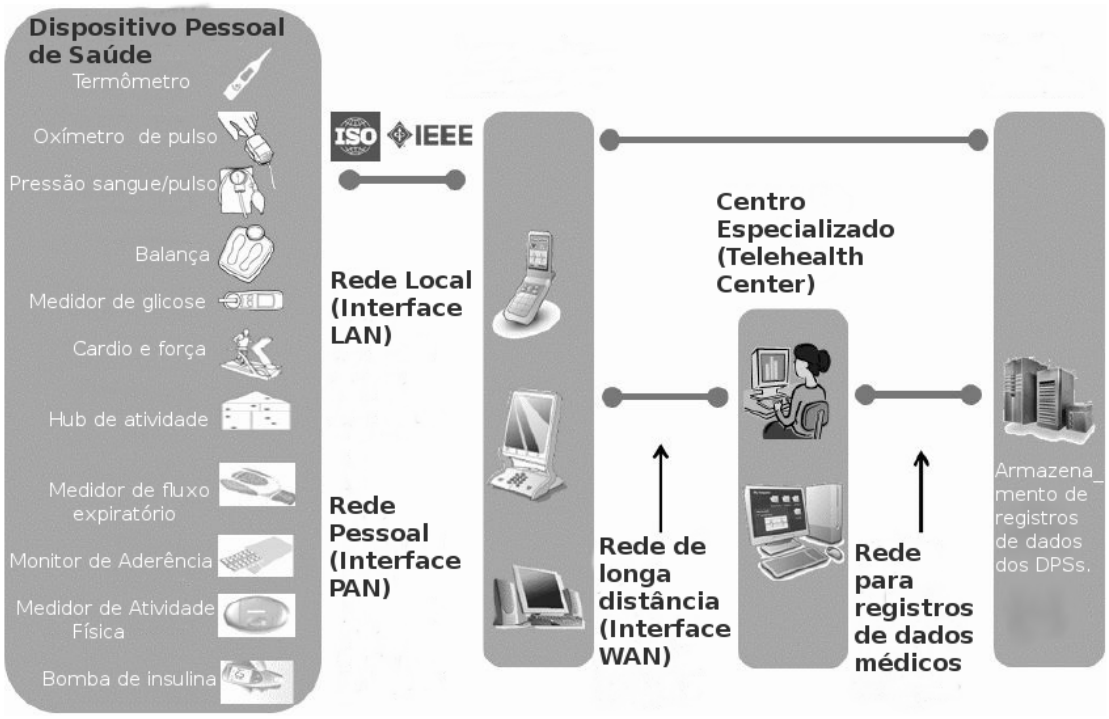


Figura 4.10. Comunicação entre DPS - Continua Connected Health Alliance (Adaptada de [IEEE Std 11073-00103 2012])

.

4.4.1. Casos de Uso

Em vários contextos, DPS são úteis para o suporte de atividades individuais de diferentes formas. Três grandes famílias de casos de usos podem ser destacadas: Saúde e Atividade Física, Independência para Terceira Idade e Gerenciamento de Doenças.

Equipamentos de força, monitores de pressão sanguínea e balanças são equipamentos frequentemente utilizados para Saúde e Atividades físicas. Os usuários deste grupo, em geral, são pessoas que utilizam vários equipamentos de atividades físicas para manter a saúde em dia. Os dados coletados pelos agentes podem ser armazenados em um PC, e posteriormente, transmitidos para *personal trainers* para avaliação. Os usuários possuem e operam seus próprios equipamentos, e buscam por ajuda de profissionais para montar ou melhorar seus planos de atividades físicas.

Já para Independência para a Terceira Idade, dispositivos como *hub* de atividade de vida independente, monitor de medicação, monitor de pressão sanguínea, balança, termômetro, medidor de glicose e oxímetro de pulso são equipamentos tipicamente utilizados. “Independência” ou “envelhecimento independente” são termos usados, por exemplo, pela *Personal Health Connected Continua Alliance*. Casas inteligentes, com dispositivos sensores e atuadores conectados, podem estender o período de tempo de vida independente para pessoas idosas. Funcionalidades como gerenciamento de aquecedores e abertura de portas podem melhorar a qualidade de vida substanciavelmente.

Por fim, para o gerenciamento de doenças, também pode-se utilizar o *hub* de atividade de vida independente, monitor de medicação, monitor de pressão sanguínea, ba-

lança, termômetro, medidor de glicose e oxímetro de pulso. Neste grupo, estão pessoas que possuem, por exemplo, sobrepeso e hipertensão, e se não houver tratamento e acompanhamento, podem correr o risco de desenvolver problemas mais graves posteriormente, como doenças crônicas. Dispositivos que fazem o acompanhamento e gerenciamento de tratamentos, podem evitar futuros gastos com consultas médicas, além de proporcionar um acompanhamento da saúde do paciente sem que ele precise sair de casa.

Em implementações atuais de DPS, os usuários precisam manualmente transferir as medidas dos dispositivos para portais na Internet ou sistemas, se desejarem compartilhá-las. Espera-se que, no futuro, esses dispositivos possam fazer esse compartilhamento de forma autônoma e inteligente para sistemas de telessaúde. Dispositivos Pessoais de Saúde sem fio podem ser embutidos em roupas (*smart clothing*) para o registro de frequência cardíaca, altitude, velocidade, perda de calorias e duração de treinos.

Em ambientes onde um DPS é compartilhado, como famílias, lares para idosos e hospitais, um único dispositivo fará várias medições de diferentes pacientes. Uma vez que, esses dispositivos estão conectados a uma rede, métodos seguros de identificação de pacientes são necessários.

Sendo assim, o IEEE 11073 permite o transporte de identificação de pacientes nos dados. De qualquer forma, normas e tecnologias dedicadas à segurança já estão disponíveis, cartões inteligentes, pulseiras, códigos de barras, RFID e muitos outros métodos.

4.4.2. Organização da Norma 11073

Interoperabilidade entre dispositivos de diferentes fabricantes é alcançada quando todos os sistemas envolvido implementam um conjunto comum de normas. A família de normas 11073 foi projetada para a comunicação entre qualquer dispositivo médico.

Os dispositivos agentes são frequentemente de baixo custo, possuem capacidades limitadas de hardware limitada, como memória RAM, CPU e fontes de energia limitadas com utilização de baterias pequenas. Por razões de simplicidade, possuem configuração fixa e são desconectados quando inativos. Em contrapartida, os dispositivos gerenciadores têm mais capacidades de hardware e podem se conectar a vários agentes. Podem possuir uma fonte de energia fixa ou baterias grandes.

A norma IEEE 11073 descreve a estrutura de comportamento dos agentes e gerenciadores utilizando o conceito de *Domain Information Model* (DIM). O DIM descreve as partes e mostra como essas partes devem ser “colocadas” juntas para formar um elemento maior. Nas implementações, são geralmente chamadas de "classes"ou "objetos". A linguagem ASN.1 (*Abstract Syntax Notation Number One*) descreve os objetos e serviços utilizados no DIM. ASN.1 é uma notação formal tradicionalmente usada para descrever transmissão de dados feitas através de protocolos de telecomunicação.

A família de normas 11073 inclui perfis de especialização, isto é, cada DPS possui uma norma associada, que descreve como esse dispositivo faz sua representação de dados e como as informações são transmitidas para os gerenciadores.

Por exemplo, a norma 11073-10408 define padrões para um termômetro, já a norma 11073-10415 para uma balança. Essas normas são chamadas de “especializações de dispositivos”. Os padrões de especialização de dispositivos definem por exemplo, o

modelo de informação do dispositivo. Neste modelo de informação, são especificados todos os atributos de cada classe ou objeto, assim como métodos e eventos. A Figura 4.11 apresenta o modelo de domínio de informação para um termômetro, conforme a norma 11073-10408.

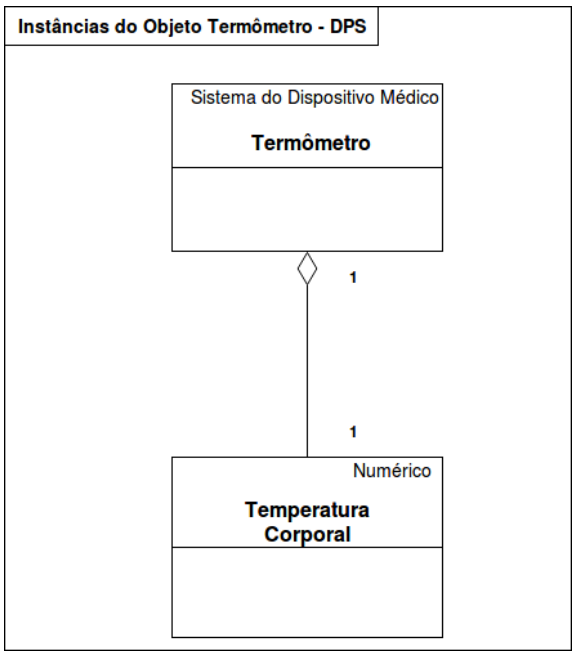


Figura 4.11. Termômetro - Modelo de Domínio de Informação.

A norma IEEE 11073-20601 define um protocolo chamado *Optimized Exchange Protocol*, que é comentado na próxima seção.

4.4.3. Optimized Exchange Protocol - IEEE 11073-20601

Dentro da família de normas 11073, a parte 11073-20601 define um protocolo chamado *Optimized Exchange Protocol* (OEP) que estabelece um framework para um modelo abstrato, o qual implementa uma conexão lógica entre sistemas, garantindo a interoperabilidade entre dispositivos de diferentes fabricantes [IEEE Std 11073-20601 2010].

Um dos serviços básicos definidos na norma é a associação entre um agente e um gerenciador. Vários agentes podem estar associados a um gerenciador. A conexão/associação pode acontecer em ambas as direções, porém, o agente normalmente inicia o processo de associação, pois é ele quem detecta novos dados a serem transmitidos. Essa associação consiste nos seguintes passos, como apresentados na Figura 4.12:

O procedimento de associação começa com o agente enviando uma mensagem de *association request* para o gerenciador. Essa mensagem contém, por exemplo, a identificação do agente, informações das funcionalidades que o agente suporta, informações de codificação e decodificação de mensagens. Após o gerenciador receber esta mensagem, ele verifica se é compatível com este tipo de agente, ou se a configuração já é conhecida. Caso a configuração não seja conhecida, o gerenciador responde com uma mensagem de *association response* dizendo que a associação pode ser feita, porém, parâmetros de configuração do agente são necessários. Na Figura 4.12, é considerado que o gerenciador já

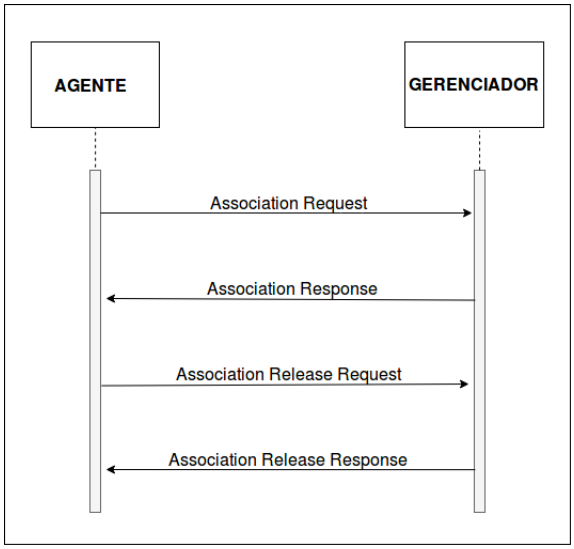


Figura 4.12. Diagrama de sequência da associação de um DPS com um gerenciador.

havia se conectado previamente ao agente, portanto, a mensagem de *association response* não contém o campo de solicitação de parâmetros de configuração do agente.

Quando o agente termina de enviar todos os dados, ele inicia o procedimento de desassociação e envia uma mensagem *association release request* para o gerenciador. O gerenciador vai responder com a mensagem *associaton release response*. Enfim, ambos os dispositivos vão entrar no estado não associado.

As conexões são normalmente feitas ponta-a-ponto e um agente trabalha com um único gerenciador, entretanto, um gerenciador pode trabalhar com múltiplos agentes. Os dados são transportados por APDUs (*Application Protocol Data Units*. Estas APDUs devem ser processadas automaticamente e podem ser segmentadas e remontadas. O tamanho máximo de uma APDU quando enviada de um agente para um gerenciador é de 63 KB, e quando enviada de um gerenciador para um agente é de 8 KB. Esses limites podem ser reduzidos pelas especializações dos dispositivos.

Antidote IEEE 11073 é uma implementação do protocolo *Optimized Exchange Protocol* (IEEE 11073-20601), desenvolvido pela Signove, como parte da *SigHealth Platform*[1]. Esta biblioteca é a primeira implementação de código aberto desse protocolo. Foi desenvolvida em ANSI-C com arquitetura modular, a qual permite a portabilidade do código para diferentes plataformas. Esta biblioteca também permite a comunicação com dispositivos certificados da *Bluetooth Continua Health Alliance* [Martins et al. 2014].

As tecnologias utilizadas para a transmissão de dados de medições biofísicas são divididas em dois grandes grupos: tecnologias de comunicação cabeadas e sem fio. Algumas dessas tecnologias são descritas resumidamente nos próximos parágrafos.

Wireless Personal Area Network (WPAN) é suportada por vários DPS presentes no mercado e está largamente disponível em *laptops* e *smartphones*. Uma WPAN é fre-

[1] *SigHealth* é uma plataforma de monitoramento remoto de pacientes e gestão de dados utilizando dispositivos pessoais sem fio para a saúde.

quentemente utilizada para transferir dados de dispositivos móveis para um *smartphone* em curtas distâncias, que posteriormente, transmite os dados para um centro especializado por meio de uma rede WAN. *Cabled PAN bus* é uma tecnologia amplamente utilizada para conectar dispositivos eletrônicos como *laptops* e *desktops* [IEEE Std 11073-00103 2012]. É utilizada também para o carregamento da baterias de eletrônicos e *smartphones*. Essa tecnologia é usada com DPS quando o uso de uma tecnologia sem fio não é viável por motivos técnicos ou por razões de usabilidade. Uma tecnologia muito comum para DPS é WPAN ou WLAN. Dispositivos pequenos e de baixa potência utilizam essa tecnologia para curtas distância, por exemplo, dentro de uma quarto ou prédio.

Já o uso de TCP, UDP e IP sobre uma rede LAN ou WLAN é realizado por dispositivos de saúde que requerem transferências de volumes maiores de dados. Esses dispositivos geralmente possuem fontes de energia fixa. Por fim, a tecnologia RFID é usada para fins de identificação. Pacientes que permanecem em hospitais por dias ou semanas podem usar pulseiras que facilitam sua identificação.

4.4.4. Segurança

Dentro do contexto de comunicação de DPS, a segurança é bastante importante, pois informações médicas são consideradas altamente sensíveis. Proteção adequada para estas informações são necessárias tanto para a troca de dados entre agentes e gerenciadores, quanto para a transmissão pela Internet. Entretanto, até o momento, a família de normas 11073 não provê métodos que garantem a segurança. Ela assume que essa proteção é garantida por outras tecnologias.

Quatro princípios centrais regem a segurança da informação em DPS: confidencialidade, integridade, disponibilidade e repúdio.

Confidencialidade é definida pela ISO como “assegurar que a informação seja acessível apenas àqueles que possuem autorização”. Dentro do contexto de saúde, uma quebra de confidencialidade pode ocorrer quando uma “escuta” ou vazamento de informação ocorre entre o agente e o gerenciador.

Integridade significa que um dado não pode ser modificado ou apagado sem autorização. Um vírus, por exemplo, pode quebrar a integridade dos dados no sistema de um agente ou gerenciador. Verificar se os dados vieram do remetente certo e não de alguém que pretende se passar pela fonte ou garantir que os dados originais alcancem o destino sem alterações são também questões inerentes à integridade dos dados.

Disponibilidade, em segurança da informação, significa que a informação deve estar disponível quando necessária. Assim, os sistemas que armazenam, processam e protegem e os canais de comunicação usados para acessar os dados devem estar funcionando corretamente e de forma confiável.

Repúdio, neste contexto, significa ter clareza de como o processo de obtenção dos dados foi realizado. Por exemplo, em um hospital, ter conhecimento da enfermeira que realizou uma medição e qual aparelho foi utilizado, é de suma importância, portanto, todo esse processo deve ser registrado.

Garantir a segurança física dos aparelhos é outro ponto crítico quando se trata de aparelhos para a saúde. Agentes podem ser roubados e trocados por outros, os quais po-

dem fornecer dados errôneos para a decisão médica. Outros problemas, como canal não confinado (conexão sem fio entre agentes e gerenciadores), uso indevido dos gerenciadores (pessoas não autorizadas acessam informações nos *smartphones* ou *desktops*) e a não proteção dos dados nos centros especializados, são outras questões críticas de segurança da informação na saúde.

4.5. Desafios e Perspectivas

As próximas subseções tratam dos principais desafios para o desenvolvimento de WBANs, que estão relacionados, principalmente, a questões de projeto da camada MAC, propagação do sinal, minimização do consumo de energia e coexistência com outras redes operando na mesma faixa de frequência. Ao final, prospecções de estudos futuros são apresentadas.

4.5.1. Desafios Relacionados ao Projeto da Camada MAC

O padrão IEEE 802.15.6 define o mecanismo de acesso ao meio e os requisitos básicos para a interoperabilidade entre dispositivos que operam em WBANs. No entanto, muitos problemas relacionados à camada MAC permanecem ainda sem solução padronizada. Um desses problemas está relacionado à mobilidade do corpo, que pode ocasionar alterações na topologia e densidade da rede. Além disso, o protocolo MAC deve suportar várias redes WBAN operando simultaneamente em diferentes aplicações.

A confiabilidade na entrega dos dados também é um dos requisitos mais importantes. Tendo esta questão em vista, o padrão permite o uso de algumas técnicas para minimizar a interferência como, por exemplo, o salto de frequência dinâmico e a transmissão de *beacon* com intervalos variados conhecidos para cada período. Além disso, permite o uso de retransmissores em cenários nos quais a confiabilidade não é possível através de uma topologia de um salto [Boulis et al. 2012].

Outra preocupação está relacionada à eficiência energética do protocolo MAC. Um dos principais desafios é satisfazer os requisitos de vazão e atraso, que podem variar no tempo, para diferentes tipos de aplicações. Outro desafio é o fornecimento de sincronização do ciclo de trabalho (*duty cycle*) de sensores com diferentes requisitos de tráfego e consumo de energia. A Subseção 4.5.3 abordará essa questão em detalhes.

A provisão de QoS (*Quality of Service*) pela camada MAC também é um dos desafios atuais. O objetivo é permitir que aplicações que trabalham com altas taxas de amostragem transmitam dados conforme necessário ou com um atraso máximo limitado. Desta forma, são necessários mecanismos eficientes de confirmação de entrega (*Acknowledgements*), retransmissão, detecção e correção de erros. Exemplos de aplicações que exigem QoS são as de emergência, que necessitam transmitir dados críticos para a sobrevivência. As soluções de QoS devem ser de baixa complexidade e escaláveis, e também devem prover suporte a diferentes demandas por parte das aplicações. Além disso, devem suportar dispositivos *plug and play* e fornecer conectividade sem interrupção, incluindo a migração entre redes diferentes (*handoff* ou *roaming*). Alguns exemplos de requisitos para aplicações em tempo real, como as de emergência ou de alarmes, são taxas de erro de bit entre 10^{-10} e 10^{-13} e; latência entre $10ms - 250ms$ [Hanson et al. 2009].

4.5.2. Desafios Relacionados ao Meio de Transmissão

Tendo em vista que a comunicação entre dispositivos WBAN deve ocorrer dentro ou nos arredores do corpo humano, um conhecimento profundo sobre estes meios de transmissão e propagação de sinais é necessário para que o desempenho dos protocolos desenvolvidos seja previsto e aprimorado. Os possíveis canais de comunicação podem envolver a transmissão entre os dispositivos localizados dentro, sobre e fora do corpo.

O corpo humano representa um ambiente de propagação complexo para sinais eletromagnéticos por possuir tecidos com diferentes propriedades elétricas e estrutura móvel complexa. Diversos trabalhos encontrados na literatura caracterizam o canal de comunicação WBAN de acordo com diferentes configurações, como a posição dos dispositivos no corpo, a banda de operação da WBAN, e diferentes condições de propagação ocasionadas pela movimentação do corpo e do ambiente ao seu redor.

A caracterização do canal pode ser realizada através do uso de diferentes metodologias. Uma delas é a descrição teórica do fenômeno de propagação eletromagnética do sinal utilizando-se modelos acurados. O objetivo da modelagem é descrever, com detalhes, aspectos específicos da propagação, considerando a influência da estrutura corporal e do padrão de irradiação da antena utilizada.

Em certos casos, como na comunicação sobre o corpo, os objetos presentes no entorno do corpo também deverão ser considerados na modelagem, tendo em vista que o sinal pode ser espalhado e/ou refletido por eles. Desta forma, a caracterização do canal pode ser dividida em duas partes: 1) a parte relativa ao corpo humano em si e; 2) a parte relativa ao ambiente em seu entorno. Entretanto, considerar detalhes profundos do ambiente não é computacionalmente viável. Para superar esta questão, os autores em [Nechayev and Hall 2008] propõem a utilização do modelo de traçado de raios (*ray-tracing model*), que é amplamente utilizado para modelagem de ambientes *indoor*.

A caracterização do canal também pode ser realizada através da obtenção de dados empíricos e tratamento estatístico para elaboração de modelos. Entretanto, para uma caracterização acurada, dados devem ser obtidos considerando-se diferentes fontes de variabilidade do sinal, como diferentes formatos de corpo, ambientes, posição dos dispositivos, e movimentação humana. Os autores em [D'Errico and Ouvry 2010] demonstram que a localização dos nós em conjunto com a movimentação do corpo apresentam impacto forte nas características do canal. Por exemplo, conforme uma pessoa caminha, caso os sensores estejam localizados nos braços, o corpo poderá obstruir a comunicação em determinados momentos devido a sua movimentação, ocasionando uma variação lenta e periódica no sinal.

O trabalho de [Hämäläinen et al. 2011] relata que a propagação também é influenciada pela idade, sexo e implantes de cada pessoa. Além desta variação, existe o desvanecimento rápido (*fast-fading*), que é ocasionado pela propagação multicaminho do sinal difratado e refletido pelo corpo e pelo ambiente. Em [Cavallari et al. 2012], os autores alertam que esta variabilidade do canal afeta fortemente o desempenho de uma rede IEEE 802.15.4. Os resultados obtidos empiricamente foram compatíveis com os obtidos a partir de simulações do canal apresentadas em [Buratti et al. 2011] e [Rosini et al. 2012]. Desta forma, é importante que o desenvolvimento e avaliação de protocolos MAC e de

roteamento sejam realizados considerando-se uma caracterização acurada do canal para que resultados realísticos possam ser obtidos.

O desenvolvimento da antena a ser utilizada também é um desafio devido a certas limitações de formato, tamanho e tipo de material. As restrições de formato e tamanho são relativas ao local do corpo no qual o sensor será instalado ou vestido. Por exemplo, é interessante que o nó possa ser instalado ou substituído em certos locais do corpo sem a necessidade de cirurgias. Desta forma, o caminho percorrido para levar o nó até sua posição dentro do corpo, além de sua posição final, irão ditar suas medidas. Além de medidas limitadas, o material utilizado para a confecção do dispositivo deverá ser biocompatível e não corrosivo, como o titânio ou a platina. Antenas construídas com esses materiais apresentam padrões de irradiação diferentes das construídas comumente com cobre. Os dispositivos também deverão ser resistentes aos padrões de movimentação do corpo para que não sejam danificados.

4.5.3. Desafios Relacionados ao Consumo de Energia

Os dispositivos WBAN poderão ser instalados dentro do corpo, tornando-se de difícil acesso para a realização de processos de manutenção, como a troca ou recarga da bateria. Tendo em vista que a manutenção pode exigir a realização de cirurgia, é interessante que seja realizada com baixa frequência. Portanto, é necessário que a vida útil da bateria de um dispositivo WBAN seja a mais longa possível.

Conforme relata a literatura, diversas são as fontes que contribuem para o consumo ineficiente da energia. Algumas delas são: colisões, recebimento de quadros que não são úteis para o nó em questão e aguardar pelo recebimento de quadros com o rádio ligado quando o canal está ocioso. Estas fontes são comuns quando o mecanismo de controle de acesso ao meio utilizado é baseado em contenção, como o CSMA/CA. Porém, podem ser desconsideradas caso opte-se pela utilização de gerenciamento centralizado para envio de tráfego, como o TDMA. Entretanto, este último possui a desvantagem de necessitar de sincronização periódica entre nós, o que novamente ocasiona consumo energético.

Os padrões IEEE 802.15.6 e 802.15.4 apresentam diferentes mecanismos de acesso ao meio e, portanto, o consumo de energia também é diferenciado. Devido ao fato de o sensoriamento no 802.15.6 ser sempre realizado antes de o contador de *backoff* ser decrementado, este padrão apresenta maior consumo energético em relação ao 802.15.4, no qual a fase de sensoriamento dura apenas dois períodos de *backoff* quando o contador chega ao valor zero. Entretanto, no padrão mais recente, a probabilidade de sucesso de transmissão de um pacote é maior em relação à encontrada no padrão anterior porque os dispositivos possuem um conhecimento mais profundo sobre o estado do canal. Em termos de consumo de energia, o padrão IEEE802.15.6 *Slotted Aloha* é considerado preferível porque não utiliza fase de escuta ao meio. Entretanto, apresenta maior probabilidade de colisão. Este compromisso deve ser avaliado e levado em consideração no decorrer do projeto da camada MAC [Cavallari et al. 2014].

O problema do consumo de energia pode ser superado através do desenvolvimento de camadas PHY e MAC energeticamente eficientes. Uma das abordagens para o problema é reduzir o ciclo de trabalho (*duty cycle*) do rádio. Uma solução estudada é a utilização do *Low Power Listening* (LPL), também conhecido por *Preamble Sampling*. Nesta

técnica, os dispositivos economizam energia ao alternar o estado do rádio entre ligado e desligado. Os nós que não desejam transmitir dados devem acordar de forma assíncrona com determinada periodicidade para verificar se existem dados sendo transmitidos para ele. Caso não exista, eles voltam a dormir. Os nós que desejam transmitir, devem verificar se o meio está vazio e a seguir enviar um preâmbulo. Foi verificado [Buettner et al. 2006] que o envio de diversos preâmbulos pequenos gera maior economia de energia em relação ao envio de apenas um preâmbulo longo. Entre cada preâmbulo curto, existe um intervalo para que ACKs sejam transmitidos. Estes preâmbulos são enviados por, pelo menos, um ciclo de sono para que todos os nós possam acordar e receber pelo menos um deles. Após o envio dos preâmbulos, os dados são enviados. Apenas os nós listados como destino e que receberem o preâmbulo deverão permanecer acordados para receber os dados. Em [Van Dam and Langendoen 2003], os autores conseguiram uma economia de energia de até 98% utilizando esta técnica para aplicações que não demandam atrasos baixos nem altas taxas de tráfego como, por exemplo, alarmes de emergência. Já para aplicações com demanda de menor atraso, como *streaming* de áudio, a economia de energia decai significativamente e a sobrecarga aumenta devido ao envio de preâmbulos.

Outra opção para a otimização do consumo de energia é o uso da técnica de codificação de rede [Movassaghi et al. 2013c], na qual as mensagens que chegam em um retransmissor são combinadas em uma única mensagem enviada em um único *slot* de tempo, reduzindo o número de transmissões por nó.

4.5.4. Desafios Relacionados à Coexistência de WBANS

Uma das bandas de frequências nas quais WBANs operam é a banda industrial, científica e médica (ISM, do inglês *Industrial, Scientific and Medical*). Esta banda é compartilhada por várias tecnologias, como IEEE 802.11 (Wi-Fi), IEEE 802.15.1 (*Bluetooth*) e IEEE 802.15.4 (*ZigBee*). Por coexistirem na mesma faixa de frequência, essas tecnologias podem causar interferência entre si, afetando as WBANs. Por isso, é importante que soluções que garantam a coexistência das WBANs com estas outras tecnologias sejam utilizadas. Este problema necessita de investigação mais profunda pela comunidade científica, tendo em vista que os padrões citados não incorporaram soluções específicamente voltadas para tal. Redes como as IEEE 802.11, por operarem com potência mais elevada em relação a WBANs e serem amplamente empregadas, devem ser foco de avaliação da coexistência.

Trabalhos como o de [Francisco et al. 2009] e [Chen and Pomalaza-Ráez 2009] relatam o aumento da taxa de perda de pacotes em redes IEEE 802.15.4 na presença de redes IEEE 802.11. Os ambientes avaliados experimentalmente incluem um quarto em um hospital e outro em um apartamento. Em [Huo et al. 2009], os autores verificam o impacto do uso de um forno de microondas e concluem que a interferência é negligenciável para distâncias superiores a 2 metros.

Em [Hernandez and Miura 2012] os autores avaliam a coexistência de WBANs IEEE 802.15.6 e IEEE 802.15.4a em UWB e mostram que a taxa de erro de bits de uma rede 802.15.6 não é afetada pela coexistência com outra rede da mesma tecnologia, ou uma rede 802.15.4a, caso o nível de sinal do tráfego desejado esteja acima de -30 dBm. Em caso contrário, uma degradação severa da qualidade da comunicação pode ser notada.

Em [Torabi and Leung 2012], é apresentada uma proposta com abordagem baseada na centralização do mecanismo de acesso ao meio que emprega sensoriamento cognitivo do espectro. Resultados mostram uma vazão quatro vezes melhor para redes 802.15.4 na presença de redes 802.11. O padrão Bluetooth LE é tido como o mais cooperativo na banda ISM de 2.4GHz por utilizar a técnica de salto em frequências. Esta técnica foi adotada também em novas versões dos padrões 802.15.4 (a 802.15.4e) para aprimorar a robustez na rede na presença de interferência.

4.5.4.1. Rádio Cognitivo em WBANs

A tecnologia de rádio cognitivo (CR, do inglês *Cognitive Radio*) pode ser implementada em WBANs para reduzir a interferência causada pela coexistência, além de ocasionar o uso mais flexível e eficiente do espectro radioelétrico, já que permite que usuários não licenciados (SU, do inglês *Secondary Users*) acessem o espectro de rádio desde que não ocasionem interferência prejudicial aos usuários licenciados (PU, do inglês *Primary Users*). Isso é possível porque rádios cognitivos, inteligentemente, adaptam seus parâmetros de rádio de acordo com critérios de aprendizado predefinidos, selecionando a melhor frequência para operação e as configurações de transmissão. As principais funções de CR, que são realizadas pelas camadas PHY e MAC, são detecção de espectro, acesso ao espectro e compartilhamento de espectro.

As WBANs habilitadas com CR, conhecidas como CRWBAN (do inglês *Cognitive Radio Wirelees Body Area Network*), apresentam os seguintes desafios: (1) Acesso ao espectro buscando evitar que os SUs ocasionem colisões e interferência aos PUs. (2) Eficiência energética buscando reduzir o desperdício de energia ocasionado, principalmente, por colisões, *overhearing*, sobrecarga de pacotes, flutuação de tráfego e escuta ociosa. (3) Detecção oportunista por parte dos SUs, que deverão supervisionar o espectro sempre que possível, mantendo uma lista de canais vazios. (4) Decisão otimizada de uso do espectro, que deverá ser tomada a partir de dados obtidos utilizando-se tempo e energia mínimos. Várias técnicas e métodos podem ser aplicados para os processos de aprendizagem e tomada de decisão, dentre eles: a computação evolutiva; a lógica difusa; e o processo de decisão de Markov. (5) Desenho *Cross-Layer*, onde a detecção do espectro é realizada pelas camadas PHY e MAC, mas a gestão do espectro (como a tomada de decisão do espectro, o agendamento e a alocação do canal) pode estar relacionada a todas as outras camadas de rede. Portanto, diferentes camadas da pilha de protocolos devem ser coordenadas [Bhandari and Moh 2015].

Outros desafios relacionados à implementação de CR em WBAN, que também estão estritamente relacionados aos desafios próprios de WBAN são segurança, privacidade, consumo de energia e QoS. Segundo [Bhandari and Moh 2015], os desafios mais importantes a serem abordados são: (1) Implementação de CR - Implementar capacidades cognitivas (aprender, detectar e adaptar) nos nós sensores vai incorrer em maior consumo de energia. Também, os usuários secundários devem alternar suas respectivas frequências de canais e potências de transmissão de acordo com o tipo de rede operacional. É necessário se determinar um ponto adequado na relação custo-benefício na implementação CRWBAN. É viável uma arquitetura na qual a CR esteja implementada somente no nó co-

letor (*sink*), que apresenta menos restrições energéticas. (2) Consumo de energia - Como já é conhecido que os sensores CRWBAN são dispositivos com restrição de energia, além da energia necessária para operações de rede normais (como descoberta de rotas, transmissão/recepção de pacotes de dados e processamento de dados), esses sensores também precisam de energia para detecção de espectro, negociação de canal e frequente transferência de espectro. (3) Melhorar a QoS - O suporte à QoS é uma questão desafiadora devido às restrições de recursos das CRWBAN. Embora os requisitos de QoS variem de acordo com a aplicação e o ambiente operacional, é necessário garantir sempre. Outro desafio em CRWBAN é proteger os direitos dos PUs, pois a comunicação dos PUs deve ser livre de interferências de SUs.

4.5.4.2. Protocolos MAC para CRWBAN

Na literatura, podem ser encontradas várias propostas de protocolos MAC para CRWBANs. O trabalho de [Bhandari and Moh 2015] apresenta alguns protocolos que consideram o uso de CR em WBAN.

No protocolo CR-MAC apresentado em [Ali et al. 2010], as informações de saúde são classificadas em críticas, moderadamente urgentes e não críticas, para isso utilizam-se respectivamente três níveis de potência de transmissão P1, P2 e P3, onde P1 < P2 < P3. O tráfego crítico tem a maior prioridade, e o tráfego moderadamente urgente tem o segundo maior. CR-MAC tem a vantagem de que garanta QoS para cada tipo de informação. Além disso, oferece alto rendimento, baixa taxa de colisão e baixo consumo energético.

No protocolo DCAA-MAC (*Dynamic Channel Adjustable Asynchronous Cognitive Radio MAC*) [Lee et al. 2011], cada nó varre e seleciona um canal com as melhores condições, por exemplo, baixa relação sinal-ruído (SNR - *Signal-to-Noise Ratio*). Depois cada nó na rede vai dormir e acorda periodicamente e de forma independente. Um nó primeiro envia um preâmbulo; então, depois de receber a mensagem de confirmação (ACK - *Acknowledgement*) do nó de destino, ele envia os pacotes de dados. No lado do receptor, o receptor detecta o preâmbulo e permanece despertado para receber os dados. Uma vez que a transmissão esteja completa, os nós remetente e receptor passam para o modo de suspensão. Se o nó detecta interferência (ruído ou um PU) em seu canal, ele muda para outro canal disponível para comunicação efetiva. Este protocolo tem uma capacidade de comutação de canal rápida, graças a isso, oferece baixa latência, eficiência energética, e não precisa de sincronização das WBANs. Além disso, provê proteção de PUs e QoS.

C-RICER (*Cognitive-Receiver Initiated Cycled Receiver*) [Nhan et al. 2014] é um protocolo projetado para WBAN em ambientes de alta interferência. O C-RICER ajusta a frequência do canal e a potência de transmissão para reduzir a interferência e o consumo de energia. O objetivo principal da aplicação de CR é manter o SNR para troca de dados e superar as desvantagens da alta interferência coexistente em WBAN. Para reduzir a energia de detecção, o coordenador do C-RICER detecta periodicamente interferência apenas no canal de trabalho. Com este protocolo, alcança-se baixa taxa de colisão e baixo consumo energético.

4.5.5. Perspectivas para o Futuro

Com relação ao problema de modelagem de canal nas WBANs, há uma necessidade de estudos mais aprofundados envolvendo cenários fora do corpo e B2B (*Body-to-Body*), que não possuem um modelo amplamente aceito. No cenário B2B, a comunicação ocorre entre os nós do corpo localizados em diferentes sujeitos humanos. Dada a dificuldade de se realizar testes experimentais comparativos, a criação de um ambiente de testes aberto à comunidade científica é essencial para que as soluções sejam testadas e comparadas.

Em relação ao uso de antenas, para melhor avaliação da propagação do sinal, trabalhos futuros devem fornecer resultados incluindo e excluindo os efeitos dos padrões de radiação da antena utilizada.

Em relação à topologia da rede, considerando-se a tendência para a criação de SoCs (*System on a Chip*) cada vez menores e mais eficientes em termos energéticos, espera-se que as WBANs sejam compostas por centenas de nós. Neste cenário, o uso de topologias com múltiplos saltos torna-se interessante, trazendo a necessidade do desenvolvimento de protocolos de roteamento específicos para essas redes.

Em relação à propagação dentro e fora do corpo humano, o uso de redes híbridas moleculares/RF [Akyildiz et al. 2008] é promissor para o futuro das WBANs. Apesar de a tecnologia estar ainda em um estágio inicial de desenvolvimento, por ela apresentar muitos benefícios, espera-se que seja desenvolvida rapidamente.

4.6. Considerações Finais

WBAN é uma tecnologia promissora que permite o desenvolvimento de novas aplicações relacionadas à saúde podendo mudar nosso estilo de vida em um futuro próximo. Desenvolvimento e padronização nessa área devem ser guiados por pesquisas sólidas, uma vez que esse tipo de rede interage diretamente com o corpo humano.

Este trabalho apresentou uma visão geral de WBANs, discutindo diversos exemplos de sensores e atuadores que podem se comportar como nós da rede. Os requisitos de comunicação em termos de taxa de bits, atraso e taxa de erros foram apresentados. Além disso, questões importantes relacionadas à saúde humana e segurança foram comentadas.

Padrões e soluções para comunicação em WBANs foram endereçados, realçando uma série de propostas encontradas na literatura, incluindo o padrão emergente IEEE 802.15.6. A família de normas IEEE 11073 também foi apresentada, pois especifica padrões para o desenvolvimento de dispositivos pessoais de saúde, que devem ser interoperáveis com sensores e atuadores que compõem as WBANs.

Desafios de pesquisa e perspectivas futuras também foram apontadas, mostrando que WBAN é um tópico promissor para trabalhos futuros na área de computação e telecomunicações.

Referências

[Akyildiz et al. 2008] Akyildiz, I. F., Brunetti, F., and Blázquez, C. (2008). Nanonetworks: A new communication paradigm. *Computer Networks*, 52(12):2260 – 2279.

[Al Ameen et al. 2011] Al Ameen, M., Liu, J., Ullah, S., and Kwak, K. S. (2011). A power efficient mac protocol for implant device communication in wireless body area networks. In *IEEE Consumer Communications and Networking Conference (CCNC)*, pages 1155–1160. IEEE.

[Al-Fuqaha et al. 2015] Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., and Ayyash, M. (2015). Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials*, 17(4):2347–2376.

[Ali et al. 2010] Ali, K. A., Sarker, J. H., and Mouftah, H. T. (2010). A mac protocol for cognitive wireless body area sensor networking. In *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, pages 168–172. ACM.

[Allen et al. 2005] Allen, B., Brown, A., Schwieger, K., Zimmermann, E., Malik, W. Q., Edwards, D. J., Ouvry, L., and Oppermann, I. (2005). Ultra wideband: Applications, technology and future perspectives. In *International Workshop on Convergent Technologies (IWCT)*.

[Asogwa et al. 2012] Asogwa, C. O., Zhang, X., Xiao, D., and Hamed, A. (2012). Experimental analysis of AODV, DSR and DSDV protocols based on wireless body area network. In *Internet of Things*, pages 183–191. Springer.

[Baan et al. 2011] Baan, R., Grosse, Y., Lauby-Secretan, B., El Ghissassi, F., Bouvard, V., Benbrahim-Tallaa, L., Guha, N., Islami, F., Galichet, L., and Straif, K. (2011). Carcinogenicity of radiofrequency electromagnetic fields. *The Lancet Oncology*, 12(7):624–626.

[Bae et al. 2012] Bae, J., Cho, H., Song, K., Lee, H., and Yoo, H.-J. (2012). The signal transmission mechanism on the surface of human body for body channel communication. *IEEE Transactions on Microwave Theory and Techniques*, 60(3):582–593.

[Bag and Bassiouni 2007] Bag, A. and Bassiouni, M. A. (2007). Hotspot preventing routing algorithm for delay-sensitive biomedical sensor networks. In *IEEE International Conference on Portable Information Devices*, pages 1–5. IEEE.

[Bhandari and Moh 2015] Bhandari, S. and Moh, S. (2015). A survey of MAC protocols for cognitive radio body area networks. *Sensors*, 15(4):9189–9209.

[Bhandari and Moh 2016] Bhandari, S. and Moh, S. (2016). A priority-based adaptive MAC protocol for wireless body area networks. *Sensors*, 16(3):401.

[Bhanumathi and Sangeetha 2017] Bhanumathi, V. and Sangeetha, C. (2017). A guide for the selection of routing protocols in wban for healthcare applications. *Human-centric Computing and Information Sciences*, 7(1):24.

[Bianchi 2000] Bianchi, G. (2000). Performance analysis of the ieee 802.11 distributed coordination function. *IEEE Journal on Selected Areas in Communications*, 18(3):535–547.

[Boulis et al. 2011] Boulis, A. et al. (2011). Castalia: A simulator for wireless sensor networks and body area networks. *NICTA: National ICT Australia*.

[Boulis et al. 2012] Boulis, A., Smith, D., Miniutti, D., Libman, L., and Tselishchev, Y. (2012). Challenges in body area networks for healthcare: the mac. *IEEE Communications Magazine*, 50(5):100–106.

[Buettner et al. 2006] Buettner, M., Yee, G. V., Anderson, E., and Han, R. (2006). X-mac: A short preamble mac protocol for duty-cycled wireless sensor networks. In *International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 307–320.

[Buratti et al. 2011] Buratti, C., D'Errico, R., Maman, M., Martelli, F., Rosini, R., and Verdone, R. (2011). Design of a body area network for medical applications: the wiserban project. In *Proceedings of the 4th International Symposium on Applied Sciences in Biomedical and Communication Technologies*, page 164. ACM.

[Cavallari et al. 2012] Cavallari, R., Guidotti, E., Buratti, C., and Verdone, R. (2012). Experimental characterisation of data aggregation in BANs with a walking subject. In *Proceedings of the 7th International Conference on Body Area Networks*, pages 191–194. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Cavallari et al. 2014] Cavallari, R., Martelli, F., Rosini, R., Buratti, C., and Verdone, R. (2014). A survey on wireless body area networks: Technologies and design challenges. *IEEE Communications Surveys & Tutorials*, 16(3):1635–1657.

[Chen and Pomalaza-Ráez 2009] Chen, C. and Pomalaza-Ráez, C. (2009). Design and evaluation of a wireless body sensor system for smart home health monitoring. In *IEEE Global Telecommunications Conference, (GLOBECOM)*, pages 1–6. IEEE.

[Cheshire and Krochmal 2013] Cheshire, S. and Krochmal, M. (2013). DNS-based service discovery. Technical report.

[Culpepper et al. 2004] Culpepper, B. J., Dung, L., and Moh, M. (2004). Design and analysis of hybrid indirect transmissions (hit) for data gathering in wireless micro sensor networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, 8(1):61–83.

[David et al. 2013] David, L., Vasconcelos, R., Alves, L., André, R., and Endler, M. (2013). A DDS-based middleware for scalable tracking, communication and collaboration of mobile nodes. *Journal of Internet Services and Applications*, 4(1):1.

[D'Errico and Ouvry 2010] D'Errico, R. and Ouvry, L. (2010). A statistical model for on-body dynamic channels. *International Journal of Wireless Information Networks*, 17(3-4):92–104.

[Dixon et al. 2012] Dixon, A. M., Allstot, E. G., Gangopadhyay, D., and Allstot, D. J. (2012). Compressed sensing system considerations for ecg and emg wireless biosensors. *IEEE Transactions on Biomedical Circuits and Systems*, 6(2):156–166.

[Donoho 2006] Donoho, D. L. (2006). Compressed sensing. *IEEE Transactions on Information Theory*, 52(4):1289–1306.

[Effatparvar et al. 2016] Effatparvar, M., Dehghan, M., and Rahmani, A. M. (2016). A comprehensive survey of energy-aware routing protocols in wireless body area sensor networks. *Journal of Medical Systems*, 40(9):201.

[Ferreira et al. 2017a] Ferreira, V. C., Muchaluat-Saade, D. C., and Albuquerque, C. V. (2017a). Estudo sobre estabilidade de rotas em redes corporais sem fio. In *Escola Regional de Computação Aplicada à Saúde (ERCAS)*.

[Ferreira et al. 2017b] Ferreira, V. C., Seixas, F. L., Muchaluat-Saade, D. C., and Albuquerque, C. V. N. (2017b). Análise do protocolo AODV para roteamento em wireless body area networks. In *VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*.

[Francisco et al. 2009] Francisco, R., Huang, L., and Dolmans, G. (2009). Coexistence of wban and wlan in medical environments. In *IEEE 70th Vehicular Technology Conference Fall (VTC'09)*, pages 1–5. IEEE.

[Galluccio et al. 2012] Galluccio, L., Melodia, T., Palazzo, S., and Santagati, G. E. (2012). Challenges and implications of using ultrasonic communications in intra-body area networks. In *9th Annual Conference on Wireless On-demand Network Systems and Services (WONS)*, pages 182–189. IEEE.

[Hämäläinen et al. 2011] Hämäläinen, M., Taparugssanagorn, A., and Iinatti, J. (2011). On the wban radio channel modelling for medical applications. In *Proceedings of the 5th European Conference on Antennas and Propagation (EUCAP)*, pages 2967–2971. IEEE.

[Hanson et al. 2009] Hanson, M. A., Powell Jr, H. C., Barth, A. T., Ringgenberg, K., Calhoun, B. H., Aylor, J. H., and Lach, J. (2009). Body area sensor networks: Challenges and opportunities. *IEEE Computer*, 42(1).

[Hao and Foster 2008] Hao, Y. and Foster, R. (2008). Wireless body sensor networks for health-monitoring applications. *Physiological Measurement*, 29(11):R27.

[Hayajneh et al. 2014] Hayajneh, T., Almashaqbeh, G., Ullah, S., and Vasilakos, A. V. (2014). A survey of wireless technologies coexistence in wban: analysis and open research issues. *Wireless Networks*, 20(8):2165–2199.

[He et al. 2015] He, P., Li, X., Yan, L., Yang, S., and Zhang, B. (2015). Performance analysis of WBAN based on AODV and DSDV routing protocols. In *2nd International Symposium on Future Information and Communication Technologies for Ubiquitous HealthCare (Ubi-HealthTech)*, pages 1–4. IEEE.

[Heaney et al. 2011] Heaney, S. F., Scanlon, W. G., Garcia-Palacios, E., Cotton, S. L., and McKernan, A. (2011). Characterization of inter-body interference in context aware body area networking (caban). In *IEEE GLOBECOM*, pages 586–590. IEEE.

[Heinzelman et al. 2000] Heinzelman, W. R., Chandrakasan, A., and Balakrishnan, H. (2000). Energy-efficient communication protocol for wireless microsensor networks. In *33rd Annual Hawaii International Conference on System Sciences*, pages 10–pp. IEEE.

[Hernandez and Miura 2012] Hernandez, M. and Miura, R. (2012). Coexistence of ieee std 802.15.6tm-2012 uwb-phy with other uwb systems. In *IEEE International Conference on Ultra-Wideband*, pages 46–50.

[Hoang et al. 2009] Hoang, D. C., Tan, Y. K., Chng, H. B., and Panda, S. K. (2009). Thermal energy harvesting from human warmth for wireless body area network in medical healthcare system. In *International Conference on Power Electronics and Drive Systems (PEDS)*, pages 1277–1282. IEEE.

[Hunkeler et al. 2008] Hunkeler, U., Truong, H. L., and Stanford-Clark, A. (2008). MQTT-S: A publish/subscribe protocol for Wireless Sensor Networks. In *3rd international conference on communication systems software and middleware and workshops (Comsware 2008)*, pages 791–798. IEEE.

[Huo et al. 2009] Huo, H., Xu, Y., Bilen, C. C., and Zhang, H. (2009). Coexistence issues of 2.4 ghz sensor networks with other rf devices at home. In *Third International Conference on Sensor Technologies and Applications (SENSORCOMM'09)*, pages 200–205. IEEE.

[ICNIRP 2009] ICNIRP (2009). International commission on non-ionizing radiation protection (ICNIRP) statement on the guidelines for limiting exposure to time-varying electric, magnetic, and electromagnetic fields (up to 300 ghz). *Health Physics*, 97(3):257–258.

[IEEE Std 11073-00103 2012] IEEE Std 11073-00103 (2012). Health informatics - personal health device communication: Overview. Standard, ISO/IEEE, New York, USA.

[IEEE Std 11073-20101 2004] IEEE Std 11073-20101 (2004). Health informatics-point-of-care medical device communication - part 20101: Application profile-base standard. Standard, ISO/IEEE, New York, USA.

[IEEE Std 11073-20601 2010] IEEE Std 11073-20601 (2010). Health informatics - personal health device communication - application profile: Optimized exchange protocol. Standard, ISO/IEEE, New York, USA.

[IEEE Std 802.15.4 2006a] IEEE Std 802.15.4 (2006a). Ieee standard for information technology– local and metropolitan area networks– specific requirements– part 15.4: Wireless medium access control (mac) and physical layer (phy) specifications for low rate wireless personal area networks (wpans). *IEEE Std 802.15.4-2006 (Revision of IEEE Std 802.15.4-2003)*, pages 1–320.

[IEEE Std 802.15.4 2006b] IEEE Std 802.15.4 (2006b). Ieee standard for local and metropolitan area networks - part 15.4: Wireless medium access control (mac) and physical layer (phy) specifications for low rate wireless personal area networks (wpans). Standard, Institute of Electrical and Electronic Engineers.

[IEEE Std 802.15.6 2012] IEEE Std 802.15.6 (2012). IEEE Standard for Local and metropolitan area networks - Part 15.6: Wireless Body Area Networks. Standard, Institute of Electrical and Electronic Engineers, New York, USA.

[Javaid et al. 2013] Javaid, N., Hayat, S., Shakir, M., Khan, M. A., Bouk, S. H., and Khan, Z. (2013). Energy efficient mac protocols in wireless body area sensor networks-a survey. *arXiv preprint arXiv:1303.2072*.

[Johnson et al. 2007] Johnson, D., Hu, Y., and Maltz, D. (2007). RFC 4728: The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4, feb. 2007. Technical report.

[Khan et al. 2012a] Khan, Z., Aslam, N., Sivakumar, S., and Phillips, W. (2012a). Energy-aware peering routing protocol for indoor hospital body area network communication. *Procedia Computer Science*, 10:188–196.

[Khan et al. 2012b] Khan, Z., Sivakumar, S., Phillips, W., and Robertson, B. (2012b). Qprd: Qos-aware peering routing protocol for delay sensitive data in hospital body area network communication. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2012 Seventh International Conference on*, pages 178–185. IEEE.

[Khan et al. 2013] Khan, Z. A., Sivakumar, S., Phillips, W., and Robertson, B. (2013). A qos-aware routing protocol for reliability sensitive data in hospital body area networks. *Procedia Computer Science*, 19:171–179.

[Kirti 2016] Kirti, T. M. (2016). Survey of IEEE 802.15 task work group. *International Journal of Engineering Science*, 2077.

[Kumari and Nand 2016] Kumari, R. and Nand, P. (2016). Performance comparison of various routing protocols in wsn and wban. In *International Conference on Computing, Communication and Automation (ICCCA)*, pages 427–431. IEEE.

[Latré et al. 2011] Latré, B., Braem, B., Moerman, I., Blondia, C., and Demeester, P. (2011). A survey on wireless body area networks. *Wireless Networks*, 17(1):1–18.

[Latré et al. 2007] Latré, B., Braem, B., Moerman, I., Blondia, C., Reusens, E., Joseph, W., and Demeester, P. (2007). A low-delay protocol for multihop wireless body area networks. In *Mobile and Ubiquitous Systems: Networking & Services, 2007. MobiQuitous 2007. Fourth Annual International Conference on*, pages 1–8. IEEE.

[Lee et al. 2011] Lee, B., Yun, J., and Han, K. (2011). Dynamic channel adjustable asynchronous cognitive radio mac protocol for wireless medical body area sensor networks. In *Communication and Networking*, pages 338–345. Springer.

[Levis et al. 2005] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., et al. (2005). Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer.

[Li and Tan 2010] Li, H. and Tan, J. (2010). Heartbeat-driven medium-access control for body sensor networks. *IEEE Transactions on Information Technology in Biomedicine*, 14(1):44–51.

[Lindsey and Raghavendra 2002] Lindsey, S. and Raghavendra, C. S. (2002). Pegasis: Power-efficient gathering in sensor information systems. In *IEEE Aerospace Conference, 2002*, volume 3, pages 3–3. IEEE.

[Marinkovic et al. 2009a] Marinkovic, S., Spagnol, C., and Popovici, E. (2009a). Energy-efficient tdma-based mac protocol for wireless body area networks. In *Third International Conference on Sensor Technologies and Applications (SENSORCOMM'09)*, pages 604–609. IEEE.

[Marinkovic et al. 2009b] Marinkovic, S. J., Popovici, E. M., Spagnol, C., Faul, S., and Marnane, W. P. (2009b). Energy-efficient low duty cycle mac protocol for wireless body area networks. *IEEE Transactions on Information Technology in Biomedicine*, 13(6):915–925.

[Martins et al. 2014] Martins, A. F., Santos, D. F., Perkusich, A., and Almeida, H. O. (2014). Upnp and ieee 11073: Integrating personal health devices in home networks. *2014 IEEE 11th Consumer Communications and Networking Conference, CCNC 2014*, pages 1–6.

[Maskooki et al. 2011] Maskooki, A., Soh, C. B., Gunawan, E., and Low, K. S. (2011). Opportunistic routing for body area network. In *Consumer Communications and Networking Conference (CCNC), 2011 IEEE*, pages 237–241. IEEE.

[Monowar and Bajaber 2015] Monowar, M. M. and Bajaber, F. (2015). On designing thermal-aware localized qos routing protocol for in-vivo sensor nodes in wireless body area networks. *Sensors*, 15(6):14016–14044.

[Movassaghi et al. 2012] Movassaghi, S., Abolhasan, M., and Lipman, J. (2012). Energy efficient thermal and power aware (etpa) routing in body area networks. In *23rd International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC)*, pages 1108–1113. IEEE.

[Movassaghi et al. 2013a] Movassaghi, S., Abolhasan, M., and Lipman, J. (2013a). A review of routing protocols in wireless body area networks. *Journal of Networks*, 8(3):559–575.

[Movassaghi et al. 2013b] Movassaghi, S., Abolhasan, M., and Lipman, J. (2013b). A review of routing protocols in wireless body area networks. *Journal of Networks*, 8(3):559–575.

[Movassaghi et al. 2014] Movassaghi, S., Abolhasan, M., Lipman, J., Smith, D., and Jamalipour, A. (2014). Wireless body area networks: A survey. *IEEE Communications Surveys & Tutorials*, 16(3):1658–1686.

[Movassaghi et al. 2013c] Movassaghi, S., Shirvanimoghaddam, M., Abolhasan, M., and Smith, D. (2013c). An energy efficient network coding approach for wireless body area networks. In *IEEE 38th Conference on Local Computer Networks (LCN)*, pages 468–475. IEEE.

[Nechayev and Hall 2008] Nechayev, Y. I. and Hall, P. S. (2008). Multipath fading of on-body propagation channels. In *IEEE Antennas and Propagation Society International Symposium*, pages 1–4.

[Nhan et al. 2014] Nhan, N.-Q., Gautier, M., and Berder, O. (2014). Asynchronous mac protocol for spectrum agility in wireless body area sensor networks. In *9th International Conference on Cognitive Radio Oriented Wireless Networks and Communications (CROWNCOM)*, pages 203–208. IEEE.

[Oey and Moh 2013] Oey, C. H. W. and Moh, S. (2013). A survey on temperature-aware routing protocols in wireless body sensor networks. *Sensors*, 13(8):9860–9877.

[Omeni et al. 2008] Omeni, O., Wong, A. C. W., Burdett, A. J., and Toumazou, C. (2008). Energy efficient medium access protocol for wireless medical body area sensor networks. *IEEE Transactions on Biomedical Circuits and Systems*, 2(4):251–259.

[Pantelopoulos and Bourbakis 2010] Pantelopoulos, A. and Bourbakis, N. G. (2010). A survey on wearable sensor-based systems for health monitoring and prognosis. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 40(1):1–12.

[PCHAlliance 2017] PCHAlliance (2017). Introduction to the continua design guidelines. Standard, Personal Connected Health Alliance.

[Perkins et al. 2003] Perkins, C., Belding-Royer, E., and Das, S. (2003). Ad hoc on-demand distance vector (AODV) routing. Technical report.

[Perkins and Bhagwat 1994] Perkins, C. E. and Bhagwat, P. (1994). Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *ACM SIGCOMM computer communication review*, volume 24, pages 234–244. ACM.

[Quwaider and Biswas 2009] Quwaider, M. and Biswas, S. (2009). Probabilistic routing in on-body sensor networks with postural disconnections. In *Proceedings of the 7th ACM international symposium on Mobility management and wireless access*, pages 149–158. ACM.

[Rafatkhah and Lighvan 2014] Rafatkhah, O. and Lighvan, M. Z. (2014). M2e2: a novel multi-hop routing protocol for wireless body sensor networks. *Int J Comput Netw Commun Secur*, 2(8):260–267.

[Razzaque et al. 2011] Razzaque, M. A., Hong, C. S., and Lee, S. (2011). Data-centric multiobjective qos-aware routing protocol for body sensor networks. *Sensors*, 11(1):917–937.

[Rhee et al. 2004] Rhee, S. H., Chung, K., Kim, Y., Yoon, W., and Chang, K. S. (2004). An application-aware mac scheme for ieee 802.15. 3 high-rate wpan. In *IEEE Wireless Communications and Networking Conference (WCNC)*, volume 2, pages 1018–1023. IEEE.

[Rodgers et al. 2015] Rodgers, M. M., Pai, V. M., and Conroy, R. S. (2015). Recent advances in wearable sensors for health monitoring. *Sensors*, 15(6):3119–3126.

[Rosini et al. 2012] Rosini, R., Martelli, F., Maman, M., D'Errico, R., Buratti, C., and Verdone, R. (2012). On-body area networks: from channel measurements to mac layer performance evaluation. In *European Wireless Conference*, pages 1–7.

[Ruzzelli et al. 2007] Ruzzelli, A. G., Jurdak, R., O'Hare, G. M., and Van Der Stok, P. (2007). Energy-efficient multi-hop medical sensor networking. In *Proceedings of the 1st ACM SIGMOBILE international workshop on Systems and networking support for healthcare and assisted living environments*, pages 37–42. ACM.

[Savci et al. 2005] Savci, H. S., Sula, A., Wang, Z., Dogan, N. S., and Arvas, E. (2005). Mics transceivers: regulatory standards and applications [medical implant communications service]. In *IEEE SoutheastCon*, pages 179–182. IEEE.

[Seyedi et al. 2013] Seyedi, M., Kibret, B., Lai, D. T., and Faulkner, M. (2013). A survey on intrabody communications for body area network applications. *IEEE Transactions on Biomedical Engineering*, 60(8):2067–2079.

[SIG 2010] SIG, B. (2010). Specification of the bluetooth system version 4.0. [Online] Available at: http://www.bluetooth.com.

[Sruthi 2016] Sruthi, R. (2016). Medium access control protocols for wireless body area networks: A survey. *Procedia Technology*, 25:621–628.

[Su and Zhang 2009] Su, H. and Zhang, X. (2009). Battery-dynamics driven tdma mac protocols for wireless body-area monitoring networks in healthcare applications. *IEEE Journal on Selected Areas in Communications*, 27(4).

[Tiwari et al. 2014] Tiwari, R., Shrivastava, S., and Das, S. (2014). Performance analysis of mobile patient network using aodv and dsr routing algorithms. In *Green Computing Communication and Electrical Engineering (ICGCCEE), 2014 International Conference on*, pages 1–6. IEEE.

[Torabi and Leung 2012] Torabi, N. and Leung, V. (2012). Robust access for wireless body area networks in public m-health. In *Proceedings of the 7th International Conference on Body Area Networks*, pages 170–176. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Torres et al. 2016] Torres, A. B., Rocha, A. R., and de Souza, J. N. (2016). Análise de desempenho de brokers mqtt em sistema de baixo custo. In *Anais do XXXVI Congresso da Sociedade Brasileira de Computação*.

[Tselishchev et al. 2010] Tselishchev, Y., Boulis, A., and Libman, L. (2010). Experiences and lessons from implementing a wireless sensor network mac protocol in the castalia simulator. In *IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE.

[Ullah et al. 2012] Ullah, S., Higgins, H., Braem, B., Latre, B., Blondia, C., Moerman, I., Saleem, S., Rahman, Z., and Kwak, K. S. (2012). A comprehensive survey of wireless body area networks. *Journal of Medical Systems*, 36(3):1065–1094.

[Ullah et al. 2014] Ullah, S., Imran, M., and Alnuem, M. (2014). A hybrid and secure priority-guaranteed mac protocol for wireless body area network. *International Journal of Distributed Sensor Networks*.

[Ullah et al. 2009] Ullah, S., Shen, B., Riazul Islam, S., Khan, P., Saleem, S., and Sup Kwak, K. (2009). A study of MAC protocols for WBANs. *Sensors*, 10(1):128–145.

[Van Dam and Langendoen 2003] Van Dam, T. and Langendoen, K. (2003). An adaptive energy-efficient mac protocol for wireless sensor networks. In *1st International Conference on Embedded Networked Sensor Systems*, pages 171–180. ACM.

[Von Buren et al. 2006] Von Buren, T., Mitcheson, P. D., Green, T. C., Yeatman, E. M., Holmes, A. S., and Troster, G. (2006). Optimization of inertial micropower generators for human walking motion. *Sensors*, 6(1):28–38.

[Watteyne et al. 2007] Watteyne, T., Augé-Blum, I., Dohler, M., and Barthel, D. (2007). Anybody: a self-organization protocol for body area networks. In *Proceedings of the ICST 2nd international conference on Body area networks*, page 6. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Xu et al. 2014] Xu, S., Zhang, Y., Jia, L., Mathewson, K. E., Jang, K.-I., Kim, J., Fu, H., Huang, X., Chava, P., Wang, R., et al. (2014). Soft microfluidic assemblies of sensors, circuits, and radios for the skin. *Science*, 344(6179):70–74.

[Yuce and Ho 2008] Yuce, M. R. and Ho, C. K. (2008). Implementation of body area networks based on mics/wmts medical bands for healthcare systems. In *30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBS)*, pages 3417–3421. IEEE.

[Zhang and Dolmans 2009] Zhang, Y. and Dolmans, G. (2009). A new priority-guaranteed mac protocol for emerging body area networks. In *Fifth International Conference on Wireless and Mobile Communications (ICWMC'09)*, pages 140–145. IEEE.

[Zhen et al. 2008] Zhen, B., Patel, M., Lee, S., Won, E., and Astrin, A. (2008). TG6 technical requirements document (TRD). *IEEE P802.15 Working Group*.

Patrocinador Diamante

GOVERNO
DO RIO GRANDE DO NORTE

Patrocinadores Bronze

ESiG
Software & consultoria

dynavideo

CABOTELECOM

Sistema FIERN

PREFEITURA DO NATAL
A NOSSA CIDADE

PARNAMIRIM
Cuidando de você.

Apoio Financeiro

CAPES

MINISTÉRIO DA EDUCAÇÃO

BRASIL
GOVERNO FEDERAL

CNPq
Conselho Nacional de Desenvolvimento Científico e Tecnológico

nic.br
Núcleo de Informação e Coordenação do Ponto BR

cgi.br
Comitê Gestor da Internet no Brasil